

Answer Set Programming

Piero Bonatti¹, Francesco Calimeri², Nicola Leone², and Francesco Ricca²

¹Dept. of Physical Sciences - Sec. Informatics, University of Naples “Federico II”, I-80126 Napoli, Italy
email: bonatti@na.infn.it

²Dept. of Mathematics, University of Calabria, I-87036 Rende (CS), Italy
e-mail: {calimeri, leone, ricca}@mat.unical.it

Abstract. Answer Set Programming (ASP), referred to also as Disjunctive Logic Programming under the stable model semantics (DLP), is a powerful formalism for Knowledge Representation and Reasoning. ASP has been the subject of intensive research studies, and, also thanks to the availability of some efficient ASP systems, has recently gained quite some popularity and is applied also in relevant industrial projects. The Italian logic programming community has been very active in this area, some ASP results achieved in Italy are widely recognized as milestones on the road to the current state of the art. After a formal definition of ASP, this chapter surveys the main contribution given by the Italian community to the ASP field in the last 25 years.

1 Introduction

Answer Set Programming (ASP), [1–5] referred to also as Disjunctive Logic Programming under the stable model semantics (DLP), is a powerful formalism for Knowledge Representation and Reasoning. Bloomed from the work of Gelfond, Lifschitz [2, 3] and Minker [6–9] in the 1980ies, it has enjoyed a continuously increasing interest within the scientific community. One of the main reasons for the success of ASP is the high expressive power of its language: ASP programs, indeed, allow 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 [10, 11] (i.e., ASP captures the complexity class $\Sigma_2^P = \text{NP}^{\text{NP}}$). Thus, ASP allows us to encode also programs which cannot be translated to SAT in polynomial time. Importantly, ASP is fully declarative (the ordering of literals and rules is immaterial), and the ASP encoding of a large variety of problems is very concise, simple, and elegant [12].

Example 1. To see an elegant ASP encoding, consider 3-Colorability, a well-known NP-complete problem. Given a graph, the problem is to decide whether there exists an assignment of one of three colors (say, red, green, or blue) to each node such that adjacent nodes always have different colors. Suppose that the graph is represented by a set of facts F using a unary predicate $node(X)$ and a binary predicate $arc(X, Y)$. Then, the following ASP program (in combination with F) computes all 3-Colorings (as stable models) of that graph.

$$\begin{aligned} r_1 : & \quad color(X, red) \vee color(X, green) \vee color(X, blue) :- node(X). \\ r_2 : & \quad :- color(X_1, C), color(X_2, C), arc(X_1, X_2). \end{aligned}$$

Rule r_1 expresses that each node must either be colored red, green, or blue;¹ due to minimality of the answer sets models, a node cannot be assigned more than one color. The subsequent integrity constraint checks that no pair of adjacent nodes (connected by an arc) is assigned the same color.

¹ Variable names start with an upper case letter and constants start with a lower case letter.

Thus, there is a one-to-one correspondence between the solutions of the 3-Coloring problem and the answer sets of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set. \square

Unfortunately, the high expressiveness of ASP comes at the price of a high computational cost in the worst case, which makes the implementation of efficient systems a difficult task. Nevertheless, starting from the second half of the 1990ies, and even more in the latest years, a number of efficient ASP systems have been released [13–22], that encouraged a number of applications in many real-world and industrial contexts [23–31]. These applications have confirmed the viability of the ASP exploitation for advanced knowledge-based tasks, and stimulated further research in this field.

The Italian research community produced, in the latest 25 years, a significant contribution in the area, addressing the whole spectrum of issues cited above; this contribution ranged from theoretical results and characterizations [32–37] to practical applications [23, 24, 26–31, 38–43], stepping through language extensions [13, 40, 44–66], evaluation algorithms and optimization techniques [67–76]. Several of the achieved results are widely recognized as milestones on the road to the current state of the art; this is, for instance, the case of the DLV project [13], that produced one of the world leading ASP systems. The Italian community is currently very active on ASP, it contributes in pushing forward the state of the art, as witnessed by the most recent results like, e.g., the ASP extension to deal with infinite domains which is at the frontier of the ASP research [57, 59, 60, 62, 63, 66].

The rest of the Chapter is structured as follows: in Section 2, ASP is formally introduced, syntax and semantics of the language are presented; Section 3 focuses on ASP properties and its theoretical characterizations; Section 4 surveys linguistic extensions; Section 5 reports on ASP with infinite domains; Section 6 first introduces the general architecture of ASP systems, and then surveys algorithms and optimization techniques; Section 7 first describes DLV and number of other ASP-based systems, and then reports on real-world ASP applications; eventually, Section 8 collects a number of further contributions of the Italian ASP community.

2 The ASP Language

In what follows, we provide a formal definition of the syntax and semantics of Answer Set Programming in the spirit of [3].

2.1 Syntax

Following a convention dating back to Prolog, strings starting with uppercase letters denote logical variables, while strings starting with lower case letters denote constants. A *term* is either a variable or a constant.² An *atom* is an expression $p(t_1, \dots, t_n)$, where p is a *predicate* of arity n and t_1, \dots, t_n are terms. A *literal* l is either an atom p (*positive literal*) or its negation $\text{not } p$ (*negative literal*). Two literals are said to be *complementary* if they are of the form p and $\text{not } p$ for some atom p . Given a literal l , $\text{not}.l$ denotes its complementary literal. Accordingly, given a set L of literals, $\text{not}.L$ denotes the set $\{\text{not}.l \mid l \in L\}$. A set L of literals is said to be *consistent* if, for every literal $l \in L$, its complementary literal is not contained in L .

A *disjunctive rule* (*rule*, for short) r is a construct:

$$a_1 \vee \dots \vee a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m. \quad (1)$$

² Note that, as common in ASP, function symbols are not considered unless explicitly specified.

where $a_1, \dots, a_n, b_1, \dots, b_m$ are literals and $n \geq 0, m \geq k \geq 0$. The disjunction $a_1 \vee \dots \vee a_n$ is called the *head* of r , while the conjunction $b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m$ is referred to as the *body* of r . A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a fact, and in this case the “:-” sign is usually omitted. The following notation will be useful for further discussion. If r is a rule of form (1), then $H(r) = \{a_1, \dots, a_n\}$ is the set of literals in the head and $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \dots, b_k\}$ and $B^-(r)$ (the *negative body*) is $\{b_{k+1}, \dots, b_m\}$. An *ASP program* (also called *Disjunctive Logic Program* or *DLP program*) P is a finite set of rules. A not-free program P (i.e., such that $\forall r \in P : B^-(r) = \emptyset$) is called *positive* or *Horn*,³ and a v-free program P (i.e., such that $\forall r \in P : |H(r)| \leq 1$) is called *normal logic program*. In ASP, rules in programs are usually required to be safe. The motivation of safety comes from the field of databases; for a detailed discussion, we refer to [77]. A rule r is *safe* if each variable in r also appears in at least one positive literal in the body of r . An ASP program is safe, if each of its rules is safe, and in the following we will only consider safe programs. A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. Sometimes a ground program is also called *propositional program*.

2.2 Semantics

We next describe the semantics of ASP programs, which is based on the answer set semantics originally defined in [3]. However, different to [3] only consistent answer sets are considered, as it is now standard practice. In ASP the availability of some pre-interpreted predicates is assumed, such as $=, <, >$. However, it would also be possible to define them explicitly as facts, so they are not treated in a special way.

Herbrand Universe and Literal Base. For any program P , the *Herbrand universe*, denoted by U_P , is the set of all constants occurring in P . If no constant occurs in P , U_P consists of one arbitrary constant. The *Herbrand literal base* B_P is the set of all ground literals constructible from predicate symbols appearing in P and constants in U_P .

Ground Instantiation. For any rule r , $Ground(r)$ denotes the set of rules obtained by replacing each variable in r by constants in U_P in all possible ways. For any program P , its ground instantiation is the set $grnd(P) = \bigcup_{r \in P} Ground(r)$. Note that for propositional programs, $P = grnd(P)$ holds.

Answer Sets For every program P , its answer sets are defined by using its ground instantiation $grnd(P)$ in two steps: first the answer sets of positive disjunctive programs are defined, then the answer sets of general programs are defined by a reduction to positive disjunctive programs and a stability condition. An interpretation I is a consistent set of ground literals $I \subseteq B_P$ w.r.t. a program P . A consistent interpretation $X \subseteq B_P$ is called *closed under P* (where P is a positive disjunctive datalog program), if, for every $r \in grnd(P)$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation which is closed under P is also called *model* of P . An interpretation $X \subseteq B_P$ is an *answer set* for a positive disjunctive program P , if it is minimal (under set inclusion) among all (consistent) interpretations that are closed under P .

Example 2. The positive program $P_1 = \{a \vee b \vee c.\}$ has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$; they are minimal and correspond to the multiple ways of satisfying the disjunction. Its extension $P_2 =$

³ In positive programs negation as failure (not) does not occur, while strong negation (\neg) may be present.

$P_1 \cup \{:- a.\}$ has the answer sets $\{b\}$ and $\{c\}$: comparing P_2 with P_1 , the additional constraint is not satisfied by interpretation $\{a\}$. Moreover, the positive program $P_3 = P_2 \cup \{b:- c., c:- b.\}$ has the single answer set $\{b, c\}$ (indeed, the remaining consistent closed interpretation $\{a, b, c\}$ is not minimal). It is easy to see that, $P_4 = P_3 \cup \{:- c.\}$ has no answer set. \square

The *reduct* or *Gelfond-Lifschitz transform* [2, 3] of a ground program P w.r.t. a set $X \subseteq B_P$ is the positive ground program P^X , obtained from P by: (i) deleting all rules $r \in P$ for which $B^-(r) \cap X \neq \emptyset$ holds; (ii) deleting the negative body from the remaining rules. An *answer set* of a program P is a set $X \subseteq B_P$ such that X is an answer set of $grnd(P)^X$.

Example 3. For the negative ground program $P_5 = \{a:- \text{not } b.\}$, $A = \{a\}$ is the only answer set, as $P_5^A = \{a.\}$. For example for $B = \{b\}$, $P_5^B = \emptyset$, and so B is not an answer set. \square

3 Properties and Theoretical Characterizations

The Italian research community provided relevant contributions to the study of ASP and its theoretical characterizations. In this respect, the main results have been achieved by the work in [32]; there, the authors provide: a declarative characterization of answer sets in terms of unfounded sets; a generalization of the well-founded (\mathcal{W}_P) operator to disjunctive logic programs; a fixpoint semantics for function-free programs; an algorithm for answer set computation; a deep analysis of the main computational problems related to the concepts. In this Section, we briefly discuss these contributions.

The definition of unfounded sets for disjunctive logic programs was given as an extension of the analogous concept defined for (disjunction-free) logic programs [78]. As for normal logic programs, they single out the atoms that are (definitely) not derivable from a given program w.r.t. a fixed interpretation; thus, according to the closed-world assumption [79], they single out atoms that can be stated to be false. In a disjunctive logic program \mathcal{P} , the union of unfounded sets for \mathcal{P} may not be an unfounded set for \mathcal{P} ; thus, the existence of the greatest unfounded set (i.e., an unfounded set that contains all other unfounded sets) is not guaranteed as in the case of normal programs. Authors proved that for unfounded-free interpretations (i.e., interpretations that do not contain any unfounded atom), the union of different unfounded sets is an unfounded set; the *greatest unfounded set* of \mathcal{P} w.r.t. I , denoted $GUS_{\mathcal{P}}(I)$, is the union of all unfounded sets.

Several interesting relationships between answer sets and unfounded sets were also discovered, which led to a simple, yet elegant, characterization of answer sets in terms of unfounded sets: disjunctive answer sets coincide with the unfounded-free models of \mathcal{P} , and a model of \mathcal{P} is an answer set iff the set of false atoms coincides with the greatest unfounded set.

Authors of [32] defined also a suitable extension of the well-founded operator \mathcal{W}_P of Van Gelder et al. [78] to the disjunctive case; this allowed to achieve another important result: the definition of a fixpoint semantics for disjunctive answer sets in terms of \mathcal{W}_P . The set of answer sets of \mathcal{P} coincides with the set of the (total) fixpoints of \mathcal{W}_P . By exploiting the theoretical results, the authors designed an algorithm for the computation of the answer set semantics of disjunctive programs. The key idea is that, since answer sets are total interpretations, computing their entire negative portion is superfluous; rather, it is sufficient to restrict the computation to those negative literals that are necessary to derive the positive part. To this end, the notion of *possibly-true literals* is introduced, which plays a crucial role in the computation. The algorithm is based on a controlled search in the space of the interpretations, and the stability of a generated model (answer set candidate) is tested by checking whether it is unfounded-free. This is done by means of a function that runs in polynomial time on *head-cycle-free (HCF)* programs [80, 81]. In the general case, the algorithm for the computation of answer sets runs in polynomial space and single exponential time.

4 Language Extensions

The standard language of ASP has been extended in several ways in order to improve its expressiveness. The Italian community provided contributions regarding two of the most relevant extensions of ASP: *Optimization Constructs* and *Aggregates*.

4.1 Optimization Constructs

The basic ASP language can be used to solve complex search problems, but does not natively provide constructs for specifying optimization problems (i.e. problems where some goal function must be minimized or maximized). The extension conceived for solving optimization problems consists of *Weak Constraints* [44, 13]. In the basic language, constraints represent a condition that *must* be satisfied; for this reason, they are also called *strong* constraints. Contrary to strong constraints, *weak constraints* allow us to express desiderata, that is, conditions that *should* be satisfied. Thus, they may be violated, and their semantics involves minimizing the number of violations. From a syntactic point of view, a weak constraint is like a strong one where the implication symbol :- is replaced by $\text{:}\sim$. The informal meaning of a weak constraint $\text{:}\sim B$ 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 after the constraint enclosed in brackets (by means of positive integers or variables). If not specified, the default value for weight and priority level is 1. In this case, the answer sets are considered which minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level and, among them, are taken those which minimize the sum of weights of the violated weak constraints in the next lower level, and so on.

4.2 Aggregates

There are some simple properties, often arising in real-world applications, which cannot be encoded in a simple and natural manner using ASP [45–48, 82–84]. Especially properties that require the use of arithmetic operators on a set of elements satisfying some conditions (like sum, count, or maximum) require rather cumbersome encodings (often requiring an “external” ordering relation over terms), if one is confined to classic ASP. Similar observations have also been made in related domains, notably database systems, which led to the definition of aggregate functions. Especially in database systems this concept is by now both theoretically and practically fully integrated. When ASP systems started to be used in real applications, the need for aggregates became apparent also here. Hence, ASP has been extended with special atoms handling aggregate functions [45–48]. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant. The most common aggregate functions compute the number of terms, the sum of non-negative integers, and minimum/maximum term in a set. Aggregates are especially useful when real-world problems have to be dealt with.

4.3 Other Extensions

In order to meet requirements of different application domains, ASP has been extended in other directions; thus, there is a number of interesting languages having the roots on ASP.

For instance, ASP has been exploited for defining and implementing *action language* (i.e., a language conceived for dealing with actions and change) \mathcal{K} [49], while, in [50] a framework for *abduction with penalization* has been proposed and implemented as a front-end for the ASP system DLV. Other ASP extensions have been conceived to deal with *Ontologies* (i.e. abstract

models of a complex domain). In particular, in [40] an ASP-based language for ontology specification and reasoning has been proposed, which extends ASP in order to deal with complex real-world entities, like classes, objects, compound objects, axioms, and taxonomies. In [51] an extension of ASP, called *HEX-Programs*, which supports higher-order atoms as well as external atoms has been proposed. External atoms allow one to embed external sources of computation in a logic program. Thus, HEX-programs are useful for various tasks, including meta-reasoning, data type manipulations, and reasoning on top of Description Logics (DL) [85] ontologies. *Template predicates* have been introduced in [52]; they are special intensional predicates defined by means of generic reusable subprograms, which have been conceived for easing coding and improving readability and compactness of programs, and allowing more effective code reusability. An extension of ASP by the introduction of the notion of resource is proposed in [53]. The resulting framework, named RASP, declaratively supports quantitative reasoning on consumption and production of resources. Various forms of preferences, policies, and cost-based criteria can be used to model the processes that produce/consume resources [54]. In [55] standard ASP has been enriched by introducing consistency-restoring rules (cr-rules) and preferences, leading to the CR-Prolog language. Basically, in this language, besides standard ASP rules one may specify CR-rules, that are expressions of the form: $r:a_1 \vee \dots \vee a_n :- {}^+body$ ($n \geq 1$). The intuitive meaning of CR-rules is: if *body* is true then one of a_1, \dots, a_n is “possibly” believed to be true. Importantly, the name of CR-prolog rules can be directly exploited to specify preferences among them. In particular, if the fact $prefer(r_1, r_2)$ is added to a CR-program, then rule r_1 is preferred over rule r_2 . This allows one to encode partial orderings among preferred answer sets by explicitly writing preferences among CR-rules.

In [56] Normal Form Nested (NFN) programs, a non-propositional language similar to Nested Logic Programming (NLP) [86] has been proposed. NFN programs often allow for more concise ASP representations by permitting a richer syntax in rule heads and bodies. It is worth noting that, NFN programs do allow for variables, whereas NLP are propositional. Since with the presence of variables domain independence is no longer guaranteed, the class of safe NFN programs is defined, which are guaranteed to be domain independent. Moreover, it is shown that for NFN programs which are also NLPs, the new semantics coincides with the one of [86]; while keeping the standard meaning of answer sets on ASP programs with variables. Finally, an algorithm which translates NFN programs into ASP programs is provided.

5 ASP with Infinite Domains

The first ASP languages were based on extensions of Datalog, that is, function-free logic programs.⁴ Without function symbols, ASP programs can only reason about finite domains, and have limited data modeling abilities. Such restrictions were motivated by complexity considerations, as answer set reasoning with unrestricted first-order normal programs is Π_1^1 -complete, and hence highly undecidable. However, by introducing suitable alternative syntactic restrictions, it is possible to improve the tradeoff between complexity and expressiveness.

In particular, the introduction of function symbols in ASP languages lead to several benefits [57]: (i) Data encapsulation support, as function symbols are the main logic programming construct for data abstraction [87]; (ii) Enhanced problem solving power, as the class of solvable problems can be extended beyond the second level of the polynomial hierarchy (that is, the class of problems solvable with Disjunctive Datalog with negation); (iii) Support for recursive data structures, such as lists, XML documents, etc. Such data structures are extremely common

⁴ In this section we use the term *function* to refer to uninterpreted functions (or constructors) as in pure logic programming.

in modern applications and functions constitute the most natural way of encoding them; (iv) Simulation and extension of description logics [88]; in this context, function symbols are needed to encode existential quantification through skolemization. Such work is of strategic importance given the important role that description logics play in the semantic web.

The first class of computationally well-behaved ASP programs with function symbols, called *finitary programs*, is due to the Italian logic programming community. They have been introduced in [58], and soon after have been followed by ω -restricted programs [89]. The latter address the challenges of ASP with functions only partially. The answer sets of ω -restricted programs are all finite, and recursion over recursive data structures is not allowed—therefore ω -restricted programs address essentially data encapsulation only. Finitary programs constitute a more ambitious effort, capable of supporting ASP programs with infinite and infinitely many answer sets, and a large class of recursive predicates, including the standard list- and tree-manipulation programs [57].

Finitary programs are characterized by two restrictions. To simplify the presentation here we deal only with normal (i.e. disjunction-free) logic programs—see [59, 60] for an account of disjunctive programs. The first restriction applies to recursion, and is expressed in terms of the notion of *dependency graph* of a program P , whose set of nodes is the Herbrand base B_P . The dependency graph contains a directed edge (A, A') if and only if there exists a rule $r \in \text{grnd}(P)$ such that $A \in H(r)$ and $A' \in B(r)$. The edge is labelled *positive* if $A' \in B^+(r)$, and *negative* if $A' \in B^-(r)$. Then we say that A *depends* on A' if there exists a path from A to A' in the dependency graph.

Now we are ready to formulate the first restriction: a program P is *finitely recursive* iff every atom in the Herbrand base of P depends only on finitely many other ground atoms. Finitely recursive programs enjoy a number of nice theoretical properties proved in [59]:⁵

- they enjoy an analog of the *compactness* property of first-order logic;
- inconsistency checking and skeptical inference are semidecidable;
- the semantics of a finitely recursive program P can be approximated through a chain of finite programs $P_1 \subseteq P_2 \subseteq \dots \subseteq P_i \subseteq \dots \subseteq \text{grnd}(P)$.

The second restriction is based on *odd-cycles*, that are cycles in the dependency graph containing an odd number of negative edges. A normal program is *finitary* iff it is finitely recursive and its dependency graph contains only finitely many odd-cycles.

Finitary programs are very expressive; they comprise a number of useful predicates, including the standard list manipulation predicates, QBF metainterpreters, and programs for reasoning about actions, just to name a few [57]. Moreover, they enjoy very good computational properties [57, 61]. If the set of atoms occurring in an odd-cycle is given, then: (a) ground credulous queries and ground skeptical queries are all decidable; (b) unrestricted ground credulous queries and ground skeptical queries are semidecidable.

Another Italian contribution in this field is the class of *finitely ground* programs [62]. They are characterized by means of an intelligent grounding transformation that turns any given disjunctive program P with functions into an equivalent ground program; P is finitely ground if this transformation yields a finite program. Finitely ground programs—due to the nature of the intelligent grounding—are well-suited for bottom-up evaluation, while finitary programs are naturally well-suited for top-down evaluations. As a consequence finitely ground programs are easier to support in systems like DLV that adopt a bottom-up grounding approach. Finitely ground programs have no restrictions on odd-cycles (and do not need them to be fed to the reasoner as an input). On the other hand, they are required to be safe, which rules out a number of interesting

⁵ Another contribution of the Italian community; best paper award at ICLP 2007.

programs, such as list- and tree-manipulation programs. Moreover, like ω -restricted programs, their semantics is always finite, both in terms of the size and the number of answer sets.

In an interesting recent work [63], however, the duality between the two program classes is starting to be reconciled, by showing how given a positive finitely recursive program P and a query Q one can construct—by a magic set transformation—a finitely ground program P' that yields the same answer to Q as P .

The classes of finitary and finitely ground programs, unfortunately, are not decidable. This result motivated further works aimed at characterizing decidable classes of well-behaved programs with function symbols. The fathers of finitely ground programs introduced *finite domain programs*, a subclass of finitely ground programs that can be effectively recognized [62].

This line of research is having an impact on the activity of other groups outside Italy. In [90], an extension of finite domain programs is proposed. In [88, 91, 92], another family of effectively recognizable, well-behaved programs is investigated. This is a very interesting line of investigation, as it covers description logics, and it may eventually lead to interesting nonmonotonic extensions thereof. Moreover, these works adopt a different strategy for achieving inference decidability, based on a tree-model property and on a reasoning method analogous to blocking.

5.1 Calculi and implementations

Further contributions stemming from the Italian community comprise resolution-based calculi for skeptical and credulous ASP reasoning with function symbols. *Skeptical resolution* [64] consists of five inference rules: resolution, negation as failure, a structural rule for removing successful literals, a rule for detecting contradictions, and a *split* rule for generating new hypotheses and carrying out reasoning by cases. The skeptical resolution calculus is complete for all finitely recursive programs [59]. Recently, a *credulous resolution* calculus [65] has been theoretically studied and experimentally evaluated on a few standard problems with encouraging results that deserve further investigations. The main advantage of resolution calculi is that they need no prior instantiation (grounding) of the input program; instantiation is incremental and on-demand, as in classical resolution. Support for function symbols is also being introduced in DLV for finitely ground programs [66]. We expect it to be soon extended to finitary programs by means of suitable extensions of the magic sets transformation adopted in [63].

5.2 Open issues

ASP with infinite domains is a lively area which is being further developed by several research groups across the world. The main ongoing investigations concern:

- extending the known decidable classes of well-behaved ASP programs;
- the systematic derivation of new classes of well-behaved programs with functions through the composition of modules belonging to known well-behaved classes [93];
- the development and improvement of reasoning mechanisms for ASP with infinite domains;
- the relationships between finitary and finitely ground programs.

6 Algorithms and Optimization Techniques

The general architecture of an ASP system, depicted in Figure 1, helps in understanding the evaluation flow of the computation carried out by a typical ASP system. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system.⁶

⁶ The input is usually read from text files, but some systems also interface to relational databases for retrieving facts stored in relational tables.

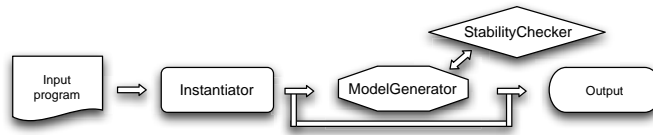


Fig. 1. General architecture of an ASP system.

In general, an input program P contains variables, and the first step of a computation of an ASP system is to eliminate these variables, generating a ground instantiation $grnd(P)$ of P . This variable-elimination process is called *instantiation* of the program (or *grounding*), and is performed by the *Instantiator* module (see Figure 1). A naïve Instantiator would produce the full ground instantiation $grnd(P)$, which is, however, undesirable from a computational point of view, as in general many useless ground rules would be generated. An ASP system, therefore, employs a more sophisticated procedure geared towards keeping the instantiated program as small as possible. A necessary condition is, of course, that the instantiated program must have the same answer sets as the original program; however, it should be noted that the Instantiator solves a problem which is in general EXPTIME-hard, the produced ground program being potentially of exponential size with respect to the input program. Optimizations in the Instantiator therefore often have a big impact, as its output is the input for the following modules, which implement computationally hard algorithms. Moreover, if the input program is normal and stratified, the Instantiator module is, in some cases, able to directly compute its answer sets (if it exists).

The subsequent computations, which constitute the non-deterministic part of an ASP system, are then performed on $grnd(P)$ by both the *Model Generator* and the *Model Checker*. Roughly, the former produces some “candidate” answer set, whose stability is subsequently verified by the latter. Model generation is the non-deterministic core of an ASP system, and it is usually implemented as a backtracking search similar to the Davis-Putnam-Logemann-Loveland (DPLL) procedure [94] for SAT solving. Basically, starting from the empty (partial) interpretation, the *ModelGenerator* module 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 exploiting the *Model Checker* module; whereas, if an inconsistency is detected chosen literals have to be undone (backtracking). For disjunctive programs model checking is as hard as the problem solved by the Model Generator, while it is trivial for non-disjunctive programs. Finally, once an answer set has been found, ASP systems typically print it in text format, and possibly the *Model Generator* resumes in order to look for further solutions.

All the aspects of the evaluation of ASP programs have been subject of analysis by the Italian research community; the obtained results, divided by evaluation task, are surveyed in the following.

Instantiation. The first contributions in this respect dates back to 1999, when some optimization techniques, based on a rewriting of the input program, have been proposed aiming at reducing the size of the instantiation generated by the grounder [67]. 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, in [68], a new join-ordering technique was proposed that sensibly improves the instantiation procedures of ASP systems. Some year later, in [69] a new backjumping technique for the instantiation of a rule has been proposed which allows for reducing both the size of the generated grounding and the time needed for producing it. All the above mentioned techniques have been incorporated in the grounder of the DLV system, and allowed for relevant improvements of the performance of the system. Notably, to our

knowledge, the technique in [69] has been successfully exploited also by other two grounders, namely *GrinGo* [95], and GIDL [96]. In the last few years, in order to exploit the power of modern multi-core/multiprocessor computers, a number of strategies for the parallelization of the instantiation procedure have been proposed [70, 71]. In particular, three levels of parallelism can be exploited during the instantiation process, namely, components, rules and single rule level. The first two levels were first employed in [70] while the third one was presented in [71]. Also these techniques have been implemented into the DLV grounder, and the resulting parallel instantiator proved to be effective on modern multi-core machines when handling both real-world and classical problem instances [70, 71]. Moreover, a distributed instantiator working on a Beowulf [97] cluster was presented in [98].

Model Generation. The Italian research community provided relevant contribution regarding all the aspects of model generation. About the propagation step, peculiar properties of ASP programs have been exploited in [72, 99], that allow to prune the search space by combining extension of the well-founded operator for disjunctive programs with a number of techniques based on disjunctive ASP program properties. The efficiency of the whole model generation process depends also on two other 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 [73] and look-back [74, 75] techniques and heuristics specifically conceived for enhancing the model generation process were proposed and implemented in the state-of-the-art ASP system DLV [13]. In a lookahead heuristic [73] 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 [73] “layers” several criteria based on peculiar properties of ASP, and basically drive the search towards “supported” interpretations (since answer sets are supported interpretations (cfr. [32, 100, 101])). In a look-back heuristics usually choices are made in such a way that the atoms most involved in conflicts are chosen first. Motivated by heuristics implemented in SAT solvers like Chaff [102], a family of new look-back heuristics tailored for disjunctive ASP programs has been proposed in [75]. Look-back heuristics are mainly exploited 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 “entail” the inconsistency is undone. In [74] a *reason calculus* that allows for determining the relevance for an inconsistency has been proposed; here the information about the choices (“reasons”) whose truth-values have caused truth-values of other deterministically derived atoms is collected and exploited for backjumping.

Native ASP systems exploit backtracking search algorithms that work directly on the ground instantiation of the input program, like the ones described above. An alternative approach to model generation is based on a rewriting into a propositional formula which is then evaluated by a boolean satisfiability solver for finding answer sets. Giunchiglia and Maratea, in collaboration with the members of the Texas Action Group at Austin, led by Prof. Vladimir Lifschitz, designed a SAT-based approach to normal logic programs [103, 104, 18], which is now considered the reference SAT-based work in ASP. A comparison among the techniques employed by ASP systems underlying strengths and weaknesses of each approach was provided in [105, 106]. Going beyond the classical methods of computing the answer sets of a logic program in [107] a method is presented that does not require a preliminary grounding phase. Moreover, the techniques for parallel evaluation of ground ASP programs on clusters were studied in [98].

Model Checking is a crucial step of the computation of the answer sets. There are two main reasons 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-NPcomplete [11]. In [76] a new transformation \mathcal{T} , which reduces stable model checking to UNSATi.e., to deciding whether a given CNF formula is unsatisfiable, is introduced. Thus, the stability of an answer set candidate M of a program P can be verified by calling a SAT solver on the CNF formula obtained by applying \mathcal{T} to P . The transformation is very efficient: it runs in logarithmic space and no new symbol is added. This approach to model checking was implemented in the ASP system DLV [13] and some experiments confirmed its efficacy [76].

7 Systems and Applications

Several ASP systems are available nowadays, and a number of practically relevant real-world applications of ASP have been developed. In the following, we first present DLV [13], a state-of-the-art ASP systems, which is widely used all over the world and is actively developed by Italian researchers; then we mention some relevant systems and application based on ASP.

7.1 The DLV System

The DLV system [13] is widely considered one of the state-of-the-art implementations of answer set programming. The development of DLV started at the end of 1996, within a research project funded by the Austrian Science Funds (FWF) and led by Nicola Leone at the Vienna University of Technology. The first stable release became available in 1997, and at present, DLV is the subject of an international cooperation between the University of Calabria and the Vienna University of Technology. After its first release, the DLV system has been significantly improved over and over in the last years. In particular, the language of DLV was enriched in several ways and currently supports the main ASP extensions: disjunction, aggregates, weak-constraints, and function symbols (see Section 4 and Section 5). Relevant 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 (see Section 6). Moreover, in order to deal with data-intensive applications a database oriented version of DLV, called DLV^{DB} , has been recently proposed [108, 109]. DLV^{DB} is able to evaluate large amount of data by exploiting an evaluation strategy working mostly onto the database, where input data reside. DLV^{DB} embodies some query-oriented optimization strategies, like magic-sets [42], capable of significantly improving query evaluation performances. As a result, at the time being, DLV is generally recognized to be a state-of-the-art implementation of disjunctive ASP. Importantly, DLV is widely used by researchers all over the world, it is employed in real-world applications (see next Section), and it is competitive from the viewpoint of efficiency with the most advanced systems in the area of Answer Set Programming [110, 111].

7.2 ASP-based Products

In this section three industrial products strongly based on ASP, and, in particular, on DLV are presented, namely: OntoDLV [39, 40], OLEX [28, 29], $H_{iL}EX$ [30, 31].

- OntoDLV [39, 40] is a system for ontologies specification and reasoning. The language of OntoDLV 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) [112], and it is endowed with a robust persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates DLV [13].

- OLEX [28, 29] (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 exploits a reasoning-based approach to text classification which synergically 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. Logic rules, indeed, provides a natural and powerful way to encode how document contents may relate to ontology concepts.

- $H_{iL}E_{iX}$ [30, 31] is an advanced system for ontology-based information extraction from semi-structured and unstructured documents. $H_{iL}E_{iX}$ implements a semantic approach to the information extraction problem able to deal with different document formats (html, pdf, doc, ...). $H_{iL}E_{iX}$ 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. Each descriptor allows us to describe: (i) an ontology object in order to recognize it in a document; or (ii) how to “generate” a new object that, in turn, may be added in the original ontology. The obtained specification is rewritten in ASP and evaluated by means of the DLV system.

7.3 Applications

We briefly illustrate here a number of real-world applications based on DLV or on DLV-based products. They can be grouped in two classes: industrial applications of DLV (developed by the company Exeura s.r.l) and other applications [38].

Industrial Applications. The main commercial applications exploiting DLV are the following:

- *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. 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 [23] 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 $H_{iL}E_{iX}$. 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. The result is a system that combines the speed of computers with the knowledge of a travel agent.

- *Automatic Itinerary Search.* In this application, a web portal conceived for better exploiting the whole transportation system of the Italian region Calabria, including both public and private companies. The system is very precise, it tells you where and what time to catch your bus/train, where to get off and transfer, how long your trip will take, walking directions etc. A set of specifically devised ASP programs are used to build the required itineraries.

- *e-Government.* In this field, an application of the OLEX system was developed, in which legal acts and decrees issued by public authorities are classified. The system was validated with the help of the employees of the Calabrian Region administration, and it performed very well by obtaining an f-measure of 92% and a mean precision of 96% in real-world documents.

- *e-Medicine.* OLEX was employed for developing a system able to classify automatically 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 employed by the personnel of the ULSS of Asolo.

Other Applications. The European Commission funded a project on Information Integration, which produced a sophisticated and efficient data integration system, called INFOMIX, which uses DLV at its computational core [26]. The powerful mechanisms for database interoperability, together with magic sets [41, 42] and other database optimization techniques, which are implemented in DLV, make DLV very well-suited for handling information integration tasks. And DLV (in INFOMIX) was successfully employed to develop in a real-life integration system for the information system of the University of Rome “La Sapienza” The DLV system has been experimented also with an application for Census Data Repair [27], in which errors in census data are identified and eventually repaired.

DLV has been employed at CERN, the European Laboratory for Particle Physics, for an advanced deductive database application that involves complex knowledge manipulation on large-sized databases.

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, which is used by the Polish Securities and Exchange Commission.

In the area of self-healing Web Services, moreover, DLV is exploited for implementing the computation of minimum cardinality diagnoses [43].

In [113] MASEL, A Multi Agent System for E-Learning and Skill Management has been proposed. In MASEL personalized learning paths are automatically composed by exploiting suitable ASP programs run on the DLV system. A prototype tool implementing MASEL using JADE (Java Agent DEvelopment Framework) was developed.

In [114] a complete on-line exam taking portal has been described, called EXAM. The system allows teachers and students to be assisted in the whole process of assessment test building, exam taking, and test correction. The system exploits ASP for automatically generating assessment tests based on user defined constraints: a teacher is made able to build up an assessment test template; her preferences are then translated into a logic specification executable by DLV.

8 Further Contributions

This Section briefly mentions several other contributions to the ASP field due to the work of Italian researchers.

In [115, 116], an integrated information retrieval agent based on an ASP inference engine, named GSA_2 , has been presented. The GSA_2 approach is general and reusable, and the result constitutes a good example of real implementation of agents based on logics.

The first purely syntactic characterization of answer sets in the context of logic programming has been introduced in [33]. In the same work, it has been pointed out explicitly that answer sets are supersets of the well-founded model (wfm) and can thus be in principle computed after a simplification w.r.t. the wfm (this property was independently discovered in [117]). In [34], the authors introduced a graphical representation of ASP programs, called Extended Dependency Graph (EDG). EDG is defined on a simplified form of programs called *kernel*. In [35, 36], kernel programs have been exploited for defining an algorithm for answer set computation, as answer sets can be characterized as *admissible colorings* of the EDG. Moreover, the kernel normal form and other normal forms of ASP programs have been studied in [37]. In [118], some features that graph representations of ASP programs should exhibit, especially isomorphism between a program and the corresponding graph, have been identified. It turns out that isomorphism is

possible only if the graph representation formalism is able to distinguish the cycles occurring in the program, and the different connections among them. Investigating the program structure is also important for understanding the effects of updates of given program on the existence, the number and the content of answer sets. In particular, a graph representation can be useful to understand what happens after asserting lemmas [119] and/or adding new rules [120]. The work [121] showed that representations like the EDG (or others that have been proposed in the literature), which are oriented to atoms and rules, can be usefully condensed into more compact representations, called *Cycle Graph*, which is oriented to components. In the Cycle Graph, vertices are not atoms or rules, but significant subprograms. The Cycle Graph allows the relationship between the syntax of programs and the existence of answer sets to be investigated, and thus can be the basis of software engineering methodologies for answer set programming. In [122] inconsistency and incompleteness in data integration are handled by introducing an “helper model” acting as a mediator between the global conceptual data model and the data sources.

ASP has been exploited as a core inference engine for a system for qualitative management of probabilistic uncertainty [123–125]. The system supports basic reasoning tasks by mechanizing various notions of comparative preference notions that represent plausible models of cognitive unconscious humans mental processes.

ASP has been integrated with arithmetic and finite domain constraint solvers in [126]. The benefits, besides enhanced expressiveness, comprise reduced memory requirements because the part of a program involving constraints needs not be instantiated. Consequently, it was possible to extend significantly the size of the problems solved by an ASP planner for Space Shuttle operations.

The mutual interdependence of ASP-based agents has been investigated [127–130] at Università Mediterranea di Reggio Calabria. In [127], agreements possibly reached by a collection of agents are represented. In [128, 129], a community of agents where individual’s conclusions rely on others’ ones is modeled by nested social predicates. This language is refined in [130] by adding social aggregates and a form of reasoning where models include also “unfounded” interpretations in case they are mutually supported by multiple agents. Finally, a form of preferences under uncertainty is modeled under ASP in [131].

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* **9** (1991) 365–385
4. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proceedings of the 16th International Conference on Logic Programming (ICLP’99), Las Cruces, New Mexico, USA, The MIT Press (November 1999) 23–37
5. 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*. Springer Verlag (1999) 375–398
6. Minker, J., ed.: *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., Washington DC (1988)
7. Minker, J., Rajasekar, A.: A Fixpoint Semantics for Disjunctive Logic Programs. *Journal of Logic Programming* **9**(1) (July 1990) 45–74
8. Lobo, J., Minker, J., Rajasekar, A.: *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts (1992)

9. Fernández, J.A., Minker, J.: Semantics of Disjunctive Deductive Databases. In: Proceedings 4th Intl. Conference on Database Theory (ICDT-92), Berlin (October 1992) 21–50
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. *ACM Transactions on Database Systems* **22**(3) (September 1997) 364–418
11. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* **33**(3) (2001) 374–425
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers (2000) 79–103
13. 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) (July 2006) 499–562
14. Simons, P.: Smodels Homepage (since 1996) <http://www.tcs.hut.fi/Software/smodels/>.
15. Simons, P., Niemelä, I., Soinen, T.: Extending and Implementing the Stable Model Semantics. *Artificial Intelligence* **138** (June 2002) 181–234
16. Zhao, Y.: ASSAT homepage (since 2002) <http://assat.cs.ust.hk/>.
17. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002), Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
18. Babovich, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. Available on <http://www.cs.utexas.edu/users/tag/cmodels.html> (2003)
19. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann Publishers (January 2007) 386–392
20. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. Technical Report cs.AI/0303009, arXiv.org (March 2003)
21. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*. Volume 3662 of *Lecture Notes in Computer Science*., Springer Verlag (September 2005) 447–451
22. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In Brewka, G., Lang, J., eds.: *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, Sydney, Australia, AAAI Press (2008) 422–432
23. 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)*. Volume 5753 of *Lecture Notes in Computer Science*., Springer (2009) 368–381
24. Leone, N., Ricca, F., Terracina, G.: An ASP-Based Data Integration System. In Erdem, E., Lin, F., Schaub, T., eds.: *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*. Volume 5753 of *Lecture Notes in Computer Science*., Springer (2009) 528–534
25. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: An ASP-Based Data Integration System. In Erdem, E., Lin, F., Schaub, T., eds.: *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*. Volume 5753 of *Lecture Notes in Computer Science*., Springer (2009) 591–597
26. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kařka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszki, 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), Baltimore, Maryland, USA, ACM Press (June 2005) 915–917
27. Franconi, E., Palma, A.L., Leone, N., Perri, S., Scarcello, F.: Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*. Volume 2250 of *Lecture Notes in Computer Science*., Springer (2001) 561–578

28. 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. Volume 3229 of Lecture Notes in Computer Science., Lisbon, Portugal (2004) 722–725
29. 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, ACM (2007) 409–416
30. 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). Volume 3 of Lecture Notes in Business Information Processing. (2008) 194–209
31. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In de Vos, M., Proveti, A., eds.: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (July 2005) 248–262
32. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation* **135**(2) (June 1997) 69–112
33. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. *Theoretical Computer Science* **149** (1995) (preliminary version in Proc. of LPNMR93).
34. Brignoli, G., Costantini, S., D’Antona, O., Proveti, A.: Characterizing and computing stable models of logic programs: the non-stratified case. In: Proc. of the 1999 Conference on Information Technology, Bhubaneswar, India. (1999)
35. Bertoni, A., Grossi, G., Proveti, A., Kreinovich, V., Tari, L.: The prospect for answer set computation by a genetic model. In: Proc. of the AAAI Spring Symposium ASP 2001, AAAI press (2001) 1–5
36. Grossi, G., Marchi, M., Pontelli, E., Proveti, A.: Improving the adjsolver algorithm for asp kernel programs. In Costantini, S., Watson, R., eds.: Proc. of ASP2007, 4th International Workshop on Answer Set Programming at ICLP07. (2007)
37. Costantini, S., Proveti, A.: Normal forms for answer sets programming. *J. on Theory and Practice of Logic Programming* **5**(6) (2005)
38. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009). Volume 5753 of Lecture Notes in Computer Science., Springer (2009) 591–597
39. 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)
40. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV⁺ System. *Journal of Applied Logics* **5**(3) (2007) 545–573
41. 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. Volume 3132 of Lecture Notes in Computer Science. (2004) 371–385
42. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences* **73**(4) (2007) 584–609
43. Friedrich, G., Ivanchenko, V.: Diagnosis from first principles for workflow executions. Technical report, Alpen Adria University, Applied Informatics, Klagenfurt, Austria (2008) http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf.
44. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering* **12**(5) (2000) 845–860
45. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (August 2005) 406–411
46. Dell’Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003, Acapulco, Mexico, Morgan Kaufmann Publishers (August 2003) 847–852
47. Faber, W., Leone, N.: On the Complexity of Answer Set Programming with Aggregates. In Baral, C., Brewka, G., Schlipf, J.S., eds.: Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR 2007. Volume 4483 of Lecture Notes in AI (LNAI), Tempe, AZ, USA, Springer Verlag (May 2007) 97–109

48. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In Alferes, J.J., Leite, J., eds.: *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*. Volume 3229 of *Lecture Notes in AI (LNAI)*., Springer Verlag (September 2004) 200–212
49. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic* **5**(2) (April 2004) 206–263
50. Perri, S., Scarcello, F., Leone, N.: Abductive Logic Programs with Penalization: Semantics, Complexity and Implementation. *Theory and Practice of Logic Programming* **5**(1–2) (2005) 123–159
51. 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*, Edinburgh, UK (August 2005) 90–96
52. Calimeri, F., Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: A system with template answer set programs. In: *JELIA*. (2004) 693–697
53. Costantini, S., Formisano, A.: Answer set programming with resources. *Journal of Logic and Computation* (2009) To appear. Draft available as Report-16/2008 of Dip. di Matematica e Informatica, Univ. di Perugia: www.dipmat.unipg.it/formis/papers/report2008_16.ps.gz.
54. Costantini, S., Formisano, A.: Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of Algorithms in Cognition, Informatics and Logic* **64**(1) (2009) 3–15
55. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In P. Doherty, J. McCarthy, M.W., ed.: *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*. (2003)
56. Bria, A., Faber, W., Leone, N.: Normal form nested programs. *Fundamenta Informaticae* (2009) Accepted for publication.
57. Bonatti, P.A.: Reasoning with infinite stable models. *Artif. Intell.* **156**(1) (2004) 75–111
58. Bonatti, P.: Reasoning with infinite stable models. In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, Morgan Kaufmann (2001) 603–610
59. Baselice, S., Bonatti, P.A., Crisculo, G.: On finitely recursive programs. *TPLP* **9**(2) (2009) 213–238
60. Bonatti, P.A.: Reasoning with infinite stable models II: Disjunctive programs. In: *Proceedings of the 18th International Conference on Logic Programming, ICLP 2002*. Number 2401 in *LNCS*, Springer (2002) 333–346
61. Bonatti, P.A.: Erratum to: Reasoning with infinite stable models [artificial intelligence 156 (1) (2004) 75–111]. *Artif. Intell.* **172**(15) (2008) 1833–1835
62. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in asp: Theory and implementation. [132] 407–424
63. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Magic sets for the bottom-up evaluation of finitely recursive programs. [133] 71–86
64. Bonatti, P.A.: Resolution for skeptical stable model semantics. *J. Autom. Reasoning* **27**(4) (2001) 391–421
65. Bonatti, P.A., Pontelli, E., Son, T.C.: Credulous resolution for answer set programming. In Fox, D., Gomes, C.P., eds.: *AAAI*, AAAI Press (2008) 418–423
66. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An asp system with functions, lists, and sets. [133] 483–489
67. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Non-monotonic Reasoning. In INAP Organizing Committee, ed.: *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDL’99)*, Prolog Association of Japan (September 1999) 135–139
68. 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. Volume 2173 of *Lecture Notes in AI (LNAI)*., Springer Verlag (September 2001) 280–294
69. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence* **51**(2–4) (2007) 195–228

70. 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) (2008) 34–54
71. Vescio, S., Perri, S., Ricca, F.: Efficient Parallel ASP Instantiation via Dynamic Rewriting. In: *Proceedings of the First Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2008)*, Udine, Italy (2008)
72. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Disjunctive Logic Programming Systems. *Fundamenta Informaticae* **71**(2–3) (2006) 183–214
73. 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) (2007) 229–266
74. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. *AI Communications – The European Journal on Artificial Intelligence* **19**(2) (2006) 155–172
75. 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) (2008) 70–89
76. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. *Artificial Intelligence* **15**(1–2) (December 2003) 177–212
77. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*. Addison-Wesley (1995)
78. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. *Journal of the ACM* **38**(3) (1991) 620–650
79. Reiter, R.: On Closed World Data Bases. In Gallaire, H., Minker, J., eds.: *Logic and Data Bases*. Plenum Press, New York (1978) 55–76
80. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence* **12** (1994) 53–87
81. Ben-Eliyahu, R., Palopoli, L.: Reasoning with Minimal Models: Efficient Algorithms and Applications. In: *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*. (1994) 39–50
82. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In Codognet, P., ed.: *Proceedings of the 17th International Conference on Logic Programming*, Springer Verlag (2001) 212–226
83. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. *Journal of the ACM* **48**(4) (2001) 880–907
84. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. *Theory and Practice of Logic Programming* **7**(3) (2007) 301–353
85. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press (2003)
86. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence* **25**(3–4) (1999) 369–389
87. Sterling, L., Shapiro, E.: *The Art of Prolog* (2nd ed.). MIT Press (1994)
88. Simkus, M., Eiter, T.: FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. In: *Proc. of LPAR'07*. Volume 4790 of *Lecture Notes in Computer Science.*, Springer (2007) 514–530
89. Syrjänen, T.: Omega-restricted logic programs. In: *Proc. LPNMR'01*. Volume 2173 of *Lecture Notes in Computer Science.*, Springer (2001) 267–279
90. Lierler, Y., Lifschitz, V.: One more decidable class of finitely ground programs. [134] 489–493
91. Eiter, T., Ortiz, M., Simkus, M.: Reasoning using knots. In Cervesato, I., Veith, H., Voronkov, A., eds.: *LPAR*. Volume 5330 of *Lecture Notes in Computer Science.*, Springer (2008) 377–390
92. Simkus, M.: Fusion of logic programming and description logics. [134] 551–552
93. Baselice, S., Bonatti, P.A.: Composing normal programs with function symbols. [132] 425–439
94. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. *Communications of the ACM* **5** (1962) 394–397
95. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In Baral, C., Brewka, G., Schlipf, J.S., eds.: *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '07)*. Volume 4483., Tempe, AZ, USA (2007) 266–271

96. Wittocx, J., Mariën, M., Denecker, M.: GidL: A grounder for FO+. In Thielscher, M., Pagnucco, M., eds.: Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning. (September 2008) 189–198
97. Beowulf.org: The Beowulf Cluster Site <http://www.beowulf.org>.
98. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing* **31**(6) (2005) 608–647
99. 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). Volume 1730 of Lecture Notes in AI (LNAI), El Paso, Texas, USA, Springer Verlag (December 1999) 177–191
100. 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, MIT Press (1989) 600–617
101. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. *Journal of Logic Programming* **19/20** (1994) 73–148
102. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, ACM (June 2001) 530–535
103. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning* **36**(4) (2006) 345–377
104. Giunchiglia, E., Lierler, Y., Maratea, M.: A sat-based polynomial space algorithm for answer set programming. In Delgrande, J.P., Schaub, T., eds.: Proceedings of the 10th International Workshop on Non-Monotonic Reasoning (NMR'2004). (2004) 189–196
105. Giunchiglia, E., Maratea, M.: On the relation between ASP and SAT procedures (or, between smodels and cmodels). In: Proc. of the Twenty-first International Conference on Logic Programming (ICLP). Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 37–51
106. Giunchiglia, E., Leone, N., Maratea, M.: On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence* **53**(1-4) (2008) 169–204
107. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Answer set programming with constraints using lazy grounding. [134] 115–129
108. 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** (2008) 129–165
109. Terracina, G., De Francesco, E., Panetta, C., Leone, N.: Enhancing a DLP system for advanced database applications. In: Proceedings of the International Conference on Web Reasoning and Rule Systems (RR 2008), Karlsruhe, Germany, Springer Verlag (2008)
110. 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.: Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07. Volume 4483 of Lecture Notes in Computer Science., Tempe, Arizona, Springer Verlag (May 2007) 3–17
111. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14–18, 2009. Volume 5753., springer (2009) 637–654
112. 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). (2007) 56–70
113. Garro, A., Palopoli, L., Ricca, F.: Exploiting agents in e-learning and skills management context. *AI Communications – The European Journal on Artificial Intelligence* **19**(2) (2006) 137–154
114. Ianni, G., Ricca, F., Panetta, C.: Specification of Assessment-Test Criteria through ASP Specification. In: Answer Set Programming: Advances in Theory and Implementation, Bath, UK, Research Press International, P.O. Box 144, Bristol BS 1YA (2005) 293–302
115. Ianni, G., Calimeri, F., Lio, V., Galizia, S.: Reasoning about the semantic web using answer set programming. In: APPIA-GULP-PRODE. (2003) 324–336

116. Ianni, G., Ricca, F., Calimeri, F., Lio, V., Galizia, S.: An agent system reasoning about the web and the user. In: WWW (Alternate Track Papers & Posters). (2004) 492–493
117. Subrahmanian, V., Nau, D., Vago, C.: Wfs + branch and bound = stable models. *IEEE Transactions on Knowledge and Data Engineering* **7**(3) (1995) 362–377
118. Costantini, S.: Comparing different graph representations of logic programs under the answer set semantics. In: Proc. of the AAAI Spring Symposium “Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning”. (2001) Stanford, CA.
119. Costantini, S., Lanzarone, G.A., Magliocco, G.: Asserting lemmas in the stable model semantics. In Maher, M., ed.: *Logic Programming – Proc. of the 1996 Joint International Conference and Symposium*, The MIT Press, USA (1996)
120. Costantini, S., Intrigila, B., Proveti, A.: Coherence of updates in answer set programming. In Brewka, G., Peppas, P., eds.: *Proc. of the IJCAI-2003 Workshop on Nonmonotonic Reasoning, Action and Change, NAC03*. (2003) 66–72
121. Costantini, S.: On the existence of stable models of non-stratified logic programs. *J. on Theory and Practice of Logic Programming* **6**(1-2) (2006)
122. Costantini, S., Formisano, A., Omodeo, E.G.: Mappings between domain models in answer set programming. In De Vos, M., Proveti, A., eds.: *Answer Set Programming, Advances in Theory and Implementation, Proc. of the 2nd Intl. ASP’03*. Volume 78 of *CEUR Workshop Proc.* (2003)
123. Capotorti, A., Formisano, A.: Comparative uncertainty: theory and automation. *Mathematical Structures in Computer Science* **18**(1) (2008)
124. Capotorti, A., Formisano, A., Murador, G.: Qualitative uncertainty orderings revised. *Electronic Notes in Theoretical Computer Science* **169** (2007) 43–59
125. Capotorti, A., Formisano, A.: Management of uncertainty orderings through ASP. In Bouchon-Meunier, B., Coletti, G., Yager, R.R., eds.: *Modern Information Processing: From Theory to Applications*. Elsevier (2004) ISBN: 0-444-52075-9.
126. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In Gabbrielli, M., Gupta, G., eds.: *ICLP*. Volume 3668 of *Lecture Notes in Computer Science.*, Springer (2005) 52–66
127. Buccafurri, F., Gottlob, G.: Multiagent Compromises, Joint Fixpoints, and Stable Models. In: *Computational Logic: Logic Programming and Beyond*. (2002) 561–585
128. Buccafurri, F., Caminiti, G.: A Social Semantics for Multi-agent Systems. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR’05, Diamante, Italy*. Volume 3662 of *Lecture Notes in Computer Science.*, Springer Verlag (September 2005) 317–329
129. Buccafurri, F., Caminiti, G.: Logic programming with social features. *TPLP* **8**(5–6) (2008) 643–690
130. Buccafurri, F., Caminiti, G., Laurendi, R.: A Logic Language with Stable Model Semantics for Social Reasoning. In: *ICLP*. (2008) 718–723
131. Buccafurri, F., Caminiti, G., Rosaci, D.: Logic Programs with Multiple Chances. In: *ECAI*. (2006) 347–351
132. de la Banda, M.G., Pontelli, E., eds.: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*. In de la Banda, M.G., Pontelli, E., eds.: *ICLP*. Volume 5366 of *Lecture Notes in Computer Science.*, Springer (2008)
133. Erdem, E., Lin, F., Schaub, T., eds.: *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*. In Erdem, E., Lin, F., Schaub, T., eds.: *LPNMR*. Volume 5753 of *Lecture Notes in Computer Science.*, Springer (2009)
134. Hill, P.M., Warren, D.S., eds.: *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*. In Hill, P.M., Warren, D.S., eds.: *ICLP*. Volume 5649 of *Lecture Notes in Computer Science.*, Springer (2009)