Answer Set Programming

Piero Bonatti[1], Francesco Calimeri[2], Nicola Leone[2], and Francesco Ricca[2]

[1] Dept. of Phisical Sciences - Sec. Informatics, University of Naples "Federico II", I-80126 Napoli, Italy
bonatti@na.infn.it
[2] Dept. of Mathematics, University of Calabria, I-87036 Rende (CS), Italy
{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),[1] is a powerful formalism for Knowledge Representation and Reasoning.[2] 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 $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$). 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 [1, 12–15].

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

[1] Stable models are also named *answer sets*.

[2] A lot of work has been done by the Italian research community both in the broader field of knowledge representation and non-monotonic reasoning, and in the related field of logic languages for databases. We refer the reader to Chapter 4 and Chapter 9, respectively, for a detailed description of the italian contributions in these specific fields which are closely related and partially overlapping with the ASP contributions.

A. Dovier, E. Pontelli (Eds.): 25 Years of Logic Programming, LNCS 6125, pp. 159–182, 2010.

assignment of one out 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.

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

Rule r_1 expresses that each node must either be colored red, green, or blue;[3] due to minimality of the stable 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.

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. □

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 [16–25], that encouraged a number of applications in many real-world and industrial contexts [26–33, 40]. 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 [34–39] to practical applications [26–33, 40–45], stepping through language extensions [16, 42, 46–68], evaluation algorithms and optimization techniques [69–78]. 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 [16], 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 [59, 61, 62, 64, 65, 68].

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.

[3] Variable names start with an upper case letter and constants start with a lower case letter.

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. [4] An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A literal l is either an atom p (*positive* literal) or its negation `not` p (*negative* literal). A set L of literals is said to be *consistent* if, for every positive literal $l \in L$, its complementary literal `not` l is not contained in L.

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

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k, \texttt{ not } b_{k+1}, \cdots, \texttt{ not } b_m. \tag{1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is called the *head* of r, while the conjunction $b_1, ..., b_k,$ `not` $b_{k+1}, ...,$ `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. If r is a rule of form (1), then $H(r) = \{a_1, \ldots, a_n\}$ 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 $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$. 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*, and a `not`-free program P (i.e., such that $\forall r \in P, |H(r)| \leq 1$) is called *normal logic program*.

In ASP, rules are usually required to be safe; the motivation comes from the field of databases, and for a detailed discussion we refer to [79]. 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; a ground program is also called *propositional*.

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.

[4] Note that, as common in ASP, function symbols are not considered unless explicitly specified (see Section 5).

Herbrand Universe and Herbrand 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 Base* B_P is the set of all ground atoms 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 for a program P is a set of ground atoms $I \subseteq B_P$. Let P be a positive program. An interpretation $X \subseteq B_P$ is called *closed under* P 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 program P, if it is minimal (under set inclusion) among all 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 $P_2 = P_1 \cup \{ :\!- a.\}$ 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 $\{b, c\}$. It is easy to see that, $P_4 = P_3 \cup \{ :\!- c\}$ has no answer set. □

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 $P_5 = \{a :\!-$ `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. □

3 Properties and Theoretical Characterizations

The Italian research community provided relevant contributions to the study of ASP and its theoretical characterizations. In this respect, a relevant bunch of results has been achieved by the work in [34], which has given the theoretical foundation for realization of the ASP system DLV system [16]. There, the authors provide: a declarative characterization of answer sets in terms of unfounded sets; a generalization of the well-founded ($\mathcal{W}_\mathcal{P}$) operator to disjunctive logic programs; a fixpoint semantics for function-free programs; an algorithm for answer set computation; an in-depth analysis of the main computational problems related to the concepts. In the 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 [80]. As for normal logic programs, unfounded sets 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 [81], 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. The authors proved that for unfounded-free interpretations (i.e., interpretations that do not contain any unfounded atom), the union of different unfounded sets is guaranteed to be an unfounded set even in the disjunctive case; 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: the answer sets of a disjunctive program $\mathcal{P}$ 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.

The authors of [34] defined also a suitable extension of the well-founded operator $\mathcal{W}_{\mathcal{P}}$ of Van Gelder et al. [80] to the disjunctive case; this allowed to achieve another important result: the definition of a fixpoint semantics for disjunctive answer sets in terms of $\mathcal{W}_{\mathcal{P}}$. The set of answer sets of $\mathcal{P}$ coincides with the (total) fixpoints of $\mathcal{W}_{\mathcal{P}}$. 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, implemented by a backtracking technique; 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 [82, 83]. 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 it does not natively provide constructs for specifying optimization problems (i.e. problems where some goal function must be minimized or maximized). 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*, introduced in [16, 46], allow one to express desiderata, that is, conditions that *should* be satisfied; their semantics involves minimizing the number of violations, thus allowing to easily encode optimization problems. From a syntactic point of view, a weak constraint is like a strong one, where the implication symbol :- is replaced by $:\sim$. 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 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. 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, 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 [47–50, 84–86]. 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, which led to the definition of aggregate functions. Especially in database systems this concept is at present both theoretically and practically fully integrated. When ASP systems started to be used in real applications, the need for aggregates become apparent also here. Hence, ASP has been extended with special atoms handling aggregate functions [47–50, 87, 88]. 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 was extended in other directions; thus, there is a number of interesting languages having the roots on ASP.

For instance, ASP was exploited for defining and implementing the *action language* (i.e., a language conceived for dealing with actions and change) $\mathcal{K}$ [51], while, in [52] a framework for *abduction with penalization* was proposed and implemented as a front-end for the ASP system DLV. Other ASP extensions were conceived to deal with *Ontologies* (i.e. abstract models of a complex domain). In particular, in [42] an ASP-based language for ontology specification and reasoning was proposed, which extends ASP in order to deal with complex real-world entities, like classes, objects, compound objects, axioms, and taxonomies. In [53] an extension of ASP, called *HEX-Programs*, which supports higher-order atoms as well as external atoms was proposed. External

atoms allows 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) [89] ontologies. *Template predicates* were introduced in [54]; they are special intensional predicates defined by means of generic reusable subprograms, which were 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 [55]. 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 [56].

In [57] standard ASP was 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: $r{:}a_1 \vee \ldots \vee a_n :\!\!-\ ^{+}body$ $(n \geq 1)$. The intuitive meaning of CR-rule r is: if $body$ is true then one of $a_1, \ldots, a_n$ 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 [58] Normal Form Nested (NFN) programs, a non-propositional language similar to Nested Logic Programming (NLP) [90] was proposed. NFN programs often allows 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 was defined. Moreover, it was shown that for NFN programs which are also NLPs, the new semantics coincides with the one of [90]; while keeping the standard meaning of answer sets on ASP programs with variables. Finally, an algorithm which translates NFN programs into ASP programs was provided.

In [91] the concept of ordered disjunctions was extended to cardinality constraints. This paved the way to the definition of a policy description language that allows to express preferences among sets of objects and to handle advanced policy description specifications. The work followed some proposals aiming at introducing preferences in policy description languages [92–94].

5 ASP with Infinite Domains

The first ASP languages were based on extensions of Datalog, that is, function-free logic programs.[5] From a syntactic viewpoint, the addition of functions is obtained by generalizing the notion of term: a *term* is either a *simple term* or a *functional term*. A *simple term* (see Section 2) is either a constant or a variable. If $t_1 \ldots t_n$ are terms and f is a function symbol (*functor*) of arity n, then: $f(t_1, \ldots, t_n)$ is a *functional term*. It

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

is easy to see that such an extension make U_P, B_P and $grnd(P)$ possibly infinite, and enhances the expressiveness of ASP. Indeed, 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 on the usage of functions, it is possible to improve the tradeoff between complexity and expressiveness.

In particular, the introduction of function symbols in ASP languages leads to several benefits [59]: (i) Data encapsulation support, as function symbols are the main logic programming construct for data abstraction [95]; (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 in modern applications and functions constitute the most natural way of encoding them; (iv) Simulation and extension of description logics [96]; 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 were introduced in [60], and soon after were followed by *ω-restricted programs* [97]. 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 [59].

Finitary programs are characterized by two restrictions. To simplify the presentation here we deal only with normal (i.e. disjunction-free) logic programs—see [61, 62] 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 $r \in grnd(P)$ 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 [61]:[6]

- 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 $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_i \subseteq \cdots \subseteq grnd(P)$.

[6] Another contribution of the Italian community; best paper award at ICLP 2007.

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 [59]. Moreover, they enjoy very good computational properties [59, 63]. 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 [64]. 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 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 [65], 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 [64].

This line of research is having an impact on the activity of other groups outside Italy. In [98], an extension of finite domain programs is proposed. In [96, 99, 100], 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* [66] 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

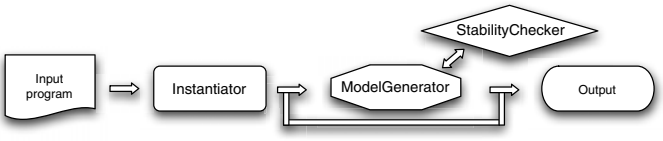

Fig. 1. General architecture of an ASP system

for generating new hypotheses and carrying out reasoning by cases. The skeptical resolution calculus is complete for all finitely recursive programs [61]. Recently, a *credulous resolution* calculus [67] was 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 [68]. We expect it to be soon extended to finitary programs by means of suitable extensions of the magic sets transformation adopted in [65].

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 [101];
- 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 typical computation carried out for computing the answer sets of an ASP program. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system.[7]

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

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

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 they exist).

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 [102] 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 cheking 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 date back to 1999, when some optimization techniques, based on a rewriting of the input program, were proposed aiming at reducing the size of the instantiation generated by the grounder [69]. 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 [70] a new join-ordering technique was proposed, that sensibly improves the instantiation procedures of ASP systems. Some year later, in [71] a new backjumping technique for the instantiation of a rule was proposed which allows for reducing both the size of the generated grounding and the time needed for producing it. All the above mentioned techniques were 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 [71] has been successfully exploited also by other two grounders, namely *GrinGo* [103], and GIDL [104].

In the last 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 [72, 73]. 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 [72] while the third one was presented in [73]. 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 [72, 73, 105].

A distributed instantiator working on a Beowulf [106] cluster was presented in [107]; further works appear in [108].

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 were exploited in [74, 109], 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 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 [75] and look-back [76, 77] techniques and heuristics specifically conceived for enhancing the model generation process were proposed and implemented in the state-of-the-art ASP system DLV [16]. In a lookahead heuristic [75] 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 [75] "layers" several criteria based on peculiar properties of ASP, and basically drives the search towards "supported" interpretations (since answer sets are supported interpretations (cfr. [34, 110, 111]). 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 [112], a family of new look-back heuristics tailored for disjunctive ASP programs were proposed in [77]. 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 [76] a *reason calculus* that allows for determining the relevance for an inconsistency was 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 [21, 113, 114], 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 [115, 116].

Techniques for parallel evaluation of ground ASP programs were studied in [117, 118] and, on clusters, in [107, 108]. Furthermore, going beyond the classical methods

of computing the answer sets of a logic program, in [119, 120] a method is presented that does not require a preliminary grounding phase.

Model Checking. is a crucial step of the computation of the answer sets. 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-NPcomplete [11]. In [78] a new transformation $\mathcal{T}$, which reduces stable model checking to UNSAT, i.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 [16] and some experiments confirmed its efficacy [78].

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 [16], 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 [16] 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}, was recently proposed [121, 122]. 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 [44], 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 [13, 123].

7.2 ASP-Based Products

In this section three industrial products strongly based on ASP, and, in particular, on DLV are presented, namely: OntoDLV [41, 42], OLEX [30, 31], HiLeX [32, 33].

• OntoDLV [41, 42] 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) [124], and it is endowed with a robust persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates DLV [16].

• OLEX [30, 31] (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.

• HiLeX [32, 33] is an advanced system for ontology-based information extraction from semi-structured and unstructured documents. HiLeX implements a semantic approach to the information extraction problem able to deal with different document formats (html, pdf, doc, ...). HiLeX 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 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 [40].

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 [125]. 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 [26] 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 HiLeX. 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 devises 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 [28]. The powerful mechanisms for database interoperability, together with magic sets [43, 44] 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 succesfully 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 [29], 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 [45].

In [126] 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 [127] 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.

The cooperation between the University of Milan-Bicocca and the University of Potsdam led to the implementation of intelligent monitoring systems based on gringo/clasp [22], where the ASP reasoning module is crucial (see, for instance, [128, 129]).

Italian researchers have exploited ASP capabilities also for diagnosis [130] and e-ealth [131] applications.

8 Further Contributions

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

In [132, 133], an integrated information retrieval agent based on an ASP inference engine, named GSA_2, was 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 was introduced in [35]. In the same work, it was 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 [134]). In [36], 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 [37, 38], kernel programs were 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 were studied in [39]. In [135], some features that graph representations of ASP programs should exhibit, especially isomorphism between a program and the corresponding graph, were 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 [136] and/or adding new rules [137]. The work [138] 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 [139] 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 was exploited as a core inference engine for a system for qualitative management of probabilistic uncertainty [140–142]. 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 was integrated with arithmetic and finite domain constraint solvers in [143]. 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 (see also [144]).

The mutual interdependence of ASP-based agents has been investigated [145–148] at Università Mediterranea of Reggio Calabria. In [145], agreements possibly reached by a collection of agents are represented. In [146, 147], a community of agents where individual conclusions rely on others ones is modeled by nested social predicates. This language is refined in [148] 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 [149].

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
2. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: ICLP/SLP 1988, pp. 1070–1080. MIT Press, Cambridge (1988)
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
4. Lifschitz, V.: Answer Set Planning. In: ICLP 1999, pp. 23–37 (1999)
5. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398 (1999)
6. Minker, J. (ed.): Foundations of Deductive Databases and Logic Programming, Washington, DC (1988)
7. Minker, J., Rajasekar, A.: A Fixpoint Semantics for Disjunctive Logic Programs. JLP 9(1), 45–74 (1990)
8. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge (1992)
9. Fernández, J.A., Minker, J.: Semantics of Disjunctive Deductive Databases. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 21–50. Springer, Heidelberg (1992)
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)

11. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Logic-Based Artificial Intelligence, pp. 79–103. Kluwer, Dordrecht (2000)
13. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The First Answer Set Programming System Competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 3–17. Springer, Heidelberg (2007)
14. Zhao, Y.: The Second Answer Set Programming Competition homepage (2009x), `http://www.cs.kuleuven.be/~dtai/ASP-competition`
15. Dovier, A., Formisano, A., Pontelli, E.: An Empirical Study Of Constraint Logic Programming And Answer Set Programming Solutions Of Combinatorial Problems. J. Exp. Theor. Artif. Intell. 21(2), 79–121 (2009)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
17. Simons, P.: Smodels Homepage (since (1996), `http://www.tcs.hut.fi/Software/smodels/`
18. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI 138, 181–234 (2002)
19. Zhao, Y.: ASSAT homepage (since 2002), `http://assat.cs.ust.hk/`
20. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI 2002, Edmonton, Alberta, Canada. AAAI Press / MIT Press (2002)
21. Babovich, Y., Maratea, M.: Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs (2003), `http://www.cs.utexas.edu/users/tag/cmodels.html`
22. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: IJCAI 2007, pp. 386–392 (2007)
23. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL 7(1), 1–37 (2006)
24. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 447–451. Springer, Heidelberg (2005)
25. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), Sydney, Australia, pp. 422–432. AAAI Press, Menlo Park (2008)
26. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An ASP-Based System for e-Tourism. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 368–381. Springer, Heidelberg (2009)
27. Leone, N., Ricca, F., Terracina, G.: An ASP-Based Data Integration System. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 528–534. Springer, Heidelberg (2009)
28. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, pp. 915–917. ACM Press, New York (2005)
29. Franconi, E., Palma, A.L., Leone, N., Perri, S.: Census Data Repair: A Challenging Application of Disjunctive Logic Programming. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 561–578. Springer, Heidelberg (2001)
30. Cumbo, C., Iiritano, S., Rullo, P.: OLEX – A Reasoning-Based Text Classifier. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 722–725. Springer, Heidelberg (2004)

31. Rullo, P., Cumbo, C., Policicchio, V.L.: Learning Rules With Negation For Text Categorization. In: ACM Symposium on Applied Computing (SAC 2007), Seoul, Korea, 11-15, pp. 409–416. ACM, New York (2007)
32. Ruffolo, M., Manna, M.: HiLeX: A System for Semantic Information Extraction from Web Documents. In: ICEIS. Lecture Notes in Business Information Processing, vol. (3), pp. 194–209. Springer, Heidelberg (2008)
33. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP 2005 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK, pp. 248–262 (2005)
34. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation 135(2), 69–112 (1997)
35. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. Theoretical Computer Science 149 (1995); preliminary version in Proc. of LPNMR93
36. Brignoli, G., Costantini, S., D'Antona, O., Provetti, 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)
37. Bertoni, A., Grossi, G., Provetti, A., Kreinovich, V., Tari, L.: The Prospect for Answer Set Computation by a Genetic Model. In: AAAI Spring Symposium ASP 2001, pp. 1–5. AAAI Press, Menlo Park (2001)
38. Grossi, G., Marchi, M., Pontelli, E., Provetti, A.: Improving the AdjSolver Algorithm for ASP Kernel Programs. In: ASP 2007, 4th International Workshop on Answer Set Programming at ICLP 2007 (2007)
39. Costantini, S., Provetti, A.: Normal Forms for Answer Sets Programming. J. on TPLP 5(6) (2005)
40. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 591–597. Springer, Heidelberg (2009)
41. 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)
42. Ricca, F., Leone, N.: Disjunctive Logic Programming With Types And Objects: The Dlv$^+$ System. Journal of Applied Logics 5(3), 545–573 (2007)
43. Cumbo, C., Faber, W., Greco, G., Leone, N.: Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 371–385. Springer, Heidelberg (2004)
44. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. JCSS 73(4), 584–609 (2007)
45. Friedrich, G., Ivanchenko, V.: Diagnosis From First Principles For Workflow Executions. Tech. Rep., http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf
46. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE 12(5), 845–860 (2000)
47. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: IJCAI 2005, pp. 406–411 (2005)
48. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003, Acapulco, Mexico, pp. 847–852 (2003)
49. Faber, W., Leone, N.: On the Complexity of Answer Set Programming with Aggregates. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 97–109. Springer, Heidelberg (2007)

50. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
51. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. ACM TOCL 5(2), 206–263 (2004)
52. Perri, S., Scarcello, F., Leone, N.: Abductive Logic Programs with Penalization: Semantics, Complexity and Implementation. TPLP 5(1–2), 123–159 (2005)
53. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: IJCAI 2005, Edinburgh, UK, pp. 90–96 (2005)
54. Calimeri, F., Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: A System with Template Answer Set Programs. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 693–697. Springer, Heidelberg (2004)
55. Costantini, S., Formisano, A.: Answer Set Programming with Resources. Journal of Logic and Computation (to appear, 2009), www.dipmat.unipg.it/~formis/papers/report200816.ps.gz Draft available as Report-16/2008 of Dip. di Matematica e Informatica, Univ. di Perugia
56. Costantini, S., Formisano, A.: Modeling Preferences And Conditional Preferences On Resource Consumption And Production In Asp. Journal of of Algorithms in Cognition, Informatics and Logic 64(1), 3–15 (2009)
57. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In: International Symposium on Logical Formalization of Commonsense Reasoning. AAAI 2003 Spring Symposium Series (2003)
58. Bria, A., Faber, W., Leone, N.: Normal Form Nested Programs. FI (2009) (accepted for publication)
59. Bonatti, P.A.: Reasoning with Infinite Stable Models. Artif. Intell. 156(1), 75–111 (2004)
60. Bonatti, P.: Reasoning with Infinite Stable Models. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, pp. 603–610 (2001)
61. Baselice, S., Bonatti, P.A., Criscuolo, G.: On Finitely Recursive Programs. TPLP 9(2), 213–238 (2009)
62. Bonatti, P.A.: Reasoning with infinite stable models II: Disjunctive programs. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 333–346. Springer, Heidelberg (2002)
63. Bonatti, P.A.: Erratum to: Reasoning with infinite stable models [artificial intelligence 156(1) (2004) 75–111]. Artif. Intell. 172(15), 1833–1835 (2008)
64. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: [150], pp.407–424
65. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In: [151], 71–86
66. Bonatti, P.A.: Resolution for Skeptical Stable Model Semantics. J. Autom. Reasoning 27(4), 391–421 (2001)
67. Bonatti, P.A., Pontelli, E., Son, T.C.: Credulous Resolution for Answer Set Programming. In: AAAI, pp. 418–423. AAAI Press, Menlo Park (2008)
68. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An ASP System with Functions, Lists, and Sets. In: [151], 483–489
69. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In: DDLP 1999, Prolog Association of Japan, pp. 135–139 (1999)
70. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 280–294. Springer, Heidelberg (2001)

71. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV Instantiator by Backjumping Techniques. AMAI 51(2-4), 195–228 (2007)
72. 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)
73. 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)
74. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Disjunctive Logic Programming Systems. FI 71(2-3), 183–214 (2006)
75. Faber, W., Leone, N., Pfeifer, G., Ricca, F.: On look-ahead heuristics in disjunctive logic programming. AMAI 51(2-4), 229–266 (2007)
76. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications 19(2), 155–172 (2006)
77. 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)
78. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. AI 15(1-2), 177–212 (2003)
79. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
80. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. J. ACM 38(3), 620–650 (1991)
81. Reiter, R.: On Closed World Data Bases. In: Logic and Data Bases, pp. 55–76. Plenum Press, New York (1978)
82. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI 12, 53–87 (1994)
83. 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 1994), pp. 39–50 (1994)
84. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, p. 212. Springer, Heidelberg (2001)
85. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with Aggregate Operators. J. ACM 48(4), 880–907 (2001)
86. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. TPLP 7(3), 301–353 (2007)
87. Elkabani, I., Pontelli, E., Son, T.C.: Smodels^{A} - A System for Computing Answer Sets of Logic Programs with Aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 427–431. Springer, Heidelberg (2005)
88. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. TPLP 7, 355–375 (2007)
89. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
90. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. AMAI 25(3-4), 369–389 (1999)
91. Mileo, A., Schaub, T.: Qualitative Constraint Enforcement in Advanced Policy Specification. In: Mellouli, K. (ed.) ECSQARU 2007. LNCS (LNAI), vol. 4724, pp. 695–706. Springer, Heidelberg (2007)

92. Bertino, E., Mileo, A., Provetti, A.: PDL with Preferences. In: POLICY, pp. 213–222. IEEE Computer Society, Los Alamitos (2005)
93. Marchi, M., Mileo, A., Provetti, A.: Specification and Execution of Declarative Policies for Grid Service Selection. In (LJ) Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 102–115. Springer, Heidelberg (2004)
94. Bertino, E., Mileo, A., Provetti, A.: PDL with Maximum Consistency Monitors. In: Zhong, N., Raś, Z.W., Tsumoto, S., Suzuki, E. (eds.) ISMIS 2003. LNCS (LNAI), vol. 2871, pp. 65–74. Springer, Heidelberg (2003)
95. Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. MIT Press, Cambridge (1994)
96. Šimkus, M., Eiter, T.: FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 514–530. Springer, Heidelberg (2007)
97. Syrjänen, T.: Omega-Restricted Logic Programs. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 267–279. Springer, Heidelberg (2001)
98. Lierler, Y., Lifschitz, V.: One More Decidable Class of Finitely Ground Programs. In: [152], pp. 489–493
99. Eiter, T., Ortiz, M., Šimkus, M.: Reasoning Using Knots. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 377–390. Springer, Heidelberg (2008)
100. Simkus, M.: Fusion of Logic Programming and Description Logics. In: [152], pp. 551–552
101. Baselice, S., Bonatti, P.A.: Composing Normal Programs with Function Symbols. In: [150], pp. 425–439
102. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communications of the ACM 5, 394–397 (1962)
103. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A New Grounder for Answer Set Programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
104. Wittocx, J., Mariën, M., Denecker, M.: GidL: A Grounder for FO+. In: Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning, pp. 189–198 (2008)
105. Perri, S., Ricca, F., Sirianni, M.: A Parallel ASP Instantiator Based on DLV. In: DAMP, pp. 73–82. ACM, New York (2010)
106. Beowulf.org: The Beowulf Cluster Site, `http://www.beowulf.org`.
107. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in Parallel Execution of Non-Monotonic Reasoning Systems. Parallel Computing 31(6), 608–647 (2005)
108. Grossi, G., Marchi, M., Pontelli, E., Provetti, A.: Experimental Analysis of Graph-based Answer Set Computation over Parallel and Distributed Architectures. J. of Logic and Computation 19(4), 697–715 (2009)
109. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS (LNAI), vol. 1730, pp. 177–191. Springer, Heidelberg (1999)
110. Marek, V.W., Subrahmanian, V.: The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In: ICLP 1989, pp. 600–617. MIT Press, Cambridge (1989)
111. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. JLP (19/20), 73–148 (1994)
112. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001, pp. 530–535 (2001)
113. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer Set Programming Based on Propositional Satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
114. Giunchiglia, E., Lierler, Y., Maratea, M.: A SAT-based Polynomial Space Algorithm for Answer Set Programming. In: Proceedings of the 10th International Workshop on Non-Monotonic Reasoning (NMR 2004), pp. 189–196 (2004)

115. Giunchiglia, E., Maratea, M.: On the relation between answer set and SAT procedures (or, between CMODELS and SMODELS). In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 37–51. Springer, Heidelberg (2005)
116. Giunchiglia, E., Leone, N., Maratea, M.: On the Relation among Answer Set Solvers. AMAI 53(1-4), 169–204 (2008)
117. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Proceedings of the 1st Intl. ASP 2001 Workshop on Answer Set
Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Stanford, pp. 174–180 (2001)
118. Le, H.V., Pontelli, E.: Dynamic Scheduling in Parallel Answer Set Programming Solvers. In: SpringSim (2), SCS/ACM, pp. 367–374 (2007)
119. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Answer Set Programming with Constraints Using Lazy Grounding. In: [152], pp. 115–129
120. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: GASP: Answer Set Programming with Lazy Grounding. FI 96(3), 297–322 (2009)
121. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with Recursive Queries in Database and Logic Programming Systems. TPLP 8, 129–165 (2008)
122. Terracina, G., De Francesco, E., Panetta, C., Leone, N.: Enhancing a DLP System for Advanced Database Applications. In: Calvanese, D., Lausen, G. (eds.) RR 2008. LNCS, vol. 5341, pp. 119–134. Springer, Heidelberg (2008)
123. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
124. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA 2007), pp. 56–70 (2007)
125. 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: Peña, R. (ed.) PADL 2010. LNCS, vol. 5937, pp. 40–42. Springer, Heidelberg (2010)
126. Garro, A., Palopoli, L., Ricca, F.: Exploiting Agents in E-Learning and Skills Management Context. AI Communications 19(2), 137–154 (2006)
127. 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, pp. 293–302 (2005)
128. Mileo, A., Merico, D., Bisiani, R.: Non-monotonic Reasoning Supporting Wireless Sensor Networks for Intelligent Monitoring: The SINDI System. In: [151], pp. 585–590
129. Mileo, A., Merico, D., Bisiani, R.: A Logic Programming Approach to Home Monitoring for Risk Prevention in Assisted Living. In: [150], pp. 145–159
130. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. TPLP 3, 425–461 (2003)
131. Bisiani, R., Merico, D., Mileo, A., Pinardi, S.: A Logical Approach to Home Healthcare with Intelligent Sensor-Network Support. The Computer Journal (2009); bxn074
132. Ianni, G., Calimeri, F., Lio, V., Galizia, S.: Reasoning about the Semantic Web using Answer Set Programming. In: APPIA-GULP-PRODE, pp. 324–336 (2003)
133. 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), pp. 492–493 (2004)
134. Subrahmanian, V., Nau, D., Vago, C.: WFS + Branch and Bound = Stable Models. IEEE TKDE 7(3), 362–377 (1995)
135. 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, CA (2001)

136. Costantini, S., Lanzarone, G.A., Magliocco, G.: Asserting Lemmas in the Stable Model Semantics. In: Logic Programming – Proc. of the 1996 Joint International Conference, USA (1996)
137. Costantini, S., Intrigila, B., Provetti, A.: Coherence of Updates in Answer Set Programming. In: IJCAI 2003 Workshop on Nonmonotonic Reasoning, Action and Change, NRAC 2003, pp. 66–72 (2003)
138. Costantini, S.: On the Existence of Stable Models of Non-Stratified Logic Programs. J. on TPLP 6(1-2) (2006)
139. Costantini, S., Formisano, A., Omodeo, E.G.: Mappings Between Domain Models in Answer Set Programming. In: Answer Set Programming, Advances in Theory and Implementation, Proc. of the 2nd Intl. ASP 2003. CEUR Workshop Proc., vol. 78 (2003)
140. Capotorti, A., Formisano, A.: Comparative Uncertainty: Theory and Automation. Mathematical Structures in Computer Science 18(1) (2008)
141. Capotorti, A., Formisano, A., Murador, G.: Qualitative Uncertainty Orderings Revised. Electronic Notes in Theoretical Computer Science 169, 43–59 (2007)
142. Capotorti, A., Formisano, A.: Management of Uncertainty Orderings Through ASP. In: Modern Information Processing: From Theory to Applications. Elsevier, Amsterdam (2004) ISBN: 0-444-52075-9
143. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
144. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog Decision Support System for the Space Shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
145. Buccafurri, F., Gottlob, G.: Multiagent compromises, joint fixpoints, and stable models. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2407, pp. 561–585. Springer, Heidelberg (2002)
146. Buccafurri, F., Caminiti, G.: A Social Semantics for Multi-agent Systems. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 317–329. Springer, Heidelberg (2005)
147. Buccafurri, F., Caminiti, G.: Logic Programming with Social Features. TPLP 8(5–6), 643–690 (2008)
148. Buccafurri, F., Caminiti, G., Laurendi, R.: A Logic Language with Stable Model Semantics for Social Reasoning. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 718–723. Springer, Heidelberg (2008)
149. Buccafurri, F., Caminiti, G., Rosaci, D.: Logic Programs with Multiple Chances. In: ECAI, pp. 347–351 (2006)
150. Garcia de la Banda, M., Pontelli, E. (eds.): ICLP 2008. LNCS, vol. 5366. Springer, Heidelberg (2008)
151. Erdem, E., Lin, F., Schaub, T. (eds.): LPNMR 2009. LNCS, vol. 5753, pp. 14–18. Springer, Heidelberg (2009)
152. Hill, P.M., Warren, D.S. (eds.): Logic Programming. LNCS, vol. 5649, pp. 14–17. Springer, Heidelberg (2009)