A

ANSWER SET PROGRAMMING

INTRODUCTION

Computer applications pervade our life, and these days many problems of everyday life are dealt with in an automated way. However, not all problems are easy to solve by a computer, some have an increased intrinsic complexity. Finding efficient and correct methods for solving them is not an easy task. Traditional software engineering is focused on an imperative, algorithmic approach, in which the computer is basically being told what steps should be followed in order to solve the given problem. Finding good algorithms for hard problems requires skill and knowledge and is often not obvious. This can be dealt with by involving an expert, but there is a serious drawback: When the specification of the problem changes slightly, perhaps only because additional information on the nature of the problem becomes available, major reengineering is often necessary. The main problem is that the knowledge about the problem and its solutions has been represented implicitly by representing a specific way of solving the problem rather than the problem itself. The case of updating representations is sometimes called elaboration tolerance.

An alternative that suits elaboration tolerance better is called declarative programming. In this approach, the problem and its solutions are specified explicitly. That is, it is expressed what features the problem and its solution must have, rather than specifying how a solution is to be obtained. Methods like this actually come natural in science but also in everyday life. Before we try to work out how to solve a problem, we usually first try to understand it and figure out how a solution would like, before trying to find a method, to obtain a solution. One of the first to put this approach into perspective in computer science was John McCarthy in the 1950s (1). He also postulated that the most natural language for specifying problems and solutions would be logic and, in particular, predicate logic.

In fact, logic is an excellent candidate for declarative programming: It provides a simple and abstract formalism, and in addition, it has the potential for automation. Similar to an abstract or electronic machine that can execute an imperative model (an algorithm) in order to obtain a solution of the modeled problem, computational logic has produced tools that allow for automatically obtaining solutions, given a declarative specification in logic. Indeed, many people nowadays use this way of solving problems: Queries to relational databases together with the database schemata are indeed declarative specifications of the solutions that the query results provide. And, indeed, the probably most widely used database query language, SQL, is basically predicate logic written in a particular way (2).

However, one wants to go beyond databases as they are used today. It has been shown that relational databases and query languages like SQL can only represent fairly simple problems. For instance, problems like finding the cheapest tour of several cities, or filling a container with items of different size, such that the value transported in the container is maximized, are typical problems that probably cannot be solved using SQL. It might seem unusual to use the word "probably" here, but underlying this conjecture is one of the most famous open problems in computer science—the question of whether P equals NP. These are complexity classes; basically, every problem has some intrinsic complexity, which is based on how many resources are required to solve it on a standard machine model, in terms of the size of the problem input. P is defined as the class of problems, which require at most an amount of time, which can be expressed as a polynomial over the input size (which is variable). NP is just a slight alteration, in which instead of a deterministic machine model, a nondeterministic machine model is assumed. A nondeterministic machine is a somewhat unusual concept: Instead of executing commands one-by-one, always going from one machine state to another, a nondeterministic machine may be in two or more states (at the same time) after having executed a command. In a sense, this means that the machine has the possibility to store and work with an unbounded number of machine states at any time. Intuitively, one would expect that a deterministic and a nondeterministic machine are quite different from each other, and that the nondeterministic machine can solve more problems under the same time constraints. However, up to now, nobody has been able to prove convincingly neither that P and NP are different, nor that they are equal. However, intuitively one would expect that they are different, and people have shown that many more unintuitive results would follow if P and NP coincided.

Logic programming is an attempt to use declarative programming with logic that goes beyond problems in P and, thus, beyond traditional databases. The main construct in logic programming is a rule, a expression that looks like $Head \leftarrow Body$, where *Body* is a logic conjunction possibly involving negation, and *Head* is either an atomic formula or a logic disjunction. This can be seen as a logic formula ($\leftarrow$ denoting implication), with the special meaning that *Head* is defined by that rule. In the beginning of the field (as described in the following section), logic programming actually attempted to become a full-scale "programming language." Its most famous language, Prolog (3), aimed at this, but had to renounce to full declarativity in order to achieve that goal. For instance, in Prolog rules, the order inside *Body* matters, as does the order among rules (most notably for termination). Moreover, Prolog also had several nonlogical constructs.

Answer set programming (ASP) is a branch of logic programming, which does not aspire to create a full general-purpose language. In this respect, it is influenced by database languages, as also these are not general-purpose languages, but suffice for a particular class of problems. ASP does, however, attempt to enlarge the class of problems that can be expressed by the language. Although, as

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

mentioned, SQL probably cannot express hard problems in NP, ASP definitely can. Actually, ASP can express all problems in the complexity class $\sum_2^p$ and its complement $\prod_2^p$, which are similar to NP, but probably somewhat larger (but at least equally large).

In ASP, the rule construct *Head* ← *Body* (where *Head* can be a disjunction) is read like a formula in nonmonotonic logics rather than classical logic. Nonmonotonic logics are an effort to formulate a logic of common sense that is adapting the semantics of logic such that it corresponds better to our everyday reasoning, which is characterized by the presence of incomplete knowledge, hypothetical reasoning, and default assumptions. It can be argued that nonmonotonic logics are much better suited in such a setting than classical logic.

Summarizing, ASP is a formalism that has emerged from logic programming. Its main representation feature are rules, which are interpreted according to common sense principles. It allows for declarative specifications of a rich class of programs, generalizing the declarative approach of databases. In ASP, one writes a program (a collection of rules), which represent a problem to be solved. This program, together with some input, which is also expressed by a collection of rules, possesses a collection of solutions (possibly also no solution), which correspond to the solutions of the modeled problem. Since these solutions are usually sets, the term "answer set" has been coined.

Concerning terminology, ASP is sometimes used in a somewhat broader sense, referring to any declarative formalism representing solutions as sets. However, the more frequent understanding is the one adopted in this article, which dates back to Ref. 4. Moreover, since ASP is the most prominent branch of logic programming in which rule heads may be disjunctive, sometimes the term "disjunctive logic programming" can be found referring explicitly to ASP. Yet other terms for ASP are A-Prolog and stable logic programming. For complementary introductory material on ASP, we refer to Refs. 5 and 6.

LOGIC PROGRAMMING

The roots of answer set programming lie predominantly in logic programming, nonmonotonic reasoning, and databases. In this section, we give an overview on the history of logic programming from the perspective of answer set programming. It, therefore, does not cover several important subfields of logic programming, such as constraint logic programming (7) or abductive logic programming (8).

As mentioned, probably the first to suggest logic, and in particular predicate logic, as a programming language was John McCarthy in the 1950s (1). McCarthy's motivating example was set in artificial intelligence, and involved planning as its main task, an agenda on which was continuously elaborated; see, for instance, Ref. 9.

Developments in computational logic, most notably the specification of the resolution principle and unification as a computational method by J. Alan Robinson in 1965 (10), acted as a catalyst for the rise of logic programming. This development eventually really set off when a working system, Prolog, developed by a group around Alain Colmerauer in Marseilles, France, became available (3). A few other, somewhat more restricted systems had been available before, but Prolog was to make the breakthrough for logic programming.

One of the prime advocates of what would become known as the logic programming paradigm has been Robert Kowalski, who provided the philosophical basis and concretizations of the logic programming paradigm, for instance, in Refs. 11 and 12. Kowalski also collaborated with Colmerauer on Prolog, and in the realm of his group in Edinburgh, Scotland, alternative implementations of Prolog were created. There has also been a standardization effort for the language, which would become known as Edinburgh Prolog and served as the de facto specification of Prolog for many years until the definition of ISO Prolog in 1995 (13).

However, logic programming, and Prolog in particular, was inspired by, but not the same as classical first-order logic. Initially the differences were not entirely clear. The first effort to provide a formal definition for the semantics of logic programming was also undertaken by Kowalski, who together with Maarten van Emden gave a semantics based on fixpoints of operators for a restricted class of logic programs (Horn programs, also called positive programs) in Ref. 14. This fixpoint semantics essentially coincided with minimal Herbrand models and with resolution-based query answering on Horn programs. The major feature missing in Horn programs is negation—however, Prolog did have a negation operator.

Indeed, the quest for finding a suitable semantics in the spirit of minimal models for programs containing negation turned out to be far from straightforward. A first attempt was made by Keith Clark in Ref. 15 by defining a transformation of the programs to formulas in classical logic, which are then interpreted using the classical model semantics. However, the approach gave arguably unintuitive results for programs with positive recursion. In particular, the obtained semantics does not coincide with the minimal model semantics on positive programs. At about the same time, Raymond Reiter formulated the Closed World Assumption in Ref. 16, which can be seen as the philosophical basis of the treatment of negation. Another milestone in the research on the intended semantics for programs with negation has been the definition of what later became known uniformly as perfect model semantics for programs that can be stratified on negation, in Refs. 17 and 18. The basic idea of stratification is that programs can be partitioned in subprograms (strata) such that the rules of each stratum contain negative predicates only if they are defined in other strata. In this way, it is possible to evaluate the program by separately evaluating its partitions in such a way that a given "stratum" is processed whenever the ones from which it (negatively) depends have already been processed.

Although an important step forward, it is obvious that not all logic programs are stratified. In particular, programs that are recursive through negation are never stratified, and the problem of assigning a semantics to nonstratified programs still remained open. There were basically two approaches for finding suitable definitions: The first approach was giving up the classical setting of

models that assign two truth values, and introduce a third value, intuitively representing unknown. This approach required a somewhat different definition, because in the two-valued approach, one would give a definition only for positive values, implicitly stating that all other constructs are considered to be negative. For instance, for minimal models, one minimizes the true elements, implicitly stating that all elements not contained in the minimal model will be false. With three truth values, this strategy is no longer applicable, as elements that are not true can be either false or undefined. For resolving this, Allen Van Gelder, Kenneth Ross, and John Schlipf introduced the notion of unfounded sets in Ref. 19, in order to define which elements of the program should be definitely false. Combining existing techniques for defining the minimal model with unfounded sets, they defined the notion of a well-founded model. In this way, any program would still be guaranteed to have a single model, just like there is a unique minimal model for positive programs and a unique perfect model for stratified programs.

The second approach consisted of viewing logic programs as formulas in nonmonotonic logics (see, for instance, Ref. 20 for an overview) rather than formulas of classical logic (with an additional minimality criterion) and as a corollary, abandoning the unique model property. Among the first to concretize this were Michael Gelfond in Ref. 21, who proposed to view logic programs as formulas of autoepistemic logic, and Nicole Bidoit and Christine Froidevaux in Ref. 22, who proposed to view logic programs as formulas of default logic. Both of these developments have been picked up by Michael Gelfond and Vladimir Lifschitz, who in Ref. 23 defined the notion of stable models, which is inspired by nonmonotonic logics, however does not refer explicitly to these, but rather relies on a reduct that effectively emulates nonmonotonic inference. It was this surprisingly simple formulation, which did not require previous knowledge on non-classical logics that has become well known. Different to well-founded models, there may exist no, one, or many stable models for one program. However, well-founded and stable models are closely related; for instance, the well-founded model of a program is contained in each stable model (cf. Ref. 24). Moreover, both approaches coincide with perfect models on stratified programs.

Yet another, somewhat orthogonal line of research concerned the use of disjunction in rule heads. This construct is appealing, because it allows for direct nondeterministic definitions. Prolog and many other logic programming languages traditionally do not provide such a feature, being restricted to so-called definite rules. Jack Minker has been a pioneer and advocate of having disjunctions in programs. In Ref. 25, he formulated the Generalized Closed World Assumption, which gave a simple and intuitive semantics for disjunctive logic programs. This concept has been elaborated on over the years, most notably by the Extended GCWA defined in Ref. 26. Eventually, also the stable model semantics has been extended to disjunctive programs in Ref. 27 by just minimally altering the definition of Ref. 23. On the other hand, defining an extension of well-founded models for disjunctive programs remains a controversial matter to this date with various rivalling definitions, (cf. Ref. 28).

The final step toward answer set programming in the traditional sense has been the addition of a second kind of negation, which has a more classical meaning than negation as failure. Combining this feature with disjunctive stable models of Ref. 27 led to the definition of answer sets in Ref. 4.

FORMAL DEFINITION OF ASP

In what follows, we provide a formal definition of the syntax and semantics of answer set programming in the spirit of Ref. 4, that is, disjunctive logic programming involving two kinds of negation (referred to as strong negation and negation as failure), under the answer sets semantics.

Syntax

Following a convention dating back to Prolog, strings starting with uppercase letters denote logical variables, whereas strings starting with lowercase letters denote constants. A *term* is either a variable or a constant. Note that, as common in ASP, function symbols are not considered.

An *atom* is an expression $\mathcal{P}(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *classical literal* l is either an atom p (in this case, it is *positive)*, or a negated atom $\neg p$ (in this case, it is *negative).* A *negation as failure (NAF) literal* ℓ is of the form l or not l, where l is a classical literal; in the former case ℓ is *positive*, and in the latter case *negative.* Unless stated otherwise, by *literal* we mean a classical literal.

Given a classical literal l, its *complementary* literal $\neg l$ is defined as $\neg p$ if $l = p$ and p if $l = \neg p$. 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

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{not } b_m. \qquad (1)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is called the *head* of r, whereas the conjunction $b_1, \ldots, b_k$, not $b_{k+1}, \ldots$, 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 "$\leftarrow$" sign is usually omitted.

The following notation will be useful for additional discussion. 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* $\mathcal{P}$ is a finite set of rules. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive* or *Horn*,[1] and a $\vee$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *normal logic program*.

[1] In positive programs, negation as failure (not) does not occur, whereas strong negation ($\neg$) may be present.

In ASP, rules in programs are usually required to be safe. The motivation of safety comes from the field of databases, where safety has been introduced as a means to guarantee that queries (programs in the case of ASP) do not depend on the universe (the set of constants) considered. As an example, a fact $p(X)$. gives rise to the truth of $p(a)$ when the universe $\{a\}$ is considered, whereas it gives rise to the truth of $p(a)$ and $p(b)$ when the universe $\{a, b\}$ is considered. Safe programs do not suffer from this problem when at least the constants occurring in the program are considered. For a detailed discussion, we refer to Ref. 2.

A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. 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.

Example 3.1. Consider the following program:

r_1 : a(X) $\vee$ b(X) $\leftarrow$ c(X,Y), d(Y), not e(X).
r_2 : $\leftarrow$ c(X,Y), k(Y), e(X), not b(X).
r_3 : m $\leftarrow$ n, o, a(1).
r_4 : e(1, 2).

r_1 is a disjunctive rule with $H(r_1) = \{a(X), b(X)\}$, $B^+(r_1) = \{c(X,Y), d(Y)\}$, and $B^{\cdot}(r_1)\} = \{e(X)\}$.$r_2$ is an integrity constraint with $B^+(r_2) = \{c(X,Y), k(Y), e(X)\}$, and $B^-(r_2) = \{b(X)\}$.r_3 is a ground, positive, and nondisjunctive rule with $H(r_3) = \{m\}, B^+(r_3) = \{n, o, a(1)\}$, and $B^-(r_3) = \emptyset$.r_4, finally, is a fact (note that $\leftarrow$ is omitted). Moreover, all of the rules are safe. □

Semantics

We next describe the semantics of ASP programs, which is based on the answer set semantics originally defined in Ref. 4. However, different than Ref. 4, only consistent answer sets are considered, as it is now standard practice.

We note that in ASP the availability of some preinterpreted 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 here.

Herbrand Universe and Literal Base. For any program $\mathcal{P}$, the *Herbrand universe*, denoted by $U_\mathcal{P}$, is the set of all constants occurring in $\mathcal{P}$. If no constant occurs in $\mathcal{P}$, $U_\mathcal{P}$ consists of one arbitrary constant[2]. The *Herbrand literal base* $B_\mathcal{P}$ is the set of all ground (classical) literals constructible from predicate symbols appearing in $\mathcal{P}$ and constants in $U_\mathcal{P}$ (note that, for each atom $\mathcal{P}$, $B_\mathcal{P}$ contains also the strongly negated literal $\neg p$).

[2]Actually, since the language does not contain function symbols and since rules are required to be safe, this extra constant is not needed. However, we have kept the classic definition in order to avoid confusion.

Example 3.2. Consider the following program:

$\mathcal{P}_0$ = {
r_1: a(X) $\vee$ b(X) $\leftarrow$ c(X, Y).
r_2: c(X) $\leftarrow$ c(X, Y), not b(X).
r_4: c(1, 2).
}

then, the universe is $U_{\mathcal{P}0}$ = {1,2}, and the base is $B_{\mathcal{P}0}$ = {a(1), a(2), b(1), b(2), c(1), c(2), c(1,1), c(1,2), c(2,1), c(2,2), $\neg$a(1), $\neg$a(2), $\neg$b(1), $\neg$b(2), $\neg$c(1), $\neg$c(2), $\neg$c(1,1), $\neg$c(1,2), $\neg$c(2,1), $\neg$c(2,2)}. □

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

Example 3.3. Consider again problem $\mathcal{P}_0$ of Example 3.2. Its ground instantiation is:

$Ground(\mathcal{P}_0)$ = {
g_1: a(1) $\vee$ b(1) $\leftarrow$ c(1, 1).
g_2: a(1) $\vee$ b(1) $\leftarrow$ c(1, 2).
g_3: a(2) $\vee$ b(2) $\leftarrow$ c(2, 1).
g_4: a(2) $\vee$ b(2) $\leftarrow$ c(2, 2).
g_5: e(1) $\leftarrow$ c(1, 1), not b(1).
g_6: e(1) $\leftarrow$ c(1, 2), not b(1).
g_7: e(2) $\leftarrow$ c(2, 1), not b(2).
g_8: e(2) $\leftarrow$ c(2, 2), not b(2).
g_9: c(1, 2).
}

Note that the atom $c(1,2)$ was already ground in $\mathcal{P}_0$, whereas the rules $g_1, \ldots, g_4$ (resp. $g_5, \ldots, g_8$) are obtained by replacing the variables in r_1 (resp. r_2) with constants in $U_{\mathcal{P}_0}$. □

Answer Sets. For every program $\mathcal{P}$, its answer sets are defined using its ground instantiation $Ground(\mathcal{P})$ in two steps: First the answer sets of positive disjunctive programs are defined, and then the answer sets of general programs are defined by a reduction to positive disjunctive programs and a stability condition.

An interpretation l is a consistent[3] set of ground classical literals $I \subseteq B p$ w.r.t. a program $\mathcal{P}$. A consistent interpretation $X \subseteq B_\mathcal{P}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive datalog program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation which is closed under $\mathcal{P}$ is also called *model* of $\mathcal{P}$. An interpretation $X \subseteq B_\mathcal{P}$ is an *answer set* for a positive disjunctive program $\mathcal{P}$, if it is minimal (under set inclusion) among all (consistent) interpretations that are closed under $\mathcal{P}$.

Example 3.4. The positive program $\mathcal{P}_1 = \{a \vee \neg b \vee c.\}$ has the answer sets $\{a\}$, $\{\neg b\}$, and $\{c\}$; note that they are minimal and correspond to the multiple ways of satisfying the disjunction. Its extension $\mathcal{P}_2 = \mathcal{P}_1 \cup \{\leftarrow a.\}$ has

[3]A set $I \subseteq B_\mathcal{P}$ is *consistent* if for each positive classical literal such that $l \in I$ it holds that $\neg l \notin I$.

the answer sets $\{\neg b\}$ and $\{c\}$, since comparing $\mathcal{P}_2$ with $\mathcal{P}_1$, the additional constraint is not satisfied by interpretation $\{a\}$. Moreover, the positive program $\mathcal{P}_3 = \mathcal{P}_2 \cup \{\neg b \leftarrow c.;\ c \leftarrow \neg b.\}$ has the single answer set $\{\neg b, c\}$ (indeed, the remaining consistent closed interpretation $\{a,\ \neg b, c\}$ is not minimal). Although, it is easy to see that, $\mathcal{P}_4 = \mathcal{P}_3 \cup \{\leftarrow c\}$ has no answer set. □

The *reduct* or *Gelfond–Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_\mathcal{P}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

- Deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds
- Deleting the negative body from the remaining rules

An *answer set* of a program $\mathcal{P}$ is a set $X \subseteq B_\mathcal{P}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Example 3.5. For the negative ground program $\mathcal{P}_5 = \{a \leftarrow \text{not}\, b.\}$, $A = \{a\}$ is the only answer set, as $\mathcal{P}_5^A = \{a.\}$. For example, for $B = \{b\}$, $\mathcal{P}_5^B = \emptyset$, and so B is not an answer set. □

Example 3.6. Consider again program $\mathcal{P}_0$ of Example 3.2, whose ground instantiation $Ground(\mathcal{P}_0)$ has been reported in Example 3.3. A naÿve way to compute the answer sets of $\mathcal{P}_0$ is to consider all possible interpretations, checking whether they are answer sets of $Ground(\mathcal{P}_0)$.

For instance, consider interpretation $I_0 = \{c(1,2), a(1), e(1)\}$, the corresponding reduct $Ground(\mathcal{P}_0)^{I_0}$ contains rules g_1, g_2, g_3, g_4, g_9, plus $e(1) \leftarrow c(1,1), e(1) \leftarrow c(1,2)$, $e(2) \leftarrow c(2,1)$, and $e(2) \leftarrow c(2,2)$, obtained by canceling the negative literals from g_5, g_6, g_7, and g_8, respectively. We can thus verify that I_0 is an answer set for $Ground(\mathcal{P}_0)^{I_0}$ and therefore also an answer set for $Ground(\mathcal{P}_0)$ and $\mathcal{P}_0$.

Let us now consider the interpretation $I_1 = \{c(1,2), b(1), e(1)\}$, which is a model of $Ground(\mathcal{P}_0)$. The reduct $Ground(\mathcal{P}_0)^{I_1}$ contains rules g_1, g_2, g_3, g_4, g_9 plus both $e(2) \leftarrow c(2,1)$ and $e(2) \leftarrow c(2,2)$ (note that both g_5 and g_6 are deleted because $b(1) \in I_1$). I_1 is not an answer set of $Ground(\mathcal{P}_0)^{I_1}$ because $\{c(1,2), b(1)\} \subset I_1$ is. As a consequence, I_1 is not an answer set of P_0.

It can be verified that $\mathcal{P}_0$ has two answer sets, I_0 and $\{c(1,2), b(1)\}$. □

KNOWLEDGE REPRESENTATION AND REASONING IN ASP

ASP has been exploited in several domains, ranging from classical deductive databases to artificial intelligence. ASP can be used to encode problems in a declarative fashion; indeed, the power of disjunctive rules allows for expressing problems that are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows one to obtain uniform solutions over varying instances.

More importantly, many problems of comparatively high computational complexity can be solved in a natural manner by following a "Guess&Check" programming methodology, which was originally introduced in Ref. 29 and refined in Ref. 30. The idea behind this method can be summarized as follows: A database of facts is used to specify an instance of the problem, whereas a set of (usually disjunctive[4]) rules, called the "guessing part," is used to define the search space; solutions are then identified in the search space by another (optional) set of rules, called "checking part," which impose some admissibility constraint. Basically, the answer sets of the program, which combines the input database with the guessing part, represent "solution candidates" those candidates are then filtered, by adding the checking part, which guarantee that the answer sets of the resulting program represent precisely the admissible solutions for the input instance. To grasp the intuition behind the role of both the guessing and the checking parts, consider the following example.

Example 4.1. Suppose that we want to partition a set of persons in two groups, while avoiding that father and children belong to the same group. Following the guess&check methodology, we use a disjunctive rule to "guess" all the possible assignments of persons to groups as follows:

$$group(P, 1) \vee group(P, 2) \leftarrow person(P).$$

To understand what this rule does, consider a simple instance of the problem, in which there are two persons: *joe* and his father *john*. This instance is represented by four facts

$$person(john).\ person(joe).\ father(john, joe).$$

We can verify that the answer sets of the resulting program (facts plus disjunctive rule) correspond to all possible assignments of the three persons to two groups:

{person(john),person(joe),father(john,joe),group(john,1),group(joe,1)}
{person(john),person(joe),father(john,joe),group(john,1),group(joe,2)}
{person(john),person(joe),father(john,joe),group(john,2),group(joe,1)}
{person(john),person(joe),father(john,joe),group(john,2),group(joe,2)}

However, we want to discard assignments in which father and children belong to the same group. To this end, we add the checking part by writing the following constraint:

$$\leftarrow group(P1,G), group(P2,G), father(P1,P2).$$

The answer sets of the augmented program are then the intending ones, where the checking part has acted as a sort of filter:

{person(john),person(joe),father(john,joe),group(john,1),group(joe,2)}
{person(john),person(joe),father(john,joe),group(john,2),group(joe,1)}

□

In the following, we illustrate the usage of ASP as a tool for knowledge representation and reasoning by example. In particular, we first deal with a problem motivated by

[4]Some ASP variants use *choice rules* as guessing part (see Refs. 31–33). Moreover, in some cases, it is possible to emulate disjunction by unstratified normal rules by "shifting" the disjunction to the body (31–36), but this is not possible in general.

classical deductive database applications; then we exploit the "Guess&Check" programming style to show how a number of well-known harder problems can be encoded in ASP.

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

The input graph is encoded by assuming that A is represented by the binary relation $arc(X, Y)$, where a fact $arc(a, b)$ means that G contains an arc from a to b; i.e., $(a, b) \in A$; although the set of nodes V is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts.

The following program then defines a relation $reachable(X, Y)$ containing all facts $reachable(a, b)$ such that b is reachable from a through the arcs of the input graph G:

$$r_1: \ reachable(X, Y) \leftarrow arc(X, Y).$$
$$r_2: \ reachable(X, Y) \leftarrow arc(X, U), \ reachable(U, Y).$$

The first rule states that that node Y is reachable from node X if there is an arc in the graph from X to Y, whereas the second rule reprents the transitive closure by stating that node Y is reachable from node X if a node U exists such that U is directly reachable from X (there is an arc from X to U) and Y is reachable from U.

As an example, consider a graph represented by the following facts:

$$arc(1, 2). \quad arc(2, 3). \quad arc(3, 4).$$

The single answer set of the program reported above together with these three facts program is $\{reachable(1, 2), reachable(2, 3), reachable(3, 4), reachable(1, 3), reachable(2, 4), reachable(1, 4), arc(1, 2), arc(2, 3), arc(3, 4)\}$. The first three reported literals are inferred by exploiting the rule r_1, whereas the other literals containing the predicate *reachable* are inferred by using rule r_2.

In the following section, we describe the usage of the "Guess&Check" methodology.

Hamiltonian Path. Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does a path in G exist starting at a and passing through each node in V exactly once?

This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates *node* (unary) and *arc* (binary), and the starting node a is specified by the predicate *start* (unary). Then, the following program $\mathcal{P}_{hp}$ solves the *Hamiltonian Path* problem:

$$r_1: \ inPath(X, Y) \vee outPath(X, Y) \leftarrow arc(X, Y).$$
$$r_2: \ reached(X) \leftarrow start(X).$$
$$r_3: \ reached(X) \leftarrow reached(Y), inPath(X, Y).$$
$$r_4: \ \leftarrow inPath(X, Y), \ inPath(X, Y1), \ Y <> Y1.$$
$$r_5: \ \leftarrow inPath(X, Y), inPath(X1, Y), \ X <> X1.$$
$$r_6: \ \leftarrow node(X), \ \text{not}\, reached(X), \ \text{not}\, start(X).$$

The disjunctive rule (r_1) guesses a subset S of the arcs to be in the path, whereas the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate *reached* is defined, which specifies the set of nodes that are reached from the starting node. Doing this is very similar to reachability, but the transitivity is defined over the guessed predicate *inPath* using rule r_3. Note that as *reached* is completely determined by the guess for *inPath*, no further guessing is needed.

In the checking part, the first two constraints (namely, r_4 and r_5) ensure that the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: (*1*) there must not be two arcs starting at the same node, and (*2*) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.

Let us next consider an alternative program $\mathcal{P}'_{hp}$, which also solves the *Hamiltonian Path* problem, but intertwines the reachability with the guess:

$$r_1: \ inPath(X,Y) \vee outPath(X, Y) \leftarrow reached(X), arc(X, Y).$$
$$r_2: \ inPath(X, Y) \vee outPath(X, Y) \leftarrow start(X), \ arc(X, Y).$$
$$r_3: \ reached(X) \leftarrow inPath(Y, X).$$
$$r_4: \ \leftarrow inPath(X, Y), \ inPath(X, Y1), \ Y <> Y1.$$
$$r_5: \ \leftarrow inPath(X, Y), inPath(X1, Y), \ X <> X1.$$
$$r_6: \ \leftarrow node(X), \ \text{not}\, reached(X), \ \text{not}\, start(X).$$

Here, the two disjunctive rules (r_1 and r_2), together with the auxiliary rule r_3, guess a subset S of the arcs to be in the path, whereas the rest of the program checks whether S constitutes a Hamiltonian Path. Here, *reached* is defined in a different way. In fact, *inPath* is already defined in a way that only arcs reachable from the starting node will be guessed. The remainder of the checking part is the same as in $\mathcal{P}_{hp}$.

Ramsey Numbers. In the previous example, we have seen how a search problem can be encoded in an ASP program whose answer sets correspond to the problem solutions. We now build a program whose answer sets witness that a property does not hold; i.e., the property at hand holds if and only if the program has no answer set. We next apply the above programming scheme to a well-known problem of number and graph theory.

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

Ramsey numbers exist for all pairs of positive integers k and m (37). We next show a program $\mathcal{P}_{ra}$ that allows us to decide whether a given integer n is *not* the Ramsey Number $R(3,4)$. By varying the input number n, we can determine $R(3,4)$, as described below. Let $\mathcal{F}_{ra}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. $\mathcal{P}_{ra}$ is the following

program:

$$\begin{array}{ll} r_1: & blue(X,Y) \vee red(X,Y) \leftarrow arc(X,Y). \\ r_2: & \leftarrow red(X,Y), red(X,Z), red(Y,Z). \\ r_3: & \leftarrow blue(X,Y), blue(X,Z), blue(Y,Z), \\ & blue(X,W), blue(Y,W), blue(Z,W). \end{array}$$

Intuitively, the disjunctive rule r_1 guesses a color for each edge. The first constraint (r_2) eliminates the colorings containing a red clique (i.e., a complete graph) with three nodes, and the second constraint (r_3) eliminates the colorings containing a blue clique with four nodes. The program $\mathcal{P}_{ra} \cup \mathcal{F}_{ra}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is *not* $R(3,4)$; that is, $n < R(3, 4)$. On the other hand, if $\mathcal{P}_{ra} \cup \mathcal{F}_{ra}$ has no answer set, then $n \geq R(3,4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $R(3,4)$.

Strategic Companies. In the examples considered so far, the complexity of the problems is located at most on the first level of the Polynomial Hierarchy (38) (in NP or co-NP). We next demonstrate that also more complex problems, located at the second level of the Polynomial Hierarchy, can be encoded in ASP. To this end, we now consider a knowledge representation problem, inspired by a common business situation, which is known under the name *Strategic Companies* (39).

Suppose there is a collection $C = \{c_1, \ldots, c_m\}$ of companies c_i owned by a holding, a set $G = \{g_1, \ldots, g_n\}$ of goods, and for each c_i we have a set $G_i \subseteq G$ of goods produced by c_i and a set $O_i \subseteq C$ of companies controlling (owning) c_i. O_i is referred to as the *controlling set* of c_i. This control can be thought of as a majority in shares; companies not in C, which we do not model here, might have shares in companies as well. Note that, in general, a company might have more than one controlling set. Let the holding produce all goods in G; i.e., $G = \cup_{c_i \in C} G_i$.

A subset of the companies $C' \subseteq C$ is a *production-preserving* set if the following conditions hold: (*1*) The companies in C' produce all goods in G; i.e., $\cup_{c_i \in C'} G_i = G$. (*2*) The companies in C' are closed under the controlling relation; i.e. if $O_i \subseteq C'$ for some $i = 1, \ldots, m$, then $c_i \in C'$ must hold.

A subset-minimal set C', which is *production-preserving*, is called a *strategic set*. A company $c_i \in C$ is called *strategic*, if it belongs to some strategic set of C.

This notion is relevant when companies should be sold. Indeed, intuitively, selling any nonstrategic company does not reduce the economic power of the holding. Computing strategic companies is on the second level of the Polynomial Hierarchy (39).

In the following discussion, we consider a simplified setting as considered in Ref. 39, where each product is produced by at most two companies (for each $g \in G, |\{c_i | g \in G_i\}| \leq 2$) and each company is jointly controlled by at most three other companies; i.e., $|O_i| \leq 3$ for $i = 1, \ldots, m$. Assume that for a given instance of Strategic Companies, $\mathcal{F}_{st}$ contains the following facts:

- *company(c)* for each $c \in C$
- *prod_by*(g, c_j, c_k), if $\{c_i | g \in G_i\} = \{c_j, c_k\}$, where c_j and c_k may possibly coincide
- *contr_by*(c_i, c_k, c_m, c_n), if $c_i \in C$ and $O_i = \{c_k, c_m, c_n\}$, where $c_k, c_m,$, and c_n are not necessarily distinct.

We next present a program $\mathcal{P}_{st}$, which characterizes this hard problem using only two rules:

$$\begin{array}{ll} r_1: & start(Y) \vee start(Z) \leftarrow prod\_by(X,Y,Z). \\ r_2: & start(W) \leftarrow contr\_by(W,X,Y,Z), strat(X), \\ & start(Y), start(Z). \end{array}$$

Here *strat*(X) means that company X is a strategic company. The guessing part of the program consists of the disjunctive rule r_1, and the checking part consists of the normal rule r_2. The program $\mathcal{P}_{st}$ is surprisingly succinct, given that Strategic Companies is a hard problem.

The program $\mathcal{P}_{st}$ exploits the minimization that is inherent to the semantics of answer sets for the check whether a candidate set C' of companies that produces all goods and obeys company control is also minimal with respect to this property.

The guessing rule r_1 intuitively selects one of the companies c_1 and c_2 that produce some item g, which is described by *prod_by*(g, c_1, c_2). If there was no company control information, the minimality of answer sets would naturally ensure that the answer sets of $\mathcal{F}_{st} \cup \{r_1\}$ correspond to the strategic sets; no further checking would be needed. However, in case control information is available, the rule r_2 checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets.

The answer sets of $\mathcal{F}_{st} \cup \mathcal{P}_{st}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}_{st}$; company c is thus strategic iff *strat*(c) is in some answer set of $\mathcal{F}_{st} \cup \mathcal{P}_{st}$.

An important note here is that the checking "constraint" r_2 interferes with the guessing rule r_1: applying r_2 may "spoil" the minimal answer set generated by r_1. For example, suppose the guessing part gives rise to a ground rule

$$r_3 : strat(c1) \quad \vee \quad strat(c2) \leftarrow prod\_by(g, c1, c2).$$

and the fact *prod_by*(g, $c1$, $c2$) is given in $\mathcal{F}_{st}$. Now suppose the rule is satisfied in the guessing part by making *strat*($c1$) true. If, however, in the checking part an instance of rule r_2 is applied that derives *strat*($c2$), then the application of the rule r_3 to derive *strat*($c1$) is invalidated, as the minimality of answer sets implies that rule r_3 cannot justify the truth of *strat*($c1$), if another atom in its head is true.

FURTHER READING AND RELATED ISSUES

In this section, we consider some additional topics that allow the reader to have a broader picture of ASP. In particular, we introduce the general architecture of ASP systems, and we briefly describe several language extensions that have been proposed so far.

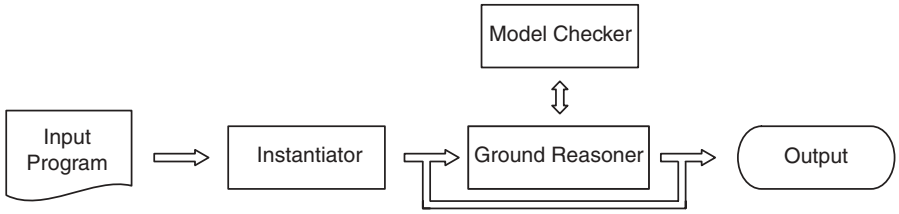

Figure 1. General architecture of an ASP system.

System Algorithms

Initially somewhat impeded by complexity considerations, reasonable algorithms and systems supporting ASP became available in the second half of the 1990s. The first widely used ones were Smodels (33,40), supporting nondisjunctive ASP, and DLV (30), supporting ASP (with disjunction) as defined in Ref 4. These two systems have been improved over the years and are still in widespread use. Later on, more systems for nondisjunctive ASP, like ASSAT (41,42), Cmodels (5), and Clasp (43) became available, and also more disjunctive ASP systems became available with the advent of GnT (44) and cmodels-3 (45).

Although, as discussed below, the systems do not use the same techniques, they basically agree on the general architecture depicted in Fig. 1.

The evaluation flow of the computation is outlined in detail. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system.[5]

In general, an input program $\mathcal{P}$ contains variables, and the first step of a computation of an ASP system is to eliminate these variables, generating a ground instantiation *ground*($\mathcal{P}$) of $\mathcal{P}$. This variable-elimination process is called *instantiation* of the program (or *grounding*) and is performed by the *Instantiator* module (see Fig. 1).

A naÿve Instantiator would produce the full ground instantiation *Ground*($\mathcal{P}$) of the input, which is, however, undesirable from a computational point of view, as in general many useless ground rules would be generated. All of the systems therefore employ different procedures, which are geared toward 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 compute directly its stable model (if it exists).

The subsequent computations, which constitute the nondeterministic part of an ASP system, are then performed on *ground*($\mathcal{P}$) by both the *Ground Reasoner* and the *Model Checker*. Roughly, the former produces some "candidate" answer set, whose stability is subsequently verified by the latter. The existing ASP systems mainly differ in the technique employed for implementing the *Ground Reasoner*. There are basically two approaches, which we will refer to as *search-based* and *rewriting-based*. In the search-based approach, the *Ground Reasoner* implements a backtracking search algorithm, which works directly on the ground instantiation of the input program. Search-based systems, like DLV and Smodels, are often referred to as "native" ASP systems, because the employed algorithms directly manipulate logic programs and are optimized for those. In the rewriting-based approach, the *Ground Reasoner* transforms the ground program into a propositional formula and then invokes a Boolean satisfiability solver for finding answer set candidates.

As previously pointed out, the *Model Checker* verifies whether an answer set candidate at hand is an answer set for the input program. This task is as hard as the problem solved by the Ground Reasoner for disjunctive programs, whereas it is trivial for nondisjunctive programs. However, there is also a class of disjunctive programs, called Head-Cycle-Free programs (34), for which the task solved by the Model Checker is provably simpler, which is exploited in the system algorithms.

Finally, once an answer set has been found, ASP systems typically print it in text format, and possibly the *Ground Reasoner* resumes in order to look for additional answer sets.

Language Extensions

The work on ASP started with standard rules, but fairly soon implementations extending the basic language started to emerge. The most important extensions to the ASP language can be grouped in three main classes:

- *Optimization constructs*
- *Aggregates*
- *Preference handling*

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). Two extensions of ASP have been conceived for solving optimization problems: *weak constraints* (30,46) and *optimize statements* (33).

In the basic language, constraints are rules with an empty head and represent a condition that *must* be satisfied, and for this reason, they are also called *strong* constraints. Contrary to strong constraints, *weak constraints* allow us to express desiderata, that is, conditions that

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

should be satisfied. Thus, they may be violated, and their semantics involves minimizing the number of violated instances of weak constraints. In other words, the presence of strong constraints modifies the semantics of a program by discarding all models that do not satisfy some of them, whereas weak constraints identify an approximate solution, that is, one in which (weak) constraints are satisfied as much as possible.

From a syntactic point of view, a weak constraint is like a strong one where the implication symbol $\leftarrow$ is replaced by $\leftsquigarrow$. The informal meaning of a weak constraint $\leftsquigarrow B$ 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). When not specified, the weak constraint is assumed to have weight 1 and priority level 1, respectively.

In this case, we are interested in the answer sets that minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level and, among them, those that minimize the sum of weights of the violated weak constraints in the next lower level, and so on. In other words, the answer sets are considered along a lexicographic ordering along the priority levels over the sum of weights of violated weak constraints. Therefore, higher values for weights and priority levels allow for marking weak constraints of higher importance (e.g., the most important constraints are those having the highest weight among those with the highest priority level).

As an example, consider the Traveling Salesman Problem (TSP). TSP is a variant of the Hamiltonian Cycle problem considered earlier, which amounts to finding the shortest (minimal cost) Hamiltonian cycle in a directed *numerically labeled* graph. This problem can be solved by adapting the encoding of the Hamiltonian cycle problem given in Section 4 in order to deal with labels, by adding only one weak constraint.

Suppose again that the graph G is specified by predicates *node* (unary) and *arc* (ternary), and that the starting node is specified by the predicate *start* (unary).

The ASP program with weak constraints solving the TSP problem is thus as follows:

$$
\begin{aligned}
r_1&: \ inPath(X,Y,C) \vee outPath(X,Y,C) \leftarrow arc(X,Y,C).\\
r_2&: \ reached(X) \leftarrow start(X).\\
r_3&: \ reached(X) \leftarrow reached(Y), inPath(Y,X,C).\\
r_4&: \ \leftarrow inPath(X,Y,\_), inPath(X,Y1,\_), Y <> Y1.\\
r_5&: \ \leftarrow inPath(X,Y,\_), inPath(X1,Y,\_), X <> X1.\\
r_6&: \ \leftarrow node(X), \text{ not } reached(X).\\
r_7&: \ \leftsquigarrow inPath(X,Y,C).[C,1]
\end{aligned}
$$

The last weak constraint *(r_7)* states the preference to avoid taking arcs with high cost in the path, and has the effect of selecting those answer sets for which the total cost of arcs selected by *inPath* (which coincides with the length of the path) is the minimum (i.e., the path is the shortest).

The TSP encoding provided above is an example of the "guess, check and optimize" programming pattern (30), which extends the original "guess and check" (see Section 4) by adding an additional "optimization part," which mainly contains weak constraints. In the example above, the optimization part contains only the weak constraint r_7.

Optimize statements are syntactically somewhat simpler. They assign numeric values to a set of ground literals, and thereby select those answer sets for which the sum of the values assigned to literals that are true in the respective answer sets are maximal or minimal. It is not hard to see that weak constraints can emulate optimize statements, but not vice versa.

Aggregates. There are some simple properties, often originating in real-world applications, which cannot be encoded in a simple and natural manner using ASP. 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 became used in real applications, it became apparent that aggregates are needed also here. First, cardinality and weight constraints (33), which are special cases of aggregates, have been introduced. However, in general, one might want to use also other aggregates (like minimum, maximum, or average), and it is not clear how to generalize the framework of cardinality and weight constraints to allow for arbitrary aggregates. To overcome this deficiency, ASP has been extended with special atoms handling aggregate functions (47–53). Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

An *aggregate function* is of the form $f(S)$, where S is a set term of the form $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of standard atoms, and f is an *aggregate function symbol*.

The most common aggregate functions compute the number of terms, the sum of non-negative integers, and the minimum/maximum term in a set.

Aggregates are especially useful when real-world problems have to be dealt with. Consider the following example application[6]. A project team has to be built from a set of employees according to the following specifications:

1. At least a given number of different skills must be present in the team.
2. The sum of the salaries of the employees working in the team must not exceed the given budget.

Suppose that our employees are provided by several facts of the form *emp*(*EmpId, Skill, Salary*); the minimum

[6]In the example, we adopted the syntax of the DLV system, the same aggregate functions can be specified also by exploiting other ASP dialects.

number of different skills and the budget are specified by the facts $nSkill(N)$ and $budget(B)$. We then encode each property stated above by an aggregate atom, and we enforce it by an integrity constraint:

r_1: $in(I) \vee out(I) \leftarrow emp(I,Sk,Sa)$.
r_3: $\leftarrow nSkill(M), not\ \#count\{Sk : emp(I,Sk,Sa), in(I)\} >= M$.
r_4: $\leftarrow budget(B), not\ \#sum\{Sa,I : emp(I,Sk,Sa), in(I)\} <= B$.

Intuitively, the disjunctive rule "guesses" whether an employee is included in the team or not, whereas the two constraints correspond one-to-one to the requirements. Indeed, the function `#count` counts the number of employees in the team, whereas `#sum` sums the salaries of the employees that are part of the team.

Note that thanks to the aggregates, the translation of the specifications is straightforward.

Preference Handling. ASP programs usually follow a "guess and check" programming pattern (see Section 4), where a set of rules (the guessing part) is used to guess a solution (or equivalently, to generate answer set candidates), whereas another set of rules, called the checking part, is added to discard solutions that are not admissible. This methodology allows the programmer to distinguish between solutions and nonsolutions. However, in many realistic applications, the possibility to make more fine-grained distinctions is required, and in particular, distinctions between more and less preferred solutions are needed (see Ref. 54 for a discussion). For this reason, there has been a substantial amount of work on extending ASP programs with preferences, and in particular, the major focus has been on qualitative approaches. This stems from the fact that for a variety of applications, numerical information is hard to obtain (preference elicitation is difficult) and often turns out to be unnecessary (see Ref. 54). Still, language extensions based on quantitative information, such as the weak constraints mentioned above, emulate qualitative preferences under certain conditions, and vice versa. There are two basic possibilities for representing qualitative preferences. In one approach, the preference is specified among rules, mirroring the fact that some rules may be more reliable than others, and striving to use a set of rules that is as preferred as possible for giving a reason to an answer. In the second approach, the preferences are specified among literals, reflecting information on either the likelihood or the desirability of the affirmations represented by the literals.

In the first kind of formalisms, preferences are specified by means of an ordering among rules. Formally, an *ordered logic program* is a pair $(\Pi, <)$ where Π is a logic program and $< \subseteq (\Pi \times \Pi)$ is a strict partial order. Given $r_1, r_2 \in \Pi$, the relation $r_1 < r_2$ expresses that r_2 has higher priority than r_1.

For example, consider the following program:

$$r_1: \neg a. \quad r_2: b \leftarrow \neg a, not\ c. \quad r_3: c \leftarrow not\ b.$$

This program has two answer sets, one given by $\{\neg a, b\}$ and the other given by $\{\neg a, c\}$. For the first answer set, rules r_1 and r_2 are applied; for the second, r_1 and r_3. However, assume that we have reason to prefer r_2 to r_3 expressed by $r_3 < r_2$. In this case, we would want to obtain just the first answer set and we say say that the first is a *preferred answer set*.

In general, defining which answer sets should be the preferred ones in this setting is not always as obvious as in the example above, and indeed several approaches have been proposed. A comprehensive comparison of three major semantics, defined by Delgrande, Schaub, Tompits (55), by Brewka and Eiter (56), and by Wang, Zhou, Lin (57), has been presented in Ref. 58.

In the second representational approach, preferences are represented among atoms, literals, or formulas. One way of specifying this has been proposed in Ref. 59, which is the use of *ordered disjunction* in rule heads. In particular, the operator $\times$ in rule heads acts as a disjunction also specifying preferences. The meaning of a rule $a_1 \times \cdots \times a_n \leftarrow body$, is that if the body is satisfied, then some a_i must be in the answer set, most preferably a_1, if this is impossible, then a_2, and so on. The formal semantics is defined by means of answer sets of split programs and of rule satisfaction degrees. There are some degrees of freedom when aggregating the satisfaction degrees of several rules, leading to different semantics, the main ones being cardinality-based, set-inclusion-based, and Pareto-based.

In the ordered disjunction approach, the construction of answer sets is amalgamated with the expression of preferences. *Optimization programs* (60), on the other hand, strictly separate these two aspects. An optimization program is a pair (P_{gen}, P_{pref}). Here, P_{gen} is an arbitrary logic program used to generate answer sets. All we require is that it produces sets of literals as its answer sets. P_{pref} is a preference program. Preference programs consist of preference rules of the form $c_1 > \cdots > c_n \leftarrow body$, where the c_i are Boolean combinations of literals built from $\vee$, $\wedge$, $\neg$ and not. As in the case of ordered disjunction, the semantics of these programs is based on the degree of satisfaction of preference rules, and as in the case of ordered disjunctions, there are several options for aggregating these satisfaction degrees for defining semantics.

Another ASP extension suitable for preference handling has been presented in Ref. 61. There, 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, which are expressions of the form: $r : a_1 \vee \cdots \vee a_n \stackrel{+}{\leftarrow} body\ (n \geq 1)$. The intuitive meaning of the CR-rule r is as follows: 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.

Other Extensions. ASP has been extended in other directions in order to meet the requirements of different application domains; hence, there is a number of interesting languages having the roots on ASP. For instance, ASP has been exploited for defining and implementing *action languages* (i.e., languages conceived for dealing with

actions and change) $\mathcal{K}$ (62), and $\mathcal{E}$ (63), whereas in Ref. 64, a framework for *abduction with penalization* has been proposed and implemented as a front-end for the ASP system DLV. A logic language called *ID-Logic* (65) has been introduced to deal with classical logic with inductive definitions (which correspond semantically to logic rules). Other ASP extensions have been conceived to deal with *Ontologies* (i.e., abstract models of a complex domain). In particular, in Ref. 66, 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 Ref. 67, an open world semantics for ASP programs has been proposed. Moreover, in Ref. 68, an extension of ASP, called *HEX-Programs*, which supports higher order atoms as well as external atoms has been 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) (69) ontologies. *Template predicates* have been introduced in Ref. 70. Template predicates 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. Finally, nested programs, allowing for nested logical expressions to occur in rules, have also been studied (71,72).

Applications

Answer set programming has been successfully applied to many areas, including:

- *Information integration*. ASP has been exploited for supporting consistent query answering, in information integration systems under the so-called Global-as-View approach (73–75), also in the presence of data inconsistencies and data incompleteness.
- *Configuration and verification management*. In product configuration (76), ASP has been used as a declarative semantics providing formal definitions for main concepts in product configuration, including configuration models, requirements, and valid configurations. And, in particular, in the field of software configuration, a prototype configurator for the complete Debian Linux system distribution has been implemented by using ASP (17).
- *Knowledge management*. ASP has a strong potential for exploitation in the area of knowledge management and semantic technologies.

  An ASP-based system for ontology representation and reasoning, called OntoDLV (66), is employed in many real-world applications, ranging from e-learning to enterprise ontologies and agent-based applications. In Ref. 78, an ASP-based approach to the problem of recognizing and extracting information from unstructured documents has been presented. In Refs. 79 and 80, a system for content classification, called OLEX, is presented, which exploits ASP to extract concepts and semantic metadata from documents.
- *Security engineering*. In Ref. 81, it is shown how security protocols can be specified and verified efficiently and effectively by embedding reasoning about actions into logic programming. In particular, two significant case studies in protocol verification have been modeled: the classical Needham–Schroeder public-key protocol and the Aziz–Diffie key agreement protocol for mobile communication.

Moreover, applications from various areas can be found in the literature, including auctions (82), scheduling (83), policy description (84), workflow management (85), outlier detection (86), linguistics (87), multiagent systems (88–90), and e-learning (90).

Concluding, ASP is an appealing tool for knowledge representation and reasoning, and thanks to the applicability of the implementations of ASP solvers to real-world problems, ASP is tackling many industrially relevant applications.

It is worth noting that ASP systems are currently away from comfortably enabling the development of industry-level applications, and like any other programming language, ASP needs tools and development methodologies to facilitate and improve the coding process. At the time of this writing, the field of software engineering for ASP has been already settled by the ASP community (91), and it is currently evolving. Indeed, both methodologies (see Section 4) and prototype tools are already available (see Refs. 66, 70, and 91–94).

BIBLIOGRAPHY

1. J. McCarthy, Programs with common sense, *Proc. Teddington Conference on the Mechanization of Thought Processes*, Her Majesty's Stationery Office, 1959.
2. S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Reading, MA: Addison-Wesley, 1995.
3. A. Colmerauer and P. Roussel, *The Birth of Prolog*. New York: ACM, 1996.
4. M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases. *New Generation Computing*, **9**: 365–385, 1991.
5. C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge, UK: Cambridge University Press, 2003.
6. M. Gelfond and N. Leone, Logic programming and knowledge representation—the A-Prolog perspective. *Artif. Intell.*, **138**(1–2): 3–38, 2002.
7. K. Marriott and P. J. Stuckey, *Programming with Constraints: An Introduction*. Cambridge, MA: MIT Press, 1998.
8. A. C. Kakas, R. A. Kowalski, and F. Toni, Abductive logic programming. *J. Logic Computation*, **2**(6): 719–770, 1992.
9. J. McCarthy and P. J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*. Edinburgh, Scotland: Edinburgh University Press, 1969.
10. J. Alan Robinson, A machine-oriented logic based on the resolution principle. *J. ACM*, **12**(1): 23–41, 1965.
11. R. A. Kowalski, Predicate logic as programming language, *IFIP Congress*, 1974, pp. 569–574.

12. R. A. Kowalski, Algorithm = logic + control. *Commun. ACM*, **22**(7): 424–436, 1979.
13. International Organization for Standardization, *ISO/IEC 13211-1:1995: Information technology—Programming languages—Prolog—Part 1: General core*. International Organization for Standardization, Geneva, Switzerland, 1995.
14. M. H. van Emden and R. A. Kowalski, The semantics of predicate logic as a programming language. *J. ACM*, **23**(4): 733–742, 1976.
15. K. L. Clark, Negation as Failure, in Herve? Gallaire and Jack Minker (eds.), *Logic and Data Bases*. New York: Plenum Press, 1978.
16. R. Reiter, On closed world data bases, in H. Gallaire and J. Minker (eds.), *Logic and Data Bases*. New York: Plenum Press, 1978.
17. K. R. Apt, H. A. Blair, and A. Walker, Towards a theory of declarative knowledge, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Washington, DC: Morgan Kaufmann, 1988.
18. A. Van Gelder, Negation as failure using tight derivations for general logic programs, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*. Washington, DC: Morgan Kaufmann, 1988.
19. A. Van Gelder, K. A. Ross, and J. S. Schlipf, Unfounded sets and well-founded semantics for general logic programs, *Proc. Seventh Symposium on Principles of Database Systems (PODS'88)*, 1988, pp. 221–230.
20. V. Wiktor Marek and M. Truszczyn?ski, *Nonmonotonic Logics—Context-Dependent Reasoning*. New York: Springer-Verlag, 1993.
21. M. Gelfond, On stratified autoepistemic theories, *Proc. Sixth National Conference on Artificial Intelligence (AAAI-87)*, 1987, pp. 207–211.
22. N. Bidoit and C. Froidevaux, Minimalism subsumes default logic and circumscription in stratified logic programming, *Proc. Symposium on Logic in Computer Science (LICS '87)*, June 1987, pp. 89–97. IEEE.
23. M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, Cambridge, MA, 1988, pp. 1070–1080.
24. A. Van Gelder, K. A. Ross, and J. S. Schlipf, The well-founded semantics for general logic programs. *J. ACM*, **38**(3): 620–650, 1991.
25. J. Minker, On indefinite data bases and the closed world assumption, in D. W. Loveland (ed.), *Proceedings 6th Conference on Automated Deduction (CADE '82)* volume 138 of *Lecture Notes in Computer Science*. New York. Springer, 1982.
26. A. H. Yahya and L. J. Henschen, Deduction in non-horn databases. *J. Automated Reasoning*, **1**(2): 141–160, 1985.
27. T. C. Przymusinski, Stable semantics for disjunctive programs. *New Generation Computing*, **9**: 401–424, 1991.
28. K. Wang and L. Zhou, Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Trans. Computational Logic*, **6**(2), April 2005.
29. T. Eiter, W. Faber, N. Leone, and G. Pfeifer, Declarative problem-solving using the DLV system, in J. Minker (ed.), *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, 2000.
30. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, The DLV System for knowledge representation and reasoning. *ACM Trans. Computational Logic*, **7**(3): 499–562, July 2006.
31. I. Niemelð and P. Simons, Smodels—an implementation of the stable model and well-founded semantics for normal logic programs, in J. Dix, U. Furbach, and A. Nerode (eds.), *Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), volume 1265 of Lecture Notes in AI (LNAI)*, Dagstuhl, Germany, July 1997, pp. 420–429.
32. T. Syrjðnen, Lparse 1.0 User's Manual, 2002. Available: http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.
33. P. Simons, I. Niemelð, and T. Soininen, Extending and implementing the stable model semantics. *Artif. Intell.* **138**: 181–234, June 2002.
34. R. Ben-Eliyahu and R. Dechter, Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, **12**: 53–87, 1994.
35. J. Dix, G. Gottlob, and V. Wiktor Marek, Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, **28**: 87–100, 1996.
36. N. Leone, P. Rullo, and F. Scarcello, Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Inform. Computat.*, **135**(2): 69–112, June 1997.
37. S. P. Radziszowski, Small ramsey numbers. *Electronic J. Combinatorics*, **1**, 1994.
38. C. H. Papadimitriou, *Computational Complexity*. Reading, MA: Addison-Wesley, 1994.
39. M. Cadoli, T. Eiter, and G. Gottlob, Default logic as a query language. *IEEE Trans. Knowledge Data Eng.*, **9**(3): 448–463, May/June 1997.
40. P. Simons, Smodels Homepage, since 1996. Available: http://www.tcs.hut.fi/Software/smodels/.
41. Y. Zhao, ASSAT homepage, since 2002. Available: http://assat.cs.ust.hk/.
42. F. Lin and Y. Zhao, ASSAT: Computing answer sets of a logic program by SAT solvers, *Proc. Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002.
43. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, Conflict-driven answer set solving, *Proc. Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Morgan Kaufmann Publishers, January 2007, pp. 386–392.
44. T. Janhunen, I. Niemelð, D. Seipel, P. Simons, and J.-H. You, Unfolding partiality and disjunctions in stable model semantics. Technical Report cs.AI/0303009, arXiv.org, March 2003.
45. Y. Lierler, Disjunctive answer set programming via satisfiability, in C. Baral, G. Greco, N. Leone, and G. Terracina (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, pp. 447–451.
46. F. Buccafurri, N. Leone, and P. Rullo, Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowledge Data Eng.*, **12**(5): 845–860, 2000.
47. F. Calimeri, W. Faber, N. Leone, and S. Perri, Declarative and computational properties of logic programs with aggregates, in *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, August 2005, pp. 406–411.
48. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer, Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, Acapulco, Mexico, August 2003, pp. 847–852.
49. M. Denecker, N. Pelov, and M. Bruynooghe, Ultimate well-founded and stable model semantics for logic programs with

aggregates, in Philippe Codognet (ed.), *Proc. 17th International Conference on Logic Programming*, New York: Springer Verlag, 2001.

50. W. Faber and N. Leone, On the complexity of answer set programming with aggregates, in C. Baral, G. Brewka, and J. S. Schlipf (eds.), *Logic Programming and Nonmonotonic Reasoning—9th International Conference, LPNMR 2007, volume 4483 of Lecture Notes in AI (LNAI)*, Tempe, AZ, May 2007, pp. 97–109.
51. W. Faber, N. Leone, and G. Pfeifer, Recursive aggregates in disjunctive logic programs: Semantics and complexity, in J. Jºlio Alferes and J. Leite (eds.), *Proc. 9th European Conference on Artificial Intelligence (JELIA 2004), volume 3229 of Lecture Notes in AI (LNAI)*, Springer Verlag, September 2004, pp. 200–212.
52. L. Hella, L. Libkin, J. Nurmonen, and L. Wong, Logics with aggregate operators. *J. ACM*, **48**(4): 880–907, 2001.
53. N. Pelov, M. Denecker, and M. Bruynooghe, Well-founded and stable semantics of logic programs with aggregates. *Theory Practice of Logic Programming*. In Press.
54. G. Brewka, Answer sets: From constraint programming towards qualitative optimization, in V. Lifschitz and I. Niemelð (eds.), *Proc. 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of LNAI, Springer, January 2004, pp. 34–46.
55. J. P. Delgrande, T. Schaub, and H. Tompits, A framework for compiling preferences in logic programs. *Theory Practice Logic Programming*, **3**(2): 129–187, March 2003.
56. G. Brewka and T. Eiter, Preferred answer sets for extended logic programs. *Artif. Intell.*, **109**(1–2): 297–356, 1999.
57. K. Wang, L. Zhou, and F. Lin, Alternating fixpoint theory for logic programs with priority, *Computational Logic—CL 2000, First International Conference, Proceedings, volume 1861 of Lecture Notes in AI (LNAI)*, London, UK, July 2000.
58. T. Schaub and K. Wang, A comparative study of logic programs with preference, in *Proc. Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, Seattle, WA, August 2001, pp. 597–602.
59. G. Brewka, Logic programming with ordered disjunction, in *Proc. 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, April 2002, pp. 67–76.
60. G. Brewka, I. Niemelð, and M. Truszczyn?ski, Answer set optimization, in G. Gottlob and T. Walsh (eds.), *IJCAI-03, Proc. of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico*, Morgan Kaufmann, August 2003, pp. 867–872.
61. M. Balduccini and M. Gelfond, Logic programs with consistency-restoring rules, in M. A. Williams P. Doherty, J. McCarthy (eds.), *International Symposium on Logical Formaliza-tion of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, 2003.
62. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Computational Logic*, **5**(2): 206–263, April 2004.
63. Y. Dimopoulos, A. C. Kakas, and L. Michael, Reasoning about actions and change in answer set programming programs, in V. Lifschitz and I. Niemelð (eds.), *Proceedings of Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, volume 2 of LNCS*. New York: Springer, 2004.
64. S. Perri, F. Scarcello, and N. Leone, Abductive logic programs with penalization: Semantics, complexity and implementation. *Theory Practice of Logic Programming*, **5**(1–2): 123–159, 2005.
65. M. MariŠn, D. Gilis, and M. Denecker, On the relation between ID-Logic and answer set programming, in J. Jºlio Alferes and J. Alexandre Leite (eds.), *Proceedings of Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, volume 3229 of Lecture Notes in Computer Science*, Springer, September 2004, pp. 108–120.
66. F. Ricca and N. Leone, Disjunctive logic programming with types and objects: The DLV+ System. *J. Appl. Logics*, **5**(3): 545–573, 2007.
67. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir, Semantic web reasoning with conceptual logic programs, *Proc. Rules and Rule Markup Languages for the Semantic Web: Third International Workshop, RuleML 2004*, Hiroshima, Japan, November 2004, pp. 113–127.
68. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, A uniform integration of higher-order reasoning and external evaluations in answer set programming, *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, Edinburgh, UK, August 2005.
69. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.), *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge, UK: Cambridge University Press, 2003.
70. F. Calimeri, G. Ianni, G. Ielpa, A. Pietramala, and M. Carmela Santoro, A system with template answer set programs, in *JELIA*, 2004, pp. 693–697.
71. D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran, A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report, *Proc. 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, 2002.
72. D. Pearce, H. Tompits, and S. Woltran, Encodings for equilibrium logic and logic programs with nested expressions, in P. Brazdil and A. Jorge (eds.), *10th Proc. Portuguese Conference on Artificial Intelligence (EPIA 2001)*, December 2001, pp. 306–320.
73. D. Lembo, M. Lenzerini, and R. Rosati, Integrating inconsistent and incomplete data sources, *Proc. of SEBD 2002*, Portoferraio, Isola d'Elba, 2002, pp. 299–308.
74. D. Lembo, M. Lenzerini, and R. Rosati, Source inconsistency and incompleteness in data integration, *Proc. Knowledge Representation meets Databases International Workshop (KRDB-02)*, Toulouse, France, 2002. CEUR Electronic Workshop Proceedings. Available: http://sunsite. informatik. rwth-aachen.de/Publications/CEUR-WS/Vol-54/.
75. N. Leone, G. Gottlob, R. Rosati, T. Eiter, W. Faber, M. Fink, G. Greco, G. Ianni, E. Kalka, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, M. Ruzzi, W. Staniszkis, and G. Terracina, The INFOMIX System for advanced integration of incomplete and inconsistent data, *Proc. 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, Baltimore, MD, June 2005, pp. 915–917.
76. T. Soininen and I. Niemelð, Developing a declarative rule language for applications in product configuration, in G. Gupta (ed.), *Proc. 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), volume 1551 of Lecture Notes in Computer Science*, Springer, 1999, pp. 305–319.
77. T. Syrjðnen, A Rule-Based Formal Model for Software Configuration. Technical Report A55, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Espoo, Finland, 1999.
78. M. Ruffolo, N. Leone, M. Manna, D. Sacca, and A. Zavatto, Exploiting ASP for semantic information extraction, in M. de Vos and A. Provetti (eds.), *Proceedings ASP05—Answer Set*

Programming: Advances in Theory and Implementation, Bath, UK, July 2005, pp. 248–262.

79. C. Cumbo, S. Iiritano, and P. Rullo, Reasoning-based knowledge extraction for text classification, in *Proceedings of Discovery Science, 7th International Conference*, Padova, Italy, October 2004, pp. 380–387.
80. R. Curia, M. Ettorre, S. Iiritano, and P. Rullo, Textual document per-processing and feature extraction in OLEX, in *Proceedings of Data Mining 2005*, Skiathos, Greece, 2005.
81. L. Carlucci Aiello and F. Massacci, Verifying security protocols as planning in logic programming. *ACM Trans. Computat. Logic*, **2**(4): 542–580, 2001.
82. C. Baral and C. Uyan, Declarative specification and solution of combinatorial auctions using logic programming, in T. Eiter, W. Faber, and M. Truszczyn?ski (eds.), *Proc. 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of Lecture Notes in AI (LNAI), Springer Verlag, 2001, pp. 186–199.
83. W. Faber, N. Leone, and G. Pfeifer, Representing school timetabling in a disjunctive logic programming language, in U. Egly and H. Tompits (eds.), *Proc. 13th Workshop on Logic Programming (WLP'98)*, Vienna, Austria, October 1998, pp. 43–52.
84. E. Bertino, A. Mileo, and A. Provetti, User preferences VS minimality in PPDL, in F. Buccafurri (ed.), *Proc. Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, September 2003, pp. 110–122.
85. G. Greco, A. Guzzo, and D. SaccÁ, A logic programming approach for planning workflows evolutions, in F. Buccafurri (ed.), *Proc. Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, September 2003, pp. 75–85.
86. G. Greco, S. Greco, and E. Zumpano, A logical framework for querying and repairing inconsistent databases, *IEEE Trans. Knowledge Data Eng.*, **15**(6): 1389–1408, 2003.
87. E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe, Reconstructing the evolutionary history of indo-european languages using answer set programming, in V. Dahl and P. Wadler (eds.), *Practical Aspects of Declarative Languages, 5th International Symposium (PADL 2003),* volume 2562 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 160–176.
88. F. Buccafurri and G. Caminiti, A social semantics for multi-agent systems, in C. Baral, G. Greco, N. Leone, and G. Terracina (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, pp. 317–329.
89. S. Costantini and A. Tocchio, The dali logic programming agent-oriented language, in J. Jºlio Alferes and J. Leite (eds.), *Proc. 9th European Conference on Artificial Intelligence (JELIA 2004), volume 3229 of Lecture Notes in AI (LNAI)*, Springer Verlag, September 2004, pp. 685–688.
90. A. Garro, L. Palopoli, and F. Ricca, Exploiting agents in e-learning and skills management context. *AI Commun. Eur. J. Artif. Intell.*, **19**(2): 137–154, 2006.
91. M. De Vos and T. Schaub (eds.), *SEA '07: Software Engineering for Answer Set Programming*, volume 281. CEUR, 2007. Available: http://CEUR-WS.org/Vol-281/.
92. M. Brain and M. De Vos, Debugging logic programs under the answer set semantics, in M. de Vos and A. Provetti (eds.), *Proc. ASP05—Answer Set Programming: Advances in Theory and Implementation*, Bath, UK, July 2005.
93. O. El-Khatib, E. Pontelli, and T. Cao Son, Justification and debugging of answer set programs in ASP, in C. Jeffery, J.-D. Choi, and R. Lencevicius (eds.), *Proc. Sixth International Workshop on Automated Debugging*, California, September 2005.
94. F. Ricca, The DLV Java Wrapper, in M. Vos de and A. Provetti (eds.), *Proceedings ASP03—Answer Set Programming: Advances in Theory and Implementation*, Messina, Italy, September 2003. Available: http://CEUR-WS.org/Vol-78/.

FURTHER READING

Y. Babovich and M. Maratea, Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. Available: http://www.cs.utexas.edu/users/tag/cmodels.html, 2003.

J. Dix, G. Gottlob, and V. Wiktor Marek, Causal models for disjunctive logic programs, in Pascal Van Hentenryck (ed.), *Proc. 11th International Conference on Logic Programming (ICLP'94)*, Santa Margherita Ligure, Italy, June 1994.

J. McCarthy, *Formalization of Common Sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.

J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*. Washington, DC: Morgan Kaufmann Publishers, Inc., 1988.

WOLFGANG FABER
NICOLA LEONE
FRANCESCO RICCA
Department of Mathematics
University of Calabria
Rende, Italy