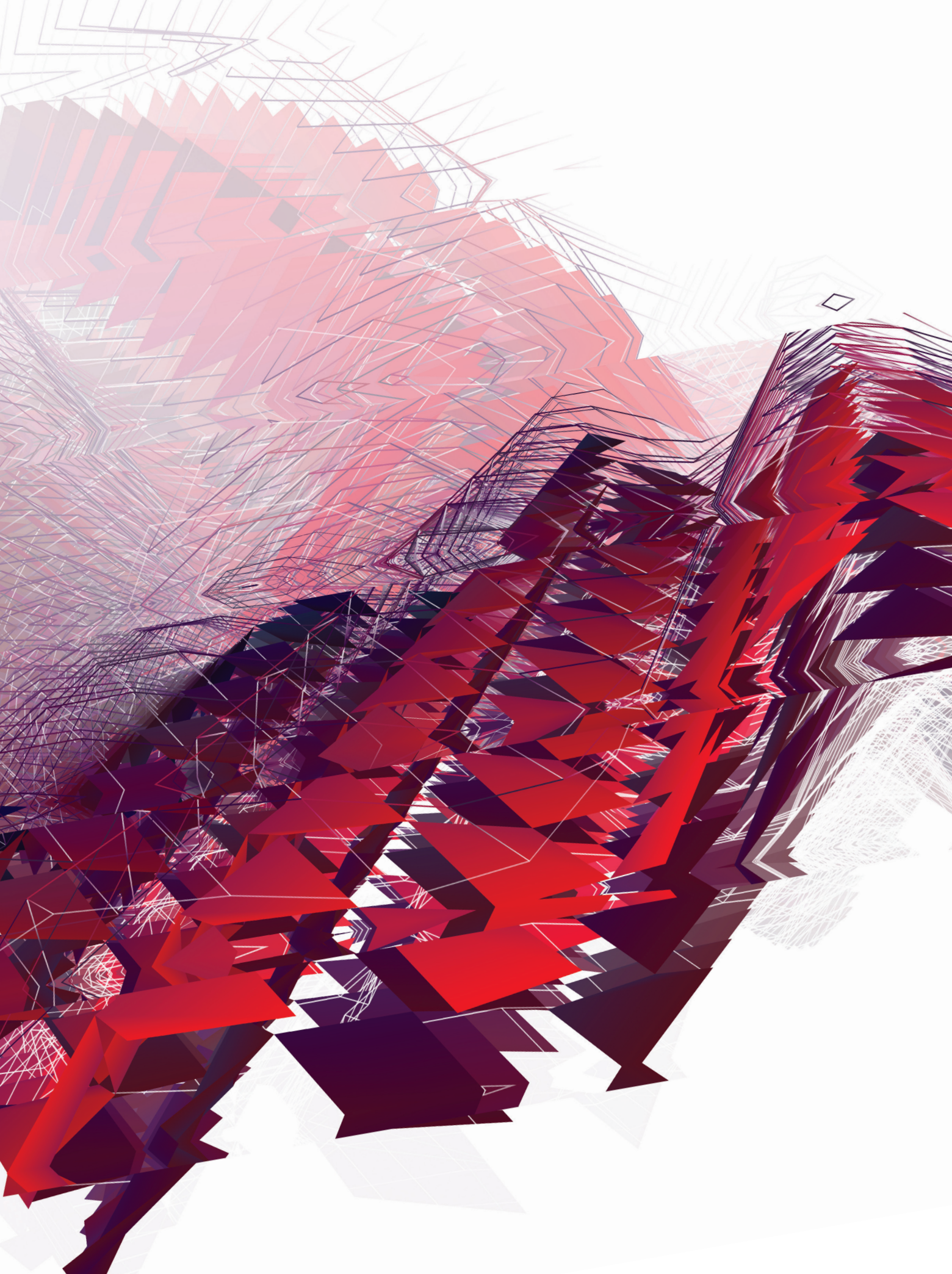


DOI:10.1145/2043174.2043195

The motivation and key concepts behind answer set programming—a promising approach to declarative problem solving.

BY GERHARD BREWKA, THOMAS EITER, AND MIROSŁAW TRUSZCZYŃSKI

Answer Set Programming at a Glance

CAN SOLVING HARD computational problems be made easy? If we restrict the scope of the question to computational problems that can be stated in terms of constraints over *binary* domains, and if we understand "easy" as "using a simple and intuitive modeling language that comes with software for processing programs in the language," then the answer is Yes! *Answer Set Programming* (ASP, for short) fits the bill.

While already well represented at research conferences and workshops, ASP has been around for barely more than a decade. Its origins, however, go back a long time; it is an outcome of years of research in knowledge representation, logic programming, and constraint satisfaction—areas that sought and studied declarative languages to model domain knowledge, as well as general-purpose computational tools for processing programs and theories that represent problem specifications in these languages. ASP borrows from each of these areas, all the time aiming

ILLUSTRATION BY GWEN VANHEE

to maintain a balance between expressivity, ease of use, and computational effectiveness. To give just a few examples, emerging applications in molecular biology, decision support systems for space shuttle controllers, and team building at Gioia Tauro Seaport (see sidebar here) bear witness to its potential.

Programs and Answer Sets

We start our ASP discussion with the propositional setting. The building blocks for programs are *atoms*, *literals*, and *rules*. *Atoms* are elementary propositions (factual statements) that may be true or false; *literals* are atoms a and their negations *not* a. *Rules* are expressions of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n \qquad (1)$$

where a and all b_i's and c_j's are atoms. Intuitively, a rule (1) is a justification to "establish" or "derive" that a (the so-called *head*) is true, if all literals to the right of $\leftarrow$ (the so-called *body*) are true in the following sense: a non-negated literal b_i is true if it has a derivation, a negated one, *not* c_j, is true if the atom c_j does not have one. For instance, the rule

light_on $\leftarrow$ *power_on, not broken*

informally means we can assert that the light is on, if we established the power is on and there is no reason to think the lamp is broken. Rules may have no body. For instance, we may have a rule:

power_on $\leftarrow$.

Such rules are called *facts*, as the head is unconditionally true, and the arrow $\leftarrow$ is typically omitted.

Programs are finite collections of rules. They are thought of as "justifications" for sets of atoms that contain precisely those atoms that can be established. It is important to point out that *not* is not a standard negation operator. Rather, it is meant to stand for a modality "non-derivable." Looking at the small program with the two rules mentioned here, *power_on* should be derived (as it is given as a fact), while intuitively *broken* should not (the program, which describes what we know, has no rule to derive *broken*). This in turn allows us to derive *light_on*.

Formalizing these intuitions posed a challenge to the knowledge representation and logic programming communities for years. Eventually, *answer sets* provided a solution that gained acceptance.

Answer sets. To trace the key points of answer sets, we consider two further examples. Let P_1 be the program consisting of the following rules:

high_salary $\leftarrow$ *employed, educated*
educated $\leftarrow$ *high_salary*
employed $\leftarrow$ *motivated*
motivated.

We can regard *motivated* as established as it is the head of a rule that has no preconditions. Consequently, the third rule allows us to derive *employed*. Can we obtain anything else? To get *high_salary* we need to have established *educated* and, similarly, to get *educated* we need to have established *high_salary*. This "vicious cycle" of dependencies cannot be broken as there is no other rule with *high_salary* or *educated* in the head. Hence, neither *high_salary* nor *educated* can be derived given the information in the program. We conclude the set {*motivated*, *employed*} is the only one the program "justifies."

This bottom-up process can be extended to an arbitrary program without the *not* operator. In the general case, however, once negation is allowed the situation gets more complicated. For instance, let P_2 consist of two rules:

open $\leftarrow$ *not closed*
closed $\leftarrow$ *not open.*

In the first example it was clear how to start and how to proceed. It is not so here. The reason is we do not know which atoms cannot be derived, therefore, we cannot verify the conditions for applying any of the rules.

A way out of the problem is to start by *assuming* which atoms will *not* be derived. For instance, let us assume that *closed* will not be derived. Then, the first rule can be used and we can establish *open*. Since *open* is established, the second rule cannot be used and *closed* indeed will not be established, *verifying* our assumption. Thus, the set {*open*} is justified by the program in the following sense. Assuming that atoms not contained in the set cannot be derived, and using program rules (under our intuitive understanding of how they work), we can derive in the bottom-up fashion precisely those atoms that are in the set. Interestingly and importantly, {*open*} is not the only set justified by the program P_2. Another one is {*closed*}: if we assume that

ASP-based Team Building at Gioia Tauro Seaport

The seaport of Gioia Tauro, Reggio Calabria, Italy, is the largest transshipment terminal on the Mediterranean coast. A crucial management task for a port of this size is to build teams of employees to handle incoming ships. This is difficult and time consuming, as one must ensure that teams have appropriate skills, the workload is divided fairly, and legal workload regulations are met. Until recently this task was performed manually, which took several hours per day.

In cooperation with Exeura Srl, a University of Calabria (UNICAL) spin-off, and ICO BLG, an Italian logistics company, Nicola Leone's group at UNICAL has developed an ASP-based system for team building based on the DLV solver.[38] Rules describe the requirements that should be fulfilled regarding: necessary skills of team members; availability of employees; fairness of workload distribution; and distribution of "heavy" or "risky" tasks. Since in practice not all requirements can be satisfied, the system has an implicit conflict handling strategy that gives higher priority to more important criteria.

The system, which has been adopted by ICO BLG for work-force management, can generate shift plans for 130 employees within a few minutes. In addition, the plan quality turned out to be considerably better and overtime was decreased by 20%.

Key factors for the success of ASP in this application were its high expressiveness and the possibility to evolve an executable specification in close interaction with domain experts on site who, although not computer experts, could help getting it right in short time.

open cannot be derived, we can use the second rule to derive *closed*. Having derived *closed*, we have that *open* cannot be derived, confirming the assumption we made.

Our examples suggest the case of programs that contain no rules with *not* in the body is easier. We do not need to make any assumptions about what cannot be derived, as no rule has negated atoms in its body. Instead, we proceed in an iterative fashion collecting atoms that can be established, in each step using atoms derived already to establish new ones. When no more atoms can be derived, the process terminates. The unique set of atoms derived in this way is justified by the program, and we call it the *answer set* of the program.

The concept of an answer set for negation-free programs (also called Horn programs) is a springboard to the general definition. The intuitions we discussed earlier in the context of the program P_2 are crucial. We start with a set M of atoms (in our example, with $\{open\}$) and make an *assumption* that no atom outside M can be derived. Given this assumption, rules that contain a negated atom *not a*, where *a* is in M, become unusable (as the nonderivability of *a* is not *assumed*; in our example, *closed* ← *not open* is unusable). These rules are "blocked" by M and can be disregarded. Therefore, we remove them from the program. In every other rule, if an atom is negated, it must have been assumed nonderivable, otherwise, the rule would have been removed. According to our reading of the rules the corresponding literal can be eliminated from the body without affecting the usability of the rule. Once this is done, we are left with a negation-free program, called the *reduct* of the program with respect to M. If the set of atoms we can derive from that program or, in other words, the answer set of that program, coincides with M, all non-derivability assumptions we made based on M are confirmed, and all atoms in M can be derived. Thus, M is justified by P. We call each such set M an *answer set* of P. The definitions of the reduct and an answer set are due to Gelfond and Lifschitz.[20] Originally, they used the term stable model and introduced the term answer set later for a generalization of the concept to a broader class of programs that feature *strong negation* and *disjunction*, which we will discuss. The new term eventually took over.

The answer set semantics of programs is the foundation of ASP. But equally important is the understanding of how programs encode search problems and their instances.

There is some similarity between rules and propositional logic implications. Indeed, the rule (1) looks like the implication

$$(b_1 \wedge \ldots \wedge b_m \wedge \neg c_1 \wedge \ldots \wedge \neg c_n) \rightarrow a \quad (2)$$

written in a "reversed" fashion. Each answer set of a program is a model of the program viewed as a set of implications (models are truth value assignments to atoms such that each implication evaluates to *true*). However, not all models are answer sets as not all models satisfy the *foundedness* requirement that atoms be derivable in the sense described here.

It should be noted that ASP has solid logic foundations, and is closely linked to nonmonotonic reasoning. In fact, programs under answer set semantics can be seen as a fragment of Reiter's Default Logic and as theories in nonmonotonic modal logics, including Moore's Autoepistemic Logic and nonmonotonic KD45.[31] David Pearce showed that the answer set semantics can be elegantly captured by a nonmonotonic variant of the logic of here and there,[35] a logic located between intuitionistic and classical logic.

Close connection to nonmonotonic logics provides ASP with the power to model default negation and, more generally, to deal with incomplete information. We illustrate that by continuing our *light_on* example. The rule

» key insights

- **Answer set programming is an emerging approach to modeling and solving search and optimization problems. It combines an expressive representation language, a model-based problem specification methodology, and efficient solving tools.**
- **The answer set programming language allows domain and problem-specific knowledge, including incomplete knowledge, defaults, and preferences, to be represented in an intuitive and natural way.**
- **Because of its strong declarative aspect, the language of answer set programming supports rapid prototyping and development of software for solving search and optimization problems, and facilitates modifications and refinements leading to better performance.**

$broken \leftarrow lightning, not\ lightning\_rod$

specifies that the lamp breaks when a lightning strikes, unless a lightning rod was installed. With this rule appended to the program here, we still derive *light_on*, as we cannot derive *broken*. However, things change if we further add the fact *lightning*. As *lightning_rod* cannot be derived, we can establish *broken*, and so *light_on* can no longer be derived. Thus, answer set programs behave *nonmonotonically*—conclusions may have to be retracted when more rules or facts are added to the theory. Further, if we add one more fact *lightning_rod*, the situation changes again; we can no longer derive *broken*, and thus *light_on* will be derived. What this shows is that ASP provides convenient ways for handling exceptions and nested exceptions.

Shorthands and further connectives. A common and important type of rules has its head atom occur negated in the body:

$a \leftarrow B, not\ a.$

If such a rule, let us denote it by r, is added to a program P that has no occurrences of a, then r works as a *constraint*. Namely, a set M of atoms is an answer set of the program $P \cup \{r\}$ if and only if M is an answer set of the program P and does not satisfy (as in propositional logic) the conjunction of literals B. In other words, adding r to P simply eliminates those answer sets of P that satisfy B. As atom a is auxiliary and thus irrelevant (we do not allow it in P), a common way to write a constraint is as a "headless" rule

$\leftarrow B$

which conveys the intuition of a constraint: satisfying B results in a contradiction.

It is also quite common that programs contain pairs of rules

$a \leftarrow B, not\ \bar{a}$
$\bar{a} \leftarrow B, not\ a,$

where neither a nor $\bar{a}$ appear as the head of any other rule in the program, and B is a conjunction of literals. This happens, in particular, when the programmer wants to refer in the program both to an atom a and to its (standard) negation. To represent the latter, the programmer introduces a new atom $\bar{a}$ and includes in the program the two rules here. Intuitively, the role of these rules is to select, in case B is satisfied, exactly one of a and $\bar{a}$; this is precisely what they do under the answer set semantics. Pairs of such rules are often written in a shorthand notation as a single *choice* rule

$\{a\} \leftarrow B.$

Strong negation, denoted with the standard negation symbol $\neg$, allows us to distinguish between having no justification for an atom a, expressed by *not* a, and having one for the negation of a, expressed by $\neg a$. In program rules, $\neg$ can only appear in front of atoms. Gelfond and Lifschitz showed that the definition of answer sets extends to programs of this form almost literally.[21] Every program P with strong negation can be reduced to an ordinary program $\bar{P}$: we simply have to replace each literal $\neg a$ in P by a new atom $\bar{a}$. It can be shown that a consistent set of literals S is a (generalized) answer set of P if and only if the set $\bar{S}$ obtained from S by the same modification is an answer set of $\bar{P}$. Thus, strong negation is only a modeling convenience. However, it makes formulating defaults as in Reiter's Default Logic easier. For example, a rule

$closed_{t+1} \leftarrow closed_t, not \neg closed_{t+1}$

might be interpreted as saying that by default, the valve remains closed at time t+1 if it was closed at time t (that is, unless there are specific reasons for it not to be). Such default rules, which embody the law of inertia, allow for an elegant solution of the frame problem that arises when one reasons about actions and their effects, for instance when modeling and solving planning problems.[1]

Modeling considerations also motivated allowing *disjunctions* in the heads of rules. Disjunctive rules

$a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$

often make representations more intuitive, for example, in a rule like

$open \vee closed \leftarrow valve.$

To eliminate the possibility for a valve to be both, a form of minimality is needed. It is reflected in the answer sets of a disjunctive program.[21] The definition uses the same process as before to "reduce" the program with respect to a candidate atom set M and yields the reduct that is free of (default) negation. However, the reduct may have disjunctions in the heads of its rules and thus, in general, there might be multiple minimal sets of atoms that satisfy all rules (and some are guaranteed to exist). The idea now is to check whether M is one of these minimal sets of the reduct. If this is the case, then M is an answer set. Importantly, unlike strong negation $\neg$, disjunction in the rule heads does increase the problem-solving capacity of programs, as witnessed by results on complexity and expressive power (see the accompanying sidebar "Complexity of ASP").

Complexity of ASP

To decide whether a given program has some answer set is *NP*-complete,[29] thus as complex as the classical propositional satisfiability problem (SAT); in the presence of disjunctive rules, the problem is NP^{NP}-complete[11] (NP^{NP} are the problems decidable in *NP* with an oracle for *NP* problems); roughly speaking, this means *NP*-completeness even if calls to a subroutine for SAT are for free.[9,22] Predicate programs have exponentially higher complexity (intuitively, this is because the reduction by grounding causes an exponential blow up in general). Regarding search problems, ASP can express all *NP*-search problems, that is, those solvable using a nondeterministic Turing machine in polynomial time, in such a way that the answer sets encode the solutions. In fact, each such problem (for example, finding some Hamiltonian cycle) is expressible by a fixed predicate program to which logical facts encoding a given problem instance (for example, a graph) are added. Again, additional constructs like disjunctive rules may increase the expressivity.

Table 1. ASP grounders.

LPARSE	www.tcs.hut.fi/Software/smodels/
DLV	www.dbai.tuwien.ac.at/proj/dlv/ or www.dlvsystem.com/
GRINGO	potassco.sourceforge.net/#gringo/

Table 2. Some ASP systems.

ASSAT	assat.cs.ust.hk/
CLASP 1	potassco.sourceforge.net/#clasp/
CMODELS	www.cs.utexas.edu/users/tag/cmodels/
DLV 2	www.dbai.tuwien.ac.at/proj/dlv/ or www.dlvsystem.com/
GNT	www.tcs.hut.fi/Software/gnt/
SMODELS	www.tcs.hut.fi/Software/smodels/
XASP	xsb.sourceforge.net/, distributed with XSB

[1] + CLASPD, CLINGO, CLINGCON, among others; http://potassco.sourceforge.net/
[2] + DLVHEX, DLVDB, DLT, DLV-COMPLEX, ONTO-DLV, and others.

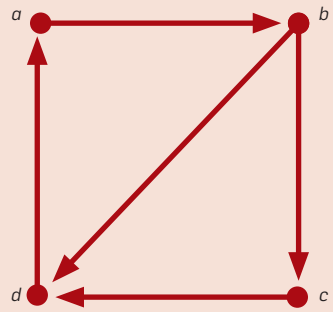
Predicate programs. The propositional case is crucial for the definition of answer set semantics. But it is the predicate version of the formalism that facilitates modeling and makes ASP an effective problem-solving technique. The language has *relation* (or *predicate*) symbols, *constant* symbols and *variables*, as well as the logical connectives we discussed earlier, but no *function* symbols (we will discuss this restriction later). A rule is an expression of the form

$$A \leftarrow B_1, \ldots, B_m, not\, C_1, \ldots, not\, C_n \qquad (3)$$

where A, B_i, and C_i are atomic formulas in the language. Rules are regarded as being implicitly universally quantified. The concepts of the head and body of the rule are defined as before and we interpret a rule (3) similarly as before, too. That is, we understand it as a device that, under some conditions, allows us to derive its head.

More formally, the semantics of a predicate program P is defined in terms of its ground version *grnd(P)*. The program *grnd(P)* consists of all ground instantiations of rules in P with respect to constants that appear in P. In case P contains no constants (a situation that does not occur in practice), one is selected arbitrarily and used to produce *grnd(P)*. The program *grnd(P)* can be regarded as a propositional one over all ground atoms in the language, and the answer sets of P are defined to be those of *grnd(P)*.

A graph for the Hamiltonian cycle problem.



The ASP Paradigm

ASP is an approach to solving search problems. The answer set semantics of programs is the foundation of ASP. But equally important is the understanding of how programs encode search problems and their instances. Niemelä[32] and Marek and Truszczynski[30] first formulated explicitly the basic principles of the ASP approach, Lifschitz[26] was the one to propose the term. In our discussion we rely on a rather intuitive understanding of a *search problem*. Namely, we assume that a search problem Π consists of a set of *instances*, D_Π, with each instance I assigned a finite set $S_\Pi(I)$ of solutions. The set $S_\Pi(I)$ may be empty, that is, problem Π may have no solution for instance I.

To solve a search problem Π, a program P_Π is designed that captures the problem specifications so that when extended with facts $D(I)$, representing an instance I of the problem, the answer sets of $P_\Pi \cup D_\Pi(I)$ describe all solutions of problem Π for the instance I. The upshot of this design is that solving the problem is reduced in a uniform way (the program P_Π is fixed and only the data component changes) to the task of finding answer sets.

We now illustrate how ASP works by analyzing the problem of finding a Hamiltonian cycle in a directed graph. The choice is not arbitrary: this is an important combinatorial problem, arising in several practical situations (for example, as an essential component of the well-known Traveling Salesperson problem). While simple to state, it is still complex enough to allow us to emphasize all key aspects of ASP. In the problem, we are given a directed graph $G = (V,E)$, where V is the set of vertices and E the set of (directed) edges of G. The goal is to find a *Hamiltonian cycle* in G, that is, a set of edges that induce in G a directed cycle going through each vertex exactly once.

We will use two relation symbols to represent graphs: *vtx* and *edge*. Let us consider the graph G shown in the accompanying figure.

We represent the graph G as the set of ground atoms

$$D_{hc}(G) = \{vtx(a), vtx(b), vtx(c), vtx(d)\} \cup \{edge(a, b), edge(b, c), edge(c, d), edge(d, a), edge(b, d)\}.$$

Next, we need to capture the specification of the problem. A key part is the *definition* of a Hamiltonian cycle. According to our description, it must be a *subset* of the edges of the graph. To describe this subset formally, we use a relation symbol *in* and expressions $in(a, b)$ that informally read: the edge (a, b) is selected for a Hamiltonian cycle. To indicate that any edge (X, Y) can be "selected" to be in a Hamiltonian cycle, we use the choice rule:

$$(HC1)\ \{in(X, Y)\} \leftarrow edge(X, Y).$$

Next, we stipulate that no two selected

edges start or end in the same vertex. To this end, we use two constraint rules:

(HC2) $\leftarrow in(V2, V1), in(V3, V1), V2 \neq V3$
(HC3) $\leftarrow in(V1, V2), in(V1, V3), V2 \neq V3.$

We stress the use of the relation symbol $\neq$ here. In the predicate version of ASP, we assume the set of constants includes integers, and the set of relation symbols includes symbols such as $=, \neq, \leq, <, \geq$, and $>$, as well as symbols for (bounded) arithmetic operations such as $+$. To be consistent with standard notation, we use the infix notation and write $X \leq Y$ instead of $\leq (X, Y)$. Similarly, we write $X + Y = Z$ instead of $+(X, Y, Z)$. All these symbols are always interpreted in the standard way.

To be a Hamiltonian cycle, the set of edges $in(x, y)$ must determine a single cycle. To enforce this condition, we need a concept of one vertex being reachable from another. To this end, we use an auxiliary relation symbol *rchble* and the following rules:

(HC4) $rchble(V, V)$
(HC5) $rchble(V1, V3) \leftarrow in(V1, V2), rchble(V2, V3).$

The rules (HC4) and (HC5) define the *transitive closure* of the relation *in*;[a] that is, all pairs of vertices (x, y) such that y can be reached from x by following zero or more edges that are "in." Clearly, the selected edges form a Hamiltonian cycle if and only if every pair of vertices is in the transitive closure. This condition is captured by the following constraint rule:

(HC6) $\leftarrow vtx(V1), vtx(V2),$ *not* $rchble(V1, V2).$

Let P_{hc} be the program consisting of the rules (HC1) - (HC6). One can show that a set of edges H is a Hamiltonian cycle in a graph G if and only if $H = \{(x, y) \mid in(x, y) \in M\}$ for some answer set M of $P_{hc} \cup D_{hc}(G)$.

Finding a Hamiltonian cycle of an arbitrary input G is an *NP*-hard problem, and under a suitable notion complete for all *NP*-search problems. In fact, complexity theory (see the sidebar "Complexity of ASP") tells us that each *NP*-search problem Π is expressible by a program P_Π as noted earlier.

a It is well-known that this is not expressible in first-order logic.

ASP for Repairing Large-Scale Biological Networks

New high-throughput methods have led to a dramatic increase of measurable data in modern molecular biology, and a number of corresponding knowledge repositories are available on the Web. However, both the data and the available biological networks are highly incomplete and error-prone, and inconsistencies are the rule rather than the exception.

In a joint project by Potsdam University, INRIA, and Institut Cochin, led by Torsten Schaub, an approach for repairing biological networks based on ASP has been developed.[19] It builds on a range of available repair actions inspired by biological use cases. Examples are modifications of the role of a node in a biological influence graph (for example, from inhibitor to activator), additions of missing links between nodes, or modifications of experimental data in cases where it is plausible to assume errors in the measurements.

The program rules encode biological knowledge about the repair actions needed and possible in a particular situation. A possible repair is then achieved by minimizing, according to a variety of strategies, the set of applied repair actions. The system uses not more than 20 rules to encode five types of repair actions with different targets.

Processing Answer Set Programs

Current tools for computing with answer set programs support several basic reasoning tasks, which include computing a single answer set (or determining that none exist), computing a given number of answer sets, and computing all of them. Most tools also support deciding whether an atom is true in every (resp. some) answer set, known as *cautious* (resp. *brave*) reasoning. These modalities are important for reasoning applications; for example, when we want to know whether a fact is true in every (resp. some) possible evolution of a system executing a sequence of actions of bounded length.

ASP processing typically works in two stages. First, the predicate program is replaced with an equivalent propositional program by so-called *variable replacement* or *grounding*. Second, that program is processed by a propositional ASP solver. Most implemented ASP processing systems make a clear distinction between the two stages and offer separate tools for each, others integrate them.

Grounding. The naive approach to grounding is to replace a program P with $grnd(P)$; but generally this is not efficient. Consider the rule $p(X) \leftarrow p(X_1), \ldots, p(X_n)$ and assume it needs to be grounded for two constants a and b. Then, the naive grounding will produce 2^{n+1} ground instances, as we can choose for X and each X_i either a or b. However, in this case, the full grounding amounts to just two propositional rules $p(a) \leftarrow p(b)$ and $p(b) \leftarrow p(a)$, as repeated literals in the bodies of rules and tautological rules, where the head atom occurs non-negated in the body, can be eliminated without affecting the answer sets. Intelligent grounding techniques incorporate such equivalences and many further optimizations. They aim to produce, given a predicate program P, a *possibly small* propositional program, not necessarily a subset of $grnd(P)$, that is equivalent to P, that is, has the same answer sets. Current grounders exploit techniques such as partial evaluation, rewriting, and a great deal of database technology to make grounding efficient. We refer to Table 1 for information on the three grounders most broadly used in ASP. Their input formats serve as de facto specifications of three most popular ASP dialects. They are quite close to each other. Nevertheless, the need for standardization is recognized by the ASP community. Extensions to the GRINGO grounder are an important step in this direction, making its input language much closer to that of

the DLV grounder.

Propositional solving. Table 2 provides pointers to several current ASP solvers. All of them more or less directly exploit methods developed in the field of satisfiability solving. Some ASP solver algorithms, often referred to as *native* (to ASP), follow the general backtracking search pattern of SAT solvers but append SAT-based propagation techniques with ones implied by an additional *foundedness* condition that models must satisfy to be answer sets.[25] It means every atom that is true in a model must be *derived* (in a certain precise sense) by a rule in the program. The search backtracks when either a contradiction is derived, or a complete and consistent assignment is found but some atoms that are true lack a derivation (are not founded). In each case, the need to backtrack indicates that some decisions made in the search earlier are incompatible with any answer set of the program and must be changed. This group of algorithms embodies a perspective on answer sets best captured by a catchphrase

(propositional) ASP = SAT + foundedness.

The answer set search outlined earlier can be improved by sophisticated search heuristics and techniques like backjumping and clause learning developed in the field of SAT solvers. The current ASP solvers take full advantage of these techniques. The native ASP solver CLASP, dressed as a SAT solver, won two tracks of the 2009 SAT solver competition.

Other successful ASP solver algorithms are based on reductions of answer set solving to satisfiability testing. They modify the formula corresponding to a program so that its models are exactly (or up to trivial projections) the answer sets of the program. One approach is to produce the so-called program *completion*. It reflects the idea that the program provides *all* conditions under which atoms are true; that is, it is a definition of the atoms in its rule heads. Accordingly, the completion is the formula containing for each atom a an equivalence saying that a holds if and only if the disjunction of the bodies of all rules with a in the head holds. The completion captures some aspects of the foundedness condition, but not all. To capture it entirely, the completion must be extended by *loop formulas*, that exclude self-supporting derivations.[28] Loosely speaking, this approach could be cast as

ASP = completion + loop formulas.

Once the completion and loop formulas are built, an off-the-shelf SAT solver is used to find models of the resulting theory and so, answer sets of the original program. In the worst case, there can be exponentially many loop formulas, which complexity theory tells is somehow unavoidable. Therefore, some ASP solvers based on this idea, for example, ASSAT, add loop formulas incrementally and test whether models are already answer sets, while others, such as CMODELS2, similarly employ special techniques to select promising loop formulas to add and to "forget" them later.

Other reductions of ASP computation to SAT solving use auxiliary atoms for level rankings to represent founded derivation by keeping track of successive rule applications. Following this direction, translations of ASP to SAT modulo difference logic have been proposed that exploit fast solvers for theories in that formalism.[33]

ASP Extensions

Motivated by the needs of applications, several extensions of the basic ASP paradigm have been proposed.

Constraints and aggregates. Constraints on sets of atoms are particularly common. For instance, one often needs to say that exactly one out of a given set of atoms is true. In the well-known n-queens problem, we must place n queens on the $n \times n$ chessboard so that no two queens attack each other. Here one of the constraints is that exactly one queen is in each row. Even though this can be naturally encoded in the basic ASP language, the grounding will result in a large number of rules. ASP input languages thus provide constructs for constraints on sets of atoms that ASP solvers handle suitably. Basically, there are two approaches.

The first approach, which originated with LPARSE, uses the concept of a *cardinality atom*. In the propositional case, it has the form

$$l\,\{a_1, \ldots, a_n\}\,k$$

and reads: at least l and at most k atoms in the set $\{a_1, \ldots, a_n\}$ are true (if l or k are missing, it implies no restriction from the respective side). In the predicate language, one can be even more concise and write expressions such as

$$L\,\{a(X) : p(X, Y)\}\,K,$$

where L, K, X, and Y are variables. The expression captures a condition that given a value for Y, for at least L and at most K of the values of X such that $p(X, Y)$ holds, $a(X)$ is true. To ensure the grounding process is well defined, syntactic conditions on variables are used.

Let us denote by $q(X, Y)$ that some queen is in row X and column Y. We can state the uniqueness constraint on

ANTON—An ASP-based Music Composition System

ANTON,[4] developed at University of Bath in cooperation with University of Glamorgan, is an automatic system for the composition of Renaissance-style music. It represents musical knowledge in the form of about 500 ASP rules. The rules describe the progression of a melody, both at the local level (the choice of the next note) and at the global level (the overall structure), the harmony that arises from the relationship between the melodic line and the supporting instruments, and also the rhythm, such as the intervals between notes, of a piece.

Given some initial information, for example, fixed notes or number of parts, the program generates answer sets representing musical pieces that satisfy the composition rules. With minor modifications, the system can also be used to detect violations of composition rules in given pieces of music.

queens in each row concisely by the following two constraint rules:

$$\leftarrow\ 2\ \{q(X, Y) : col(Y)\}, row(X)$$
$$\leftarrow\ \{q(X, Y) : col(Y)\}\ 0, row(X).$$

The first rule states that for no row X there are distinct Y and Y' such that $q(X, Y)$ and $q(X, Y')$ are true (no row contains two or more queens). The second rule states that for no row X, it holds that all atoms $q(X, Y)$ are false (there is no row without queens). There is a more general version of cardinality constraints, *weight* constraints, where each atom is associated with a weight and the bounds constrain the sum of the weights of atoms that have some property.

The second approach to modeling constraints on sets of atoms follows the idea of aggregates familiar from SQL in databases.[16] Those implemented in ASP languages include *count*, *sum*, *maximum*, and *minimum* and follow closely the database syntax. In the DLV input language, the unique-queen constraint is expressible by

$$\leftarrow\ 1\ !=\ \#count\{Y : q(X, Y)\}, row(X).$$

The input language of GRINGO also recognizes aggregates such as *count* and *sum* but specifies bounds as in cardinality constraints; this points to the need for standardization of ASP input languages.

Preferences. A basic assumption of the ASP paradigm is that problems are modeled in a way such that answer sets represent their solutions. However, it is impossible to further distinguish between better and poorer solutions. One way to address this problem is to introduce preferences. Simple forms of preferences can be expressed using *#minimize* and *#maximize* statements that are supported by several of the existing ASP solvers. They allow us to associate weights with specific literals. The generated answer sets then are those for which the sum of the weights of satisfied literals is minimal/maximal. The DLV system provides so-called "weak constraints," which carry a weight of importance; they should be satisfied if possible, but their violation does not "kill" answer sets. The answer sets of a program P plus a set W of weak constraints are those answer sets of P that minimize the sum of the weights of violated weak constraints. Other, non-numerical approaches use an external partial preference order on rules or special syntactic constructs in the rules; for example, Brewka et al.[6] In each case the available preference information induces a corresponding ordering on answer sets, and the best ones are chosen.

Modularity and external data access. Modularity is an important notion in software development. In the context of ASP it is only beginning to receive the attention it deserves but already several key concepts and ideas have been developed.[10,23] Modularization is a way to structure and ease the program development process. Modular ASP programs consist of modules that are combined through suitable interfaces. This way parts of a program can be developed and verified independently, and they can be more easily reused. A related issue is to integrate external sources into ASP programs. In a rule one would often like to access a database, an ontology or some other source of information. To serve this, HEXprograms[13] provide a universal interface for arbitrary sources of external computation through the notion of external atom, which is akin to a remote procedure call but facilitates proper recursion.

Applications

The ASP paradigm is rather new but it has already led to many successful applications. We briefly discuss a few examples in different categories. Further examples can be found in the team-building sidebar noted earlier as well as the ones entitled "ASP for Repairing Large-Scale Biological Networks" and "ANTON—An ASP-based Music Composition System."

Applications in science and humanities. An illustrative example is phylogenetic systematics—the study of evolutionary relations between species based on their shared traits.[15] These relations can form a tree (called a "phylogeny") where leaves represent the species, internal vertices their ancestors, and edges the genetic relationships between them. The computational task is to construct phylogenies, and researchers demonstrated the applicability and effectiveness of ASP-based methods for these tasks by analysis of natural languages and parasite-host systems species of oak trees.

Industrial applications. An early, almost prototypical industrial application for ASP is product configuration.[39] The general idea is to have rules in a program that generate the space of all combinations of product components. Constraint rules then filter out configurations that are impossible, either due to some given, fixed restrictions on how components can be combined, or due to a violation of specific user requirements. Another early application is a decision support system for the space shuttle.[34] During normal shuttle operations, astronauts follow pre-scripted plans. However, in case of failure different courses of action are needed to ensure safety of the crew and completion of the mission. As exponentially many failures are possible, pre-planning for all exceptional circumstances is unfeasible, and decision support is needed. Based on failure information, the ASP system suggests a course of action.

Data management. INFOMIX[b] is a project on advanced information integration. The main task is to provide a uniform interface to pre-existing data sources, where an information integration system frees the user from finding and accessing relevant data sources, and from cleaning and combining data in them. Here, in particular, proper handling of incomplete and inconsistent data is challenging. The INFOMIX prototype showed that ASP provides effective technology to deal with advanced information integration tasks. ASP also proved to be a valuable host for realizing query engines in the context of the Web. In fact, one of the first SPARQL reasoning engines for querying RDF data sources has been realized via an ASP encoding.[37]

Artificial intelligence. Given the fact that ASP has roots in knowledge representation and nonmonotonic reasoning, its usage for problem solving in artificial intelligence (AI) has been investigated early on. Classic AI problems including planning, diagnosis, and agent decision making have been reduced to ASP, resulting

b www.mat.unical.it/infomix/

in effective realizations (several are available, for example, as DLV frontends). As it turned out, thanks to its features—high expressiveness, nondeterminism via multiple answer sets, and high declarativity—ASP is a valuable host language for domain-specific AI formalisms, allowing for quick experimental prototyping. A recent example of this is repair of Web-service workflows,[18] where these features were fruitfully exploited.

Relation to Other Formalisms

ASP is just one of many ways to solve search problems by means of logic reasoning procedures. We briefly comment on three formalisms for declarative problem solving that are both related and relevant to ASP.

ASP and SAT solving. The key idea of ASP—to encode the solutions of a search problem in the models of a logical theory for declarative problem solving—had been exploited before. In a landmark paper, Kautz and Selman[24] showed that encoding a planning problem as a theory in propositional logic, with plans represented by models, and using SAT solvers to find models and so plans, could outperform specialized planners. Applications of similar nature led to a boom in SAT solver technology.

While both SAT solving and core ASP apply in principle to the same problems, there are differences. First, ASP supports variables that range over finite domains and enable uniform and compact representation of problems independently of data. In the Hamiltonian cycle example, we have a *single*, *fixed* program that works uniformly with *all* input graphs. ASP grounders produce instances for ASP solvers based on the program and the input graph. Having problem specifications separate from data facilitates debugging and testing, supports optimization and developing reusable problem modules, all topics currently under research. There is no such separation of problem specification and data in SAT, where the two are hardwired into programs that generate satisfiability instances to be solved. This makes development of software engineering techniques for SAT difficult. Second, any problem that can be modeled in SAT can be modeled equally well in ASP. But there are problems—typically involving concepts defined inductively such as reachability in graphs—that are easy to cast in ASP, but representing them appropriately for SAT solving results in larger instances that slow down solving. In a similar vein, the language of ASP offers constructs such as "minimized" disjunction, aggregates and priorities that are useful in practical applications, are easy to use, and are supported by most current ASP solvers. These constructs require specialized ad hoc treatment when modeling for SAT solving. For some of them concise representations are not even possible.

Thanks to its features—high expressiveness, nondeterminism via multiple answer sets, and high declarativity—ASP is a valuable host language for domain-specific AI formalisms, allowing for quick experimental prototyping.

ASP and Prolog. Prolog is the most widely known logic programming language. For some time, however, the interest in Prolog has been declining, in part because expectations of ambitious endeavors like the Fifth Generation Project could not be met. Is ASP, which is sometimes called *Answer Set Prolog*, a better Prolog? The two are similar in syntax and there are semantic connections, too. For a large class of programs, if Prolog returns "yes" (respectively "no") to a ground query, then the query belongs (respectively, does not belong) to the unique answer set of the program. But in spite of these similarities, ASP and Prolog are actually quite different. Prolog was designed as a general purpose, Turing-complete programming language. It uses function symbols for nested terms to build potentially infinite data structures, and recursion for unbounded computation; solutions are computed by query answering, which amounts to *proof search*. In contrast, ASP was not conceived for such generality and works over a finite domain of "flat" data (though work on function symbols in ASP is under way); solutions are encoded in answer sets; that is, in *models*, and thus model-finding, not proof-finding, methods matter.

To be an effective Prolog programmer one needs to understand how to use terms as data structures, not quite intuitive, and not part of any standard CS curriculum, and to understand Prolog's evaluation strategy, SLD resolution with unification, which is arguably quite difficult to master with no adequate logic background. To

complicate matters more, the order of rules in a Prolog program and of subgoals (literals) in rule bodies matters. Changing it may turn a working program useless. These features give a programmer control over the execution of search and make Prolog a programming language, a formalism in which one can implement algorithms. In this sense, Prolog misses *true declarativity*. ASP, on the other hand, offers ways to model specifications yet does not allow the programmer to control the search. Consequently, while less expressive, ASP is "more declarative:" it is intuitive, requires less background in logic, and its semantics is robust to changes in the order of literals in rules and rules in programs. Still, to solve practical application problems in ASP efficiently some experience is required. Typically, there are alternative ways to model a problem as an answer set program, and the resulting programs may perform quite differently. One of the more obvious and in the same time more important considerations for designing efficient answer set programs is that the size of the ground program be possibly small.

ASP and constraint programming. Constraint programming is concerned with modeling and solving problems, where solutions are assignments of values from finite domains to *decision* variables. These assignments are subject to constraints given in the problem statement.

For instance, we can specify the *n*-queens problem as follows: assign to each of n decision variables $x_1, \ldots, x_n$, a value from $1, \ldots, n$ so that $x_i \neq x_j$, for $i \neq j$, and $|x_i - x_j| \neq |i - j|$. To solve a problem like this in constraint programming, one describes it in some high-level *modeling* language, such as ESSENCE or ZINC, and then maps the description into a set of constraints in some low-level format or, in other words, into a constraint satisfaction problem (CSP), which is then solved. The similarities with ASP—modeling in a high-level language and compiling to a low-level representation—are evident. But there are differences.

High-level languages including those mentioned here closely follow mathematical notation and, in particular, support using sets, relations, functions and partitions as possible values of decision variables; modeling requires some mathematical sophistication. Mapping the high-level specification of a problem into constraints that will lend themselves well to processing also requires certain mathematical background, and expertise in constraint modeling and solving. On the other hand, the language of ASP and its extensions were developed with knowledge representation applications in mind and their constructs were designed to capture patterns of natural language statements, definitions, and default negation. The language is simple and intuitive to use. In addition, once a problem is modeled in ASP all subsequent steps are performed automatically. A grounder compiles a program into its propositional form and a solver computes solutions. There are also differences at the solving stage. For constraint programming this step consists of solving a CSP over an arbitrary but finite value domain. For ASP, all domains are binary (the variables are propositional atoms). This restriction opens a way to highly efficient implementations, as witnessed by the recent impressive advances in SAT solving technology.

Ongoing Developments

ASP processing tools are under continuous development and already achieved levels that make them effective in large-scale practical applications. Efforts to increase efficiency by new grounding technology and solving methods, but also non-ground evaluation are under way. To a large degree the advances are the result of a communitywide effort to build benchmarks, collect hard test problems and instances, and organize regular ASP system competitions.

However, the situation is quite different as concerns basic *software development support* in ASP. Although the first integrated development environment ASPIDE[c] was recently announced, much remains to be done. One of the areas in need of progress is program debugging. Even if developing answer set programs benefits from the declarative nature of ASP, discovering errors is difficult. There is some research in this direction already,[5,7] but the ideas proposed need to be explored further. Methodologies for development and optimization are also important issues. Much progress was made in understanding the theory behind modularity of answer set programs. We discussed some of that research earlier. Here we mention research on *strong equivalence*[27] or, to put it informally, equivalence for replacement within larger systems, and further notions of equivalence.[40] Elegant technical results are now available, but their impact on practical developments remains open.

Function symbols often make modeling easier and the resulting encodings more readable and concise. Thus, not allowing them in ASP (except in built-ins for arithmetic) was perceived as a limitation. But allowing uninterpreted function symbols renders most of the ASP program processing techniques useless, as ground programs typically become infinite. A middle ground can be found, though. It requires imposing restrictions on how function symbols can occur in programs. Some globally constrain atom dependency in the grounded program,[3,8] while others locally constrain the rule syntax.[14] The LPARSE grounder was the first to offer (albeit limited) support of function symbols, while GRINGO and the DLV system (latest release) include some of the more recent advances. Recent research indicates that ASP can provide a full first-order language for nonmonotonic reasoning, with the notion of an answer set extended to this setting.[17,36] Computational support and further research will be required, however, to make this available for practical applications.

Integration of SAT solving with constraint solving techniques known as Satisfiability Modulo Theories has proved successful for SAT. The ASP community has recently taken up this idea, with CLINGCON (see Table 2) being a very promising system combining ASP with specialized constraint solvers.

Quantitative methods turned out to be extremely effective in knowledge representation applications in which uncertainty cannot be avoided. ASP as

c www.mat.unical.it/~ricca/aspide/

it exists now is not designed for such applications. This is a drawback and so there are already research efforts to enhance ASP with means to combine probabilities and utilities with qualitative representations of uncertainty.[2] This research direction has not yet matured, though, and it is too early to say how successful such integration will turn out to be.

Conclusion

The aim of this article was to provide the reader with a basic understanding of the main motivation, the most important concepts, and the relevant techniques underlying ASP, a rather new yet highly promising declarative problem-solving paradigm.

We covered answer set semantics, both for propositional and predicate programs, discussed the ASP paradigm, and related it to some other problem-solving approaches. Moreover, we presented algorithms and solvers, several extensions of the basic approach, and some illustrative applications. This article should not be viewed as a complete overview of the field. It is meant as an appetizer. For a more complete picture we recommend Eiter et al.[12] or Baral.[1]

Acknowledgments

The authors are grateful to the reviewers for comments that helped improve the presentation of the material. Brewka's work was supported by the DFG grant Br1817/3; Eiter's work was supported by the Austrian Science Fund (FWF) grants P20840 and P20841, Vienna Science and Technology Fund (WWTF) ICT08-020, and the European Commission grant ICT FP7 231875. Truszczyński's work was supported by NSF grant IIS-0913459.

References

1. Baral, C. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.
2. Baral, C., Gelfond, M. and Rushton, J.N. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming 9*, 1 (2009), 57–144.
3. Baselice, S., Bonatti, P.A. and Criscuolo, G. On finitely recursive programs. *Theory and Practice of Logic Programming 9*, 2 (2009), 213–238.
4. Boenn, G., Brain, M., Vos, M.D. and Fitch, J. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming 11*, 2-3 (2011), 397–427.
5. Brain, M. and Vos, M.D. Debugging logic programs under the answer set semantics. In *Proc. 3rd International Workshop on Answer Set Programming, CEUR Workshop Proceedings 142*, 2005. M. De Vos and A. Provetti, Eds.
6. Brewka, G., Niemelä, I. and Truszczyński, M. Answer set optimization. In *Proc. 18th International Joint Conference on Artificial Intelligence*. G. Gottlob and T.Walsh, Eds. Morgan Kaufmann, 2003, 867–872.
7. Brummayer, R. and Järvisalo, M. Testing and debugging techniques for answer set solver development. *Theory and Practice of Logic Programming 10*, 4-6 (2010) 741–758.
8. Calimeri, F., Cozza, S., Ianni, G. and Leone, N. Computable functions in ASP: Theory and implementation. In *Proc. 24th International Conference on Logic Programming*, LNCS 5366. M. Garcia de La Banda and E. Pontelli, Eds. Springer, 2008, 407–424.
9. Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and expressive power of logic programming. *ACM Computing Surveys 33*, 3 (2001), 374–425.
10. Dao-Tran, M., Eiter, T., Fink, M. and Krennwallner, T. Modular nonmonotonic logic programming revisited. In *Proc. 25th International Conference on Logic Programming*, LNCS 5649. P. M. Hill and D.S. Warren, Eds. Springer, 2009, 145–159.
11. Eiter, T. and Gottlob, G. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence 15*, 3/4 (1995), 289–323.
12. Eiter, T., Ianni, G., and Krennwallner, T. Answer set programming: A primer. *Reasoning Web*, LNCS 5689. S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R. A. Schmidt, Eds. Springer, 2009, 40–110.
13. Eiter, T., Ianni, G., Schindlauer, R. and Tompits, H. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proc. 19th International Joint Conference on Artificial Intelligence*. L. P. Kaelbling and A. Saffiotti, Eds. 2005, 90–96.
14. Eiter, T. and Simkus, M. FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Trans. Computational Logic 11*, 2 (2010).
15. Erdem, E. Applications of answer set programming in phylogenetic systematics. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday,* LNCS 6565. M. Balduccini and T. C. Son, Eds. Springer, 2011, 415–431.
16. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T. and Ielpa, G. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming 8*, 5-6 (2008), 545–580.
17. Ferraris, P., Lee, J. and Lifschitz, V. Stable models and circumscription. *Artificial Intelligence 175*, 1 (2011), 236–263.
18. Friedrich, G., Fugini, M., Mussi, E., Pernici, B. and Tagni, G. Exception handling for repair in service-based processes. *IEEE Trans. on Software Engineering 36*, 2 (2010) 198–215.
19. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S. and Veber, P. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proc. 12th International Conference on Principles of Knowledge Representation and Reasoning*. F. Lin, U. Sattler, and M. Truszczynski, Eds., 2010, 497–507.
20. Gelfond, M. and Lifschitz, V. The stable model semantics for logic programming. *Logic Programming: The 5th International Conference and Symposium.* R.A. Kowalski and K. Bowen, Eds. MIT Press, Cambridge, MA, 1988, 1070–1080,
21. Gelfond M. and Lifschitz, V. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9* (1991), 365–385.
22. Greco, S., Molinaro, C., Trubitsyna, I. and Zumpano, E. NP datalog: A logic language for expressing search and optimization problems. *Theory and Practice of Logic Programming 10*, 2 (2010), 125–166.
23. Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research 35* (2009), 813–857.
24. Kautz, H.A. and Selman, B. Planning as satisfiability. In *Proc. 10th European Conference on Artificial Intelligence*. B. Neumann, Ed. 1992, 359–363.
25. Leone, N., Rullo, P. and Scarcello, F. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation 135*, 2 (June 1997), 69–112.
26. Lifschitz, V. Answer set programming and plan generation. *Artificial Intelligence 138* (2002), 39–54.
27. Lifschitz, V., Pearce, D. and Valverde, A. Strongly equivalent logic programs. *ACM Trans. Computational Logic 2*, 4 (2001), 526–541.
28. Lin, F. and Zhao, Y. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc˙18th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence*, 2002, 112–117.
29. Marek , V.W. and Truszczyński, M. Autoepistemic logic. *J. ACM 38*, 3 (1991) 588–619.
30. Marek , V.W. and Truszczyński, M. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm—A 25-Year Perspective.* K. Apt, K.V. V.W. Marek, M.W. Truszczyński and D.S. Warren, Eds. Springer, 1999, 375–398.
31. Marek , V.W. and Truszczyński, M. *Nonmonotonic Logics – Context-Dependent Reasoning.* Springer, 1993.
32. Niemelä, I. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 3–4 (1999), 241–273.
33. Niemelä, I. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence 53*, 1 (2008), 313–329.
34. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. A Prolog decision support system for the space shuttle. In *Proc. 1st International Workshop on Answer Set Programming*. A. Provetti and T. C. Son, Eds, 2001.
35. Pearce, D. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence 47*, 1-2 (2006), 3–41.
36. Pearce, D. and Valverde, A. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proc. 9th European Conference on Logics in Artificial Intelligence*, LNCS 3229. Springer, 2004, 147–160.
37. Polleres, A. From SPARQL to rules (and back). In *Proc. 16th International Conference on World Wide Web.* C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider, and P.J. Shenoy, Eds. ACM, 2007, 787–796.
38. Ricca, F., Grasso, G., Alviano, M., Manna, M. Lio, V. Liritano, S. and Leone, N. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, 2011; doi:10.1017/S147106841100007X.
39. Soininen, T. and Niemelä, I. Developing a declarative rule language for applications in product configuration. In *Proc. 1st International Workshop on Practical Aspects of Declarative Languages*, LNCS 1551. G. Gupta, Ed. Springer, 1999, 305–319.
40. Woltran, S. A common view on strong, uniform, and other notions of equivalence in answer-set programming. *Theory and Practice of Logic Programming 8*, 2 (2008), 217–234.

Gerhard Brewka (brewka@informatik.uni-leipzig.de) is a professor of computer science at University of Leipzig's Informatics Institute, Leipzig, Germany.

Thomas Eiter (eiter@kr.tuwien.ac.at) is a professor of computer science at Vienna Univ. of Technology's Institute of Information Systems, Vienna, Austria.

Mirosław Truszczyński (mirek@cs.uky.edu) is a professor at University of Kentucky's Department of Computer Science, Lexington, KY.