# ASP Essentials: Modelling and Efficient Solving

**Giuseppe Mazzotta**
University of Calabria, Rende, Italy

**Francesco Ricca**
University of Calabria, Rende, Italy

## Abstract

Answer Set Programming (ASP) is a logic-based Knowledge Representation and Reasoning (KRR) paradigm that facilitates rapid prototyping of solutions for complex problems. It is particularly effective for tackling Deep Reasoning tasks involving exponentially large search spaces, such as combinatorial search and optimization. While getting started with ASP is relatively easy, mastering its advanced constructs and scaling solutions to real-world problem sizes can be challenging. This paper provides an introduction to ASP, guiding the reader from the fundamentals of the language to the application of programming methodologies and the computation of answer sets. Beyond the core framework, the paper also examines selected extensions of ASP that enable the modeling of complex problems, as well as compilation techniques designed to enhance solving efficiency. Furthermore, it mentions some recent tools that combine ASP with LLMs.

**2012 ACM Subject Classification** Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Logic programming and answer set programming; Computing methodologies → Artificial intelligence

**Keywords and phrases** Answer Set Programming, ASP with Quantifiers, Grounding Bottleneck, Compilation-based ASP solving, Neurosymbolic AI, LLMs

**Digital Object Identifier** 10.4230/OASIcs.RW.2024/2025.8

**Category** Invited Paper

**Funding** This work was supported by the Italian Ministry of Industrial Development (MISE) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005; and by the Italian Ministry of Research (MUR) under projects: PNRR FAIR - Spoke 9 - WP 9.1 CUP H23C22000860006, and Tech4You CUP H23C22000370006.

## 1 Introduction

Answer Set Programming (ASP) [13] is a well-established declarative paradigm in the field of Artificial Intelligence, specifically designed for knowledge representation and reasoning. Rooted in the stable model semantics [40], ASP provides a high-level formalism that enables the specification of complex problems in terms of logical rules. In this framework, the problem-solving process is divided into two distinct phases: users describe the problem declaratively by encoding it as a set of logical rules, and dedicated computational systems are then employed to determine its stable models (or answer sets) [40]. Each answer set corresponds to a valid solution of the original problem, thus creating a clear separation between problem specification and solution computation.

The declarative nature of ASP allows both researchers and practitioners to abstract away from procedural implementation details. ASP allows problem statements to be expressed in an intuitive and concise manner, while leaving the intricacies of search and optimization to highly optimized solvers such as CLINGO [37], WASP [2] and DLV [1]. This separation of concerns ensures that the computational effort required to explore large and complex solution spaces is managed efficiently by the solver, making ASP particularly suitable for complex reasoning, including problems that are more complex than NP ones.


Over the last two decades, ASP has demonstrated its effectiveness through a wide range of applications, spanning both academic research and industrial contexts [28, 35]. Notably, it has been employed in areas such as planning [64], scheduling [22, 16], natural language processing [20, 55, 63, 69], among many others [25, 41, 4, 30]. The versatility of ASP as a modeling tool, coupled with the continuous advancements in solver technology, has cemented its role as a powerful formalism within the broader landscape of declarative AI.

This paper aims at offering a gentle introduction to ASP to guide the reader through the fundamental aspects of the formalism, beginning with the core principles of the language and progressing to the systematic application of programming methodologies for modeling and problem-solving. Particular emphasis is placed on the computation of answer sets, which constitute the central reasoning mechanism of ASP and serve as the basis for computing solutions of complex computational tasks.

In addition to presenting the foundations, the paper broadens its scope to examine an extension of ASP that enhance its expressive power and enable the natural modeling of complex problems, namely Answer Set Programming with Quantifiers (ASP(Q)) [5].

ASP(Q) has been recently proposed as a natural declarative extension of ASP, aimed at modeling problems across the entire Polynomial Hierarchy (PH). While standard ASP is known to capture problems within the class $\Sigma_2^P$ [21], which already encompasses a broad range of practically relevant applications, many decision problems of theoretical and applied importance reside in higher levels of the hierarchy [62, 65]. Over the years, several language extensions have been developed to expand the expressive power of ASP [11, 36]. In this context, ASP(Q) provides a more natural and declarative approach to representing such problems, ensuring a uniform treatment of complexity classes beyond $\Sigma_2^P$. More recently, the works of [3] and [32] have introduced efficient systems for the evaluation of ASP(Q) specifications, supplying empirical evidence of the practical feasibility and potential impact of this extension.

The paper also investigates computational aspects of ASP, overviewing the compilation techniques that have been recently proposed to enhance the efficiency of ASP solving [19, 53, 23, 24]. Traditional systems such as CLINGO [37] and DLV [1] follow the *Ground&Solve* approach [47], where a program is first grounded into a propositional form and then solved via a Conflict Driven Clause Learning-like algorithm [51]. While highly successful, these systems face the well-known grounding bottleneck, where variable elimination alone may exhaust computational resources in practice [15, 57]. Compilation-based methods address this issue by bypassing grounding in certain cases, initially focusing on constraints [19, 53]. More recently, the PROASP system [23] has shown that compilation can also handle rules generating answer sets and not only constraints, enabling an unrestricted integration of grounding and compilation for more effective and efficient evaluation [24].

Finally, the paper brings the attention of the reader to the promising combination of ASP with Large Language Models (LLMs). It mentions some recent efforts aimed at increasing the usability and accessibility of ASP, particularly through the development of tools that automatically generate ASP programs from natural language descriptions [12, 18]. These tools aim at bridging the gap between high-level human problem statements and formal ASP encodings, thus lowering the entry barrier for non-expert users and expanding the range of potential applications. In this context, the development of neurosymbolic AI systems leveraging ASP as a reasoning component is also noteworthy. Yang et al.[69] demonstrated that LLMs can serve as few-shot semantic parsers, producing ASP logical forms without retraining. Other works applied prompt engineering to encode and solve logic puzzles in ASP[42], or proposed hybrid systems such as LLM2LAS [46], which integrates LLMs with ILASP [49] to acquire commonsense knowledge from narrative QA.

Although the combination of ASP and LLMs is still a relatively new and emerging area of research, the results reported in the literature are promising. They not only reveal the intrinsic difficulties faced by current NLP tools (including LLMs) in generating ASP programs, but also demonstrate the feasibility of developing dedicated systems that combine ASP and LLMs.

### Paper structure

The paper is structured as follows:

- Section 2 introduces the ASP syntax and semantics.
- Section 3 presents the Guess&Check methodology [27], the main programming approach for modeling problems in ASP, and illustrates it through classical examples of well-known problems in computer science.
- Section 4 provides an overview of standard ASP solving techniques adopted by state-of-the-art systems, as well as novel methodologies for efficient ASP solving.
- Section 5 introduces the ASP(Q) formalism and demonstrates its expressiveness through two modeling examples of interesting problem in Graph Theory.
- Section 6 discusses some recent combinations of ASP with NLP tools, which is an emerging and relevant research topic.
- Section 7 summarizes the main contributions and concludes the paper.

## 2 Answer Set Programming

In this section we introduce the ASP language and its semantics. Then we shift the attention to practical modeling examples together with possible integration with Large Language Models for enabling the automatic composition of ASP programs.

### The Language

In ASP *variables* are alphanumeric strings starting with uppercase letter; whereas *constants* are either numbers or alphanumeric string starting with lowercase letter. A *term* is either a variable or a constant. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate of arity $n$, with $n \geq 0$, and $t_1, \ldots, t_n$ is a sequence of terms. A *literal* is either an atom $a$ or its negation *not* $a$, where *not* represents *negation as failure*. A literal of the form $a$ is said to be *positive*; otherwise it is *negative*. A *rule* is an expression of the form $h$ :- $l_1, \ldots, l_n$, where $h$ is a standard atom, referred to as *head*, which can also be omitted, and $l_1, \ldots, l_n$ is a conjunction of literals with $n \geq 0$, referred to as *body*. A rule with an empty head is said to be a *strong constraint*; whereas a rule with an empty body is said to be a *fact*. For a rule $r$ we denote by $H(r)$ the head of $r$; and by $B(r)$ the set of literal appearing in the body of $r$. A *weak constraint* is an expression of the form $:\sim l_1, \ldots, l_n$ $[W@L, \vec{T}]$, where $l_1, \ldots, l_n$, with $n > 0$, is conjunction of literals referred to as body; $W$ and $L$ are terms referred to as *weight* and *level*; and $\vec{T}$ is a possibly empty sequence of terms. An ASP *program* is finite set of rules possibly containing also weak constraints. An ASP program without weak constraints is said to be *plain*. In what follows we are going to use also *choice rules* which allows a compact and intuitive modeling. A choice rule is an expression of the form $\{h_1; \ldots; h_m\}$ :- $l_1, \ldots, l_n$ where $h_1, \ldots, h_m$ are atoms and $l_1, \ldots, l_n$ are literals. It is important to point out that choice rules are just shorthands for a set of normal rules of the form $h_i$ :- $l_1, \ldots, l_n, not\ nh_i$ $nh_i$ :- $l_1, \ldots, l_n, not\ h_i$, where for each $i \in \{1, \ldots, m\}$, $nh_i$ is a fresh atom not appearing anywhere else.

### Stable Model Semantics

Given a program $P$, the *Herbrand Universe*, denoted by $U_P$, is the set of constants appearing in $P$; whereas the *Herbrand Base*, denoted by $B_P$, is the set of ground standard atoms that can be obtained from predicate in $P$ and constants in $U_P$. Given a rule $r \in P$ we denote by $ground(r)$ the set of ground instantiation of $r$ which can be obtained by mapping variables in $r$ to constants in $U_P$. Similarly, $ground(P)$ is the the union of $ground(r)$ for each $r \in P$. An *interpretation* $I \subseteq B_P$ is a set of ground atoms. Let $I$ be an interpretation then a positive (resp. negative) ground literal $l = a$ (resp. $l = not\ a$) is true w.r.t. $I$, denoted by $I \models l$, if $a \in I$ (resp. $a \notin I$); otherwise $l$ is false w.r.t. $I$, denoted by $I \not\models l$. A conjunction of literals $l_1, \ldots, l_n$ is true w.r.t. $I$, denoted by $I \models l_1, \ldots, l_n$, if $l_i$ is true w.r.t. $I$ for each $i \in \{1, \ldots, n\}$. A ground rule $r$ is *satisfied* w.r.t. $I$ if the head of $r$ is true w.r.t. $I$ whenever the body of $r$ is true w.r.t. $I$. $I$ is a *model* of $P$ if each rule $r \in ground(P)$ is satisfied w.r.t. $I$. Let $I$ be a model of $P$ then $P^I$ denotes the Gelfond-Lifschitz reduct [40] obtained from rules in $P$ by $(i)$ removing rules having in the body at least one negative literal false w.r.t. $I$; and $(ii)$ removing negative literals from the body of remaining rules. Then, $I$ is an *stable model* (i.e., *answer set*) of $P$ if $I$ is $\subset$-minimal model of $P^I$. For a program $P$ we denote by $AS(P)$ the set of answer set of $P$. Finally $P$ is said to be *coherent* if it admits at least one answer set (i.e., $AS(P) \neq \emptyset$); otherwise $P$ is *incoherent*.

▶ **Example 1.** Let $P$ be the following ASP program:

```
f(1).

a(X) :- f(X), not b(X).
b(X) :- f(X), not a(X).

c(X) :- f(X), not d(X).
d(X) :- f(X), not c(X).

:- a(X), not c(X).
```

The ground instantiation of $P$ (i.e., $ground(P)$) is of the form:

```
f(1).

a(1) :- f(1), not b(1).
b(1) :- f(1), not a(1).

c(1) :- f(1), not d(1).
d(1) :- f(1), not c(1).

:- a(1), not c(1).
```

Let us consider the interpretation $M_1 = \{f(1), a(1), c(1)\}$. $M_1$ satisfies all rules in $ground(P)$ and so it is a model of $P$. In this case, the GL-reduct of $P$ w.r.t. $M_1$ is the following program:

```
f(1).
a(1) :- f(1).
c(1) :- f(1).
```

Since $M_1$ is also a $\subset$-minimal model of the reduct then $M_1$ is an answer set of $P$. Similarly, $M_2 = \{f(1), b(1), c(1)\}$ and $M_3 = \{f(1), b(1), d(1)\}$ are answer sets of $P$. ⌟

Weak constraints do not affect the coherence of ASP programs. On the other hand, they are used to weight answer sets and so allows to define preferences among answer sets of ASP programs. More precisely, let $P$ be an ASP program with weak constraint and $M \in AS(P)$ be an answer set of $P$, then the set of weak constraint violations $vs(P, M)$ is defined as

$\{(w, l, \vec{t}) \mid\sim l_1, \ldots, l_n \ [w@l, \vec{t}] \in ground(P), I \models l_1, \ldots, l_n\}$. Let $l$ be an integer, then the cost of the answer set $M$ at level $l$ is defined as $\mathcal{C}(P, M, l) = \sum_{(w,l,\vec{t}) \in vs(P,M)} w$. Then we say that $M$ is *dominated* by an answer set $M' \in AS(P)$ if the there exists an integer $l$ such that $\mathcal{C}(P, M, l) > \mathcal{C}(P, M', l)$ and for each $l' > l$, $\mathcal{C}(P, M, l') = \mathcal{C}(P, M', l')$. Finally, $M$ is an *optimal answer set* of $P$ if $M$ is not dominated by any $M' \in AS(P)$. We denote by $OptAS(P)$ the set of optimal answer set of the program $P$.

▶ **Example 2.** Let us consider the program $P$ from Example 1 augmented with the following weak constraints:

```
:~ b(X). [1@2,X]
:~ c(X). [1@1,X]
```

Let us now compute the weak constraint violations for each answer set. More precisely, $vs(P, M_1) = \{(1, 1, 1)\}$, $vs(P, M_2) = \{(1, 1, 1), (1, 2, 1)\}$, and $vs(P, M_3) = \{(1, 2, 1)\}$. For $M_1$ we have that $\mathcal{C}(P, M_1, 1) = 1$ and $\mathcal{C}(P, M_1, 2) = 0$. For $M_2$ we have that $\mathcal{C}(P, M_2, 1) = \mathcal{C}(P, M_2, 2) = 1$. For $M_3$, instead, we have that $\mathcal{C}(P, M_1, 1) = 0$ and $\mathcal{C}(P, M_1, 2) = 1$. Thus, $M_1$ is not dominated neither by $M_2$ nor $M_3$ and so $M_1$ is an optimal answer set. On the other hand $M_1$ dominates both $M_2$ and $M_3$ and so the only optimal answer set is $M_1$. ⌟

## 3 ASP Modeling

The ASP language offers a natural and declarative way for modeling hard combinatorial problems. More precisely, it is possible to model such problems (typically $NP$-complete problems) into an ASP program whose answer sets correspond to solutions of the modeled problem. The standard ASP modeling methodology is referred to as *Guess&Check* [27]. According to such methodology, when we approach a problem we need to define two types of rules: (*i*) rules that generate the space of candidate solutions, referred to as *Guess*, and (*ii*) rules that filter out generated solutions that do not satisfy required properties for being a solution of the modeled problem, referred to as *Check*. Intuitively, the *Guess* part of an ASP program contains choice rules and/or rules forming even loop through negation. Then, the *Check* part of an ASP program discards candidate solution typically by means of stratified rules (i.e. no loops through negation) and strong constraints. For example, the rules *a :- not b* and *b :- not a* (i.e., *Guess* part) forms an even loop through negation and generates two candidate solutions: one in which is included *a* instead of *b* and the other one in which *b* is included instead of *a*. Then a constraint of the form *:- a* (i.e. *Check* part) can be used to discard all those solutions which include *a*. Building on this simple example, we now demonstrate how the *Guess&Check* methodology can be applied in practice by presenting ASP encodings of three classical problems in graph theory: Reachability, Clique, and Hamiltonian Path.

### Reachability Problem

Given a directed graph $G$, the reachability problem consists of determining each pair of nodes $(x, y)$ such that $y$ can be reached from $x$. In database terms, this corresponds to computing the transitive closure of the *edge* relation, which is a classical use case for deductive databases. ASP offers a concise and natural encoding of this problem. The following program computes all pairs of reachable nodes, represented as atoms over the predicate *reach*/2:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Intuitively, the first rule derives all pairs of nodes $x, y$ such that $y$ is directly linked to $x$ by an edge. The second rule, instead, recursively derives new pairs of reachable nodes. Intuitively, if the node $z$ is reachable from $x$ (i.e., $reach(x, z)$ is true), and $z$ is linked to $y$ by an edge (i.e., $edge(z, y)$ is true) then $y$ is reachable from $x$ and so $reach(x, y)$ is derived.

### Clique Problem

Given an undirected graph $G = \langle V, E \rangle$ then a subset of nodes $C \subseteq V$ is a *clique* if for each pair of distinct node $x, y \in C$, $\{x, y\} \in E$. The clique problem consists of verifying whether the input graph $G$ admits a clique. The following ASP program can be used to compute cliques of arbitrary undirected graphs.

```
{clique(X)} :- node(X).

:- clique(X), clique(Y), X!=Y, not edge(X,Y).
```

As it can be observed, the Clique problem can be modeled by means of two rules. The choice rule guesses whether a node $x$ should be included or not in the candidate clique by means of atoms over predicate *clique*. As a result it generates the space of possible solution that are all possible subset of nodes (i.e., all candidate cliques). Then, the constraint imposes that it is not possible that we have selected two distinct nodes $x$ and $y$ and they are not linked by an edge. As a result, answer set of this program corresponds to subset of nodes which are strongly connected, and so they form a clique. Thus, the above program can be used to computes cliques of arbitrary graphs encoded as facts over predicates *node* and *edge*. For example, the following facts encodes a complete graph with three nodes:

```
node(1).   node(2).   node(3).
edge(1,2).   edge(2,1).
edge(1,3).   edge(3,1).
edge(2,3).   edge(3,2).
```

The ASP encoding presented so far can be naturally extended to capture the optimization variant of the Clique problem, where the goal is to compute the largest cliques of a given graph. This can be achieved by augmenting the original encoding with the following weak constraint:

```
:~ node(X), not clique(X). [1@1,X]
```

More precisely, the weak constraint assigns a penalty of 1 for each node $x$ that is not included in the clique. Consequently, the cost of an answer set corresponds to the number of nodes excluded from the clique. By minimizing this cost, we are indeed maximizing the size of the clique, thus ensuring that the optimal answer sets correspond to the largest cliques in the input graph.

### Hamiltonian Path Problem

Another interesting problem in Graph Theory is the well-known Hamiltonian Path Problem which played a relevant role in the ASP literature because it is one of the canonical examples of non-tight [29] program (i.e., rules with positive recursion). The Hamiltonian Path problem consists of determining the existence of a finite sequence of nodes $v_1, \ldots, v_k$ of an undirected graph $G = \langle V, E \rangle$, such that: (i) every node $v \in V$ appears exactly once in the sequence, and (ii) for each $1 \leq i < k$, $\{v_i, v_{i+1}\} \in E$. A suitable encoding for such problem is the following:

```
% Guess path and starting node
{start(X)} :- node(X).
{inPath(X,Y)}:-edge(X,Y).

% Check starting node
foundStart :- start(X).
:- not foundStart.
:-start(X), start(Y), X!=Y.

% Check incoming/outgoing edges
:-inPath(X1,Y), inPath(X2,Y), X1!=X2.
:-inPath(X,Y1), inPath(X,Y2), Y1!=Y2.
:-start(X), inPath(Y,X).

% Check each node is reached
reach(X):-start(X).
reach(Y):-reach(X), inPath(X,Y).
:- node(X), not reach(X).
```

The above program contains a *Guess* part made of two choice rules which generate the space of candidate solution. In particular, the first one guesses whether a node must be considered as starting node or not by means of atom over predicate *start*. The second one, instead, guesses a candidate path in the graph $G$ by selecting a subset of the edges by means of atoms over predicate *inPath*. All the renaming rules correspond to the *Check* part that ensures conditions for being an hamiltonian path are met by the candidate solution.

More precisely, the first check block discards those solutions having zero or more than one starting nodes. Specifically, the first rule derives an atom $foundStart$ if there exists at least one starting node; the first constraint imposes that is not possible that $foundStart$ is derived as false; and, finally, the second constraint imposes that is not possible to have two distinct nodes, $x$ and $y$, which are both selected as starting nodes. Thus, the first check block ensures that a solution (i.e., an answer set) contains exactly one starting node. The second check block, instead, is aimed at ensuring that edges included in the path are such that there is at most one incoming (resp. outgoing) edge for each node of the graph; and no incoming edges for the starting node. In this way, each node appears at most once along the path, and the guessed path does not return to the starting node, thereby preventing the computation of an Hamiltonian cycle. Finally, the last check block computes nodes that can be reached from the starting node and by means of the final constraint imposes that each node in the graph must be reached. As a result, this program can be used to compute Hamiltonian paths of arbitrary undirected graphs.

## 4 ASP Solving and Recent Advancement

ASP solving is a compelling research area in the realm of Artificial Intelligence and Logic Programming as the availability efficient systems able to compute answer set of ASP programs makes ASP a concrete solution to many real world problems [28, 39]. In this section we are going to present the standard *Ground&Solve* approach employed by state-of-the-art ASP systems as well as novel evaluation technique known as Compilation-based ASP solving.

### 4.1 The Ground&Solve Approach

According to the *Ground&Solve* approach, answer sets of ASP programs are computed in two steps, namely *grounding* and *solving*. During *grounding*, an input program $P$ is instantiated by computing ground instantiations of each rule of $P$ (i.e. $ground(P)$). This phase is typically carried out by grounder systems such as GRINGO and IDLV which try to keep the size of the

**Algorithm 1** CDCL with Post Propagation.

```
   Input : An ASP program Π
 1 begin
 2   I := ∅
 3   Loop
 4     I = eagerPropagation(Π, I)
 5     if I is inconsistent then
 6       restoreConsistency(Π, I)
 7       if I is inconsistent then
 8         return ⊥
 9       end
10     end
11     else if I is total then
12       return I
13     end
14     I := I ∪ choose(Π)
15   end
16 end
```

ground program as small as possible by avoiding the materialization of trivially satisfied rules (i.e., the so called *intelligent grounding* [14]). At solving stage instead, a solver module, such as WASP [2] or CLASP [37], performs a propositional search of answer sets of the program $P$ by implementing an extension of the CDCL (Conflict-Driven Clause Learning) algorithm equipped with ASP-specific *propagators* [47] reported in Algorithm 1. Intuitively, the solver module computes an answer set incrementally starting from empty interpretation $I$. At each iteration of the solving loop the solver first computes deterministic inferences (i.e., assignment of truth values unequivocally) implied by the current interpretation $I$ (i.e., eager propagation). Basically, eager propagation, corresponding to unit propagation in SAT solvers [51], which amounts to deriving the deterministic consequences of $I$ that are implied by the program in input. As soon as no further consequences can be derived by the eager propagation, then the interpretation $I$ is analyzed. If $I$ is inconsistent then the solver analyzes conflictual literals to resolve the conflict. If the consistency of $I$ cannot be restored then the solver terminates by returning $\bot$ which means no answer set exists and so $P$ is incoherent. Otherwise, the consistency of $I$ is restored by backjumping to the decision which led to the conflict. Otherwise, if $I$ is total and consistent then the solver stops by returning $I$ as answer set of $P$. If none of these conditions holds (i.e., $I$ is consistent but not total) then a new literal is heuristically chosen, added to $I$, and the loop continues.

## 4.2 Compilation-based ASP Solving

One of the well-known limitations of the *Ground&Solve* approach discussed so far is the grounding bottleneck problem [38]. In many cases of practical interest [15, 57], the grounding phase alone exhausts the available computational resources – both in terms of time and memory – thereby preventing the solving stage from even starting.

In recent years, several techniques have been proposed to alleviate the grounding bottleneck. Among them, one of the most promising approaches is compilation-based ASP solving [19, 53, 23, 24]. The main idea behind compilation-based techniques is to avoid the grounding phase by compiling a program into rule specific propagators. Obtained propagators are able to simulate inferences coming from ground rules without explicitly materializing them, thus completely bypassing the grounding phase.

More in detail, let $\Pi$ be an ASP program, then $\Pi$ is first compiled into an ad-hoc solver, referred to as $\Pi$-solver. The obtained solver is tailored on the program $\Pi$ and so it can be used to evaluate every instance of $\Pi$. Thus, given an instance $F$, that is a set of facts, the $\Pi$-solver searches for an answer set of $\Pi \cup F$ without materializing any ground rule.

### Compilation Stage

In the compilation stage, a program $\Pi$ is first transformed by applying a sequence of rewriting steps which give as output two programs, namely $\Pi_{prop}$ and $\Pi_{gen}$. The program $\Pi_{prop}$, referred to as *propagator program*, contains rules and constraints which simulate inferences of rules in $\Pi$. Thus, by compiling rules of the propagator program into ad-hoc eager propagators, it is possible to obtain a module of the resulting solver, namely *Propagator*, which can be used to compute rules inferences during solving. On the other hand, the program $\Pi_{gen}$, referred to as *generator program*, contains rules defining the domain of predicates appearing in $\Pi_{prop}$. These rules allow to compute ground atoms required to evaluate a program $\Pi$ w.r.t. an instance $F$. Thus, by compiling rules in $\Pi_{gen}$ into ad-hoc bottom up evaluation procedure, it is possible to obtain a module of the resulting solver, namely *Generator*, which generates ground atoms over predicates in $\Pi_{prop}$. Finally, *Generator* and *Propagator* modules are integrated with a SAT solver to obtain the $\Pi$-solver.

▶ **Example 3.** Let us consider the following program:

```
% Rule r1
a(X) :- d(X), not na(X).
% Rule r2
na(X) :- d(X), not a(X).
% Rule r3
:- a(X), a(Y), X<Y.
```

If we consider rules $r_1$ and $r_2$, and a possible constant $x$, then the following propagations may derive from them:

- $a(x)$ is true if and only if $d(x)$ is true and $na(x)$ is false.
- $na(x)$ is true if and only if $d(x)$ is true and $a(x)$ is false.

On the other hand, if we consider the constraint $r_3$, the following propagations ensure that $r_3$ is satisfied:

- if there exists $x$ such that $a(x)$ is true then for each value $y > x$, $a(y)$ must be false;
- if there exists $y$ such that $a(y)$ is true then for each value $x < y$, $a(x)$ must be false.

These inferences can be simulated by means of Algorithm 2 and 3 which report, respectively, ad-hoc propagators for $r_1$ and $r_3$. We avoid to report the propagator for $r_2$, as it is identical to that of $r_1$ except for swapping the predicate $a$ with $na$, and vice versa.

On the other hand, the *generator* module computes ground atoms over predicates $a$ and $na$, and so, the generator program contains the following rules:

```
a(X) :- d(X).
na(X) :- d(X).
```

A bottom up evaluation of generator rules is reported in Algorithm 4. Basically, it iterates over atoms of the form $d(x)$ in the program instance $F$ and generates $a(x)$ and $na(x)$. ⌟

### Solving Stage

Given a program instance $F$, the $\Pi$-solver searches for an answer set of $\Pi \cup F$. To this end, the instance $F$ is given as input to the generator module which generates the set of ground atoms that are relevant for the computation of answer sets of $\Pi \cup F$. Then, generated

**Algorithm 2** Propagator for $a(X)$ :- $d(X), not\ na(X)$.

```
Input : An interpretation I, and a literal l ∈ I
begin
    if pred(l) = "a":
        x := l[0]
        if l ∈ I⁺:
            propagate {d(x), not na(x)}
        else:
            if d(x) ∈ I⁺:
                propagate {na(x)}
            else if not na(x) ∈ I⁻:
                propagate {not d(x)}
    if pred(l) = "d":
        x := l[0]
        if l ∈ I⁺:
            if not na(x) ∈ I⁻:
                propagate {a(x)}
        else:
            propagate {not a(x)}
    if pred(l) = "na":
        x := l[0]
        if l ∈ I⁻:
            if d(x) ∈ I⁺:
                propagate {a(x)}
        else:
            propagate {not a(x)}
```

**Algorithm 3** Propagator for :- $a(X), a(Y), X < Y$.

```
Input : An interpretation I, and a literal l ∈ I
begin
    if pred(l) = "a" and l ∈ I⁺:
        x := l[0]
        for all y ∈ U_Π s.t. y > x:
            propagate not a(y)
        y := l[0]
        for all x ∈ U_Π s.t. x < y:
            propagate not a(x)
```

**Algorithm 4** Generator Module.

```
Input : A program instance F
Input : A set of ground atoms At
begin
    At = ∅
    for each d(x) ∈ F
        At = At ∪ {a(x)}
        At = At ∪ {na(x)}
    return At
```

atoms become propositional variables of the SAT solver which starts the CDCL. During CDCL, the SAT solver relies on the *Propagator* module to compute all the inferences coming from compiled rules. Obtained inferences are propagated by the SAT solver and the CDCL continues. As soon as a conflict is generated in the SAT solver, then the *Propagator* is used to compute the so called reason of each inference (i.e., the literals which caused such inferences). Obtained reasons are analyzed by the SAT solver which will restore consistency, if possible. If consistency cannot be restored then the SAT solver exits by returning incoherent (i.e., no answer set of $\Pi \cup F$ exists). Otherwise, as soon as the SAT solver computes a total and consistent interpretation then an answer set of $\Pi \cup F$ is obtained.

▶ **Example 4.** Let us consider the compiled solver of Example 3 and the program instance $F = \{d(1), d(2)\}$. In this case, the *Generator* module generate ground atoms $a(1)$, $na(1)$, $a(2)$, and $na(2)$. These atoms become decision variables of the SAT solver which starts the CDCL. First of all, the SAT solver heuristically chooses an atom to be assigned. For example, let us assume that the SAT solver chooses $a(1)$ that is assigned to true. The atom $a(1)$ is given to the propagator module that, from the rule $r_1$, derives *not* $na(1)$ and, from the constraint $r_3$, derives *not* $a(2)$. At this point the SAT solver receives $\{not\ na(1), not\ a(2)\}$ and assign both $na(1)$ and $a(2)$ to false. Then, the SAT solver invokes again the *Propagator* module by giving as input *not* $a(2)$. In this case, the propagator module infers, from rule $r_2$, the atom $na(2)$ which is assigned by the SAT solver obtaining the total interpretation $I = \{d(1), d(2), a(1), na(2)\}$ that is an answer set of $\Pi \cup F$. ⌟

## Hybrid Compilation-based ASP Solving

Compilation-based ASP solving has proven highly effective for the evaluation of grounding-intensive ASP programs [53, 23]. However, experimental comparisons have shown that, on benchmarks where grounding is feasible, compilation-based techniques can introduce some overhead in solving time [23]. This is mainly due to the fact that the SAT solver has no ground rules and its heuristic is poorly informed. This observation motivated the development of hybrid compilation-based ASP solving [24]. The hybrid approach combines the strengths of the traditional *Ground&Solve* method with compilation-based ASP solving by exploiting compilation also for the grounding phase. Given a program $\Pi$, an arbitrary subset of its rules (possibly those without grounding issues) can be selected for grounding, while the remaining rules are compiled into ad-hoc propagators to be simulated during solving. Through specific rewritings that preserve the semantics of $\Pi$, the generator program, produced in the compilation stage, can be extended with the rules selected for grounding. For those rules the generator module not only produce head atoms but materialize all their ground instantiations. As a result, the obtained $\Pi$-solver receives, from the Generator module, atoms required to compute an answer sets as well as a set of clauses that encodes grounded rules (i.e., computed by the Clark's completion [17] applied on rules ground instantiations). In this setting, the SAT solver performs inferences on grounded rules directly from its internal clauses, while delegating the remaining rules inferences to the Propagator module, which contains rule-specific propagators for the non-grounded part. This hybrid method has been implemented on top of the first compilation-based solver, PROASP[23, 24], which supports a representative fragment of the ASP language, namely tight programs [29].

Recent experiments demonstrate that hybrid compilation-based ASP solving can achieve significant performance improvements [24]. In particular, the hybrid system PROASP has been compared with state-of-the-art solver CLINGO on hard benchmarks from the ASP Competitions [15], as well as on grounding-intensive instances drawn from the literature [53,

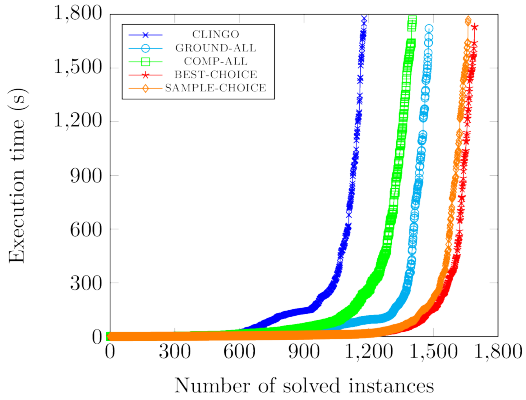


**Figure 1** Execution time comparison from [24].

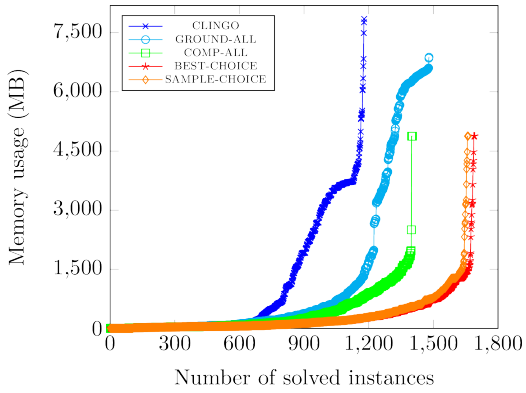


**Figure 2** Memory usage systems comparison from [24].

23]. The benchmark suite included 14 distinct problems for a total of 2366 instances. This evaluation investigated different blending strategies, selected according to a coarse-grained syntactic criterion that determines which types of propagators are applied. The overall results obtained in [24] are reported in Figures 1 and 2.

As it has been observed in [24], when no grounding issues are present, the approach can produce ad-hoc solvers that ground all rules, effectively simulating *Ground&Solve* via compilation. For benchmarks with severe grounding issues, it can generate fully grounding-free solvers that overcome the limitations of *Ground&Solve*, thus reducing the overall memory footprint. Finally, for all those case in the middle where grounding problems are confined to specific rules, the hybrid method can produce solvers that blend the advantages of both strategies. For further details and result we refer the reader to [24].

## 5 Beyond NP: ASP with Quantifiers

As it has been showed so far, ASP is an effective solution for modeling and solving hard combinatorial problems, typically $NP$-complete problems. This class of problems includes a vast majority of problems of practical interest, but, many important decision problems belong to higher complexity classes [62, 65]. Despite ASP allows to model problems up to the second level of the polynomial hierarchy, thanks to advanced techniques such as *saturation* [26], there are still a large number of problems which cannot be modeled in ASP. To this end, many language extensions have been proposed [11, 5, 36].

Among these, Answer Set Programming with Quantifiers (ASP(Q)) [5] introduced the notion of quantifier over answer sets of ASP programs to provide a natural and compact way of modeling problems in the entire polynomial hierarchy.

### The ASP(Q) Syntax and Semantics

An ASP(Q) program is an expression of the form:

$$\Box_1 P_1 \ldots \Box_n P_n : C : C^\omega \tag{1}$$

where $C^\omega$ is a set of weak constraints referred to as global weak constraints, $C$ is a stratified (i.e., no loop through negation) program with strong constraints, and for each $i \in \{1, \ldots, n\}$, $\Box_i \in \{\exists^{st}, \forall^{st}\}$ and $P_i$ is an ASP program (possibly with weak constraints). An ASP(Q) program is said to be *existential* if $\Box_1 = \exists^{st}$; otherwise *universal*. The coherence of ASP(Q) programs is defined inductively:

- $\exists^{st} P : C : C^\omega$ is coherent if there exists $M \in OptAS(P)$ such that $C \cup fix_P(M)$ is coherent;
- $\forall^{st} P : C : C^\omega$ is coherent if for all $M \in OptAS(P)$, $C \cup fix_P(M)$ is coherent;
- $\exists^{st} P\ \Pi$ is coherent if there exists $M \in OptAS(P)$ such that $\Pi_{P,M}$ is coherent;
- $\forall^{st} P\ \Pi$ is coherent if for all $M \in OptAS(P)$, $\Pi_{P,M}$ is coherent;

where $fix_P(M) = \{a. \mid M\} \cup \{\ :\!\!-\ a \mid a \in B_P \setminus M\}$, $\Pi$ is of the form (1), and $\Pi_{P,M}$ is obtained from $\Pi$ by replacing $P_1$ with $P_1 \cup fix_P(M)$.

Let $\Pi$ be an existential ASP(Q) program of the form (1), then $M_1 \in OptAS(P_1)$ is a *quantified answer set* of $\Pi$ if $\Box_2 P_2 \cup fix_{P_1}(M_1) \ldots \Box_n P_n : C : C^\omega$ is coherent. We denote by $QAS(\Pi)$ the set of quantified answer set of $\Pi$.

▶ **Example 5.** Let $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C : \emptyset$ be the following ASP(Q):

```
%@exists // Program P1
a :- not b.
b :- not a.
%@forall // Program P2
c :- a.
d :- b, not e.
e :- b, not d.
%@constraint // Program C
:-c.
```

In this case, the program $P_1$ has two (optimal) answer sets which are $M_1 = \{a\}$ and $M_1' = \{b\}$. Let us consider $M_1$ first. According to the semantics of ASP(Q), $M_1$ is a quantified answer set if $\forall^{st} P_2 \cup fix_{P_1}(M_1) : C : \emptyset$ is coherent. The program $P_2' = P_2 \cup fix_{P_1}(M_1) = P_2 \cup \{a.;\ :\!\!-\ b\}$ has only one (optimal) answer set that is $M_2 = \{a, c\}$. Thus, if $M_2$ makes the constraint program $C$ coherent then $M_1$ is a quantified answer set of $\Pi$. This is not the case, since the program $C \cup fix_{P_2'}(M_2) = C \cup \{a.;\ :\!\!-\ b; c.;\ :\!\!-\ d,\ :\!\!-\ e\}$ is incoherent. Thus $M_1$ is not a quantified answer set. On the other hand, $M_1'$ is a quantified answer set if $\forall^{st} P_2 \cup fix_{P_1}(M_1') : C : \emptyset$ is coherent. In this case, the program $P_2' = P_2 \cup fix_{P_1}(M_1') = P_2 \cup \{b.;\ :\!\!-\ a\}$ has two (optimal) answer sets which are $M_2' = \{b, d\}$ and $M_2'' = \{b, e\}$. Thus, $M_1'$ is a quantified answer set if both $M_2'$ and $M_2''$ makes the program $C$ coherent. In particular, we have that the program $C \cup fix_{P_2'}(M_2') = C \cup \{b.;\ :\!\!-\ a; d.;\ :\!\!-\ c,\ :\!\!-\ e\}$ has exactly one answer set that is $M_2'$ and so it is coherent. Similarly, the program $C \cup fix_{P_2'}(M_2'') = C \cup \{b.;\ :\!\!-\ a; e.;\ :\!\!-\ c,\ :\!\!-\ d\}$ has exactly one answer set that is $M_2''$ and so it is coherent. Since the program $C$ is coherent for each answer set of $P_2'$ then $M_1'$ is a quantified answer set of $\Pi$. ⌟

While local weak constraints (weak constraint in subprograms $P_i$) may affect the coherence of ASP(Q) programs, global weak constraints in $C^\omega$ do not. In particular, they can be used to express preferences among quantified answer sets of existential ASP(Q) programs. More

precisely, let $\Pi$ be an existential ASP(Q) program, $M \in QAS(\Pi)$ be an answer set of $P$, and $l$ be an integer, then the cost of $M$ at level $l$ is defined as $\mathcal{C}(C^\omega, M, l) = \sum_{(w,l,\vec{t}) \in vs(C^\omega, M)} w$. Let $M_1, M_2 \in QAS(\Pi)$, then $M_1$ is *dominated* by $M_2$ if the there exists an integer $l$ such that $\mathcal{C}(C^\omega, M_1, l) > \mathcal{C}(C^\omega, M_2, l)$ and for each $l' > l$, $\mathcal{C}(C^\omega, M_1, l') = \mathcal{C}(C^\omega, M_2, l')$. Finally, $M \in QAS(\Pi)$ is an *optimal quantified answer set* of $\Pi$ if $M$ is not dominated by any $M' \in QAS(\Pi)$. We denote by $OptQAS(P)$ the set of optimal quantified answer sets of $\Pi$.

## 5.1 Modeling Examples

The ASP(Q) formalism provides a powerful and intuitive framework for modeling complex problems that go beyond the class $NP$. With the availability of efficient ASP(Q) systems, such as PYQASP[32] and QASP[3], this approach has been successfully employed in diverse application domains, including outlier detection [9], probabilistic answer set programming [8, 6, 7], among many others [34, 33, 5, 54].

To showcase the expressiveness of ASP(Q), we present two illustrative modeling examples. These cases demonstrate how ASP(Q) naturally captures complex reasoning tasks and effectively models both decision and optimization problems beyond the $NP$ class.

### Clique Coloring Problem

An interesting problem in graph theory is the Clique Coloring problem [52]. Given a graph $G = (V, E)$ and an integer $k$, a $k$-clique-coloring [52] is a function $c : V \to \{1, \ldots, k\}$ such that every maximal clique of $G$ contains two vertices of different colors. Checking whether a graph $G$ has a $k$-coloring is $\Sigma_2^P$-complete [62].

Based on the problem description, here we can construct an ASP(Q) program made of two quantifiers. First, an existential quantifiers searches a possible $k$-coloring function $c : V \to \{1, \ldots, k\}$ and then a universal quantifiers ensure that for each maximal clique, at least two nodes have different colors. Thus, a suitable ASP(Q) for the clique coloring problem is the following:

```
%@exists // Program P1
{assign(N,C)}:-node(N),color(C).

colored(N) :- assign(N,C).
:-node(N), not colored(N).
:-assign(N,C1), assign(N,C2), C1!=C2.

%@forall // Program P2
{clique(N)}:-node(N).

remove(N1):- node(N1), clique(N2), N1!=N2, not edge(N1,N2).

:-clique(N1), remove(N1).
:-node(N1), not remove(N1), not clique(N1).

%@constraint // Program C
diffColors :- clique(N1),clique(N2), assign(N1,C1),assign(N2,C2), C1!=C2.
:- not diffColors.
```

Intuitively, rules of the program $P_1$ guess a possible assignment of colors to nodes and then ensure that each node is colored exactly with one color. Thus, (optimal) answer sets of the program $P_1$ correspond to possible $k$-colorings of nodes of the input graph $G$. Rules of the program $P_2$, instead, guesses a subset of nodes of $G$ and then check that selected nodes form a maximal clique (i.e. there is a edge between each pair of nodes and no other node can be added to the clique). Finally, the constraint program $C$ ensure that at least two nodes of the

clique are colored with different colors. Thus, a quantified answer set is an (optimal) answer set $M_1$ of $P_1$ (i.e., a possible $k$-coloring for nodes of $G$) such that, for every (optimal) answer set of $P_2 \cup \mathit{fix}_{P_1}(M_1)$ (i.e. every maximal clique of $G$), the program $C$ is coherent (i.e. two nodes are colored with different colors). Thus, a quantified answer set is a $k$-clique-coloring for the input graph $G$.

### Path Vapnik–Chervonenkis Dimension

The Vapnik–Chervonenkis (VC) dimension is a key concept in machine learning theory [68]. It quantifies the capacity of a set of functions that a statistical classification algorithm can learn [10]. In statistical learning theory, it is used to estimate probabilistic upper bounds on a classification model's test error [67]. Moreover, the VC-dimension has been also studied for set systems induced by graph properties [48], such as cliques, paths, etc. In particular, the Path VC-dimension problem is defined as follows.

Given a graph $G = (V, E)$ and then a subset of nodes $X \subseteq V$ is *shattered* by subpaths of $G$ if for each $S \subseteq X$ there exists a subpath in $G$ that contains all nodes in $S$ and none of the nodes in $X \setminus S$. The *Path VC-dimension* [48] of $G$ is the size of the largest set $X$ that is shattered by $G$. Deciding whether the Path VC-dimension of $G$ is greater or equal than an integer $k$ is $\Sigma_3^P$-complete problem [62]. Thanks to availability of weak constraints in ASP(Q), it is possible to model also optimization problems. Thus, in this case we describe a suitable ASP(Q) encoding for computing the largest set $X$ which is shattered by subpaths in $G$.

To this end, we need to use three quantifiers. More precisely, an existential quantifier is required to look for possible set $X \subseteq V$. Then, to check that a set $X \subseteq V$ is shattered by subpaths of $G$, two quantifiers are needed. First, a universal quantifier iterates over possible set $S \subseteq X$ and, finally, an existential one checks the existence of a subpath of $G$ which contains all nodes in $S$ and none of the nodes in $X \setminus S$. More in detail, an ASP(Q) encoding for the Path VC-dimension problem is the following:

```
%@exists // Program P1
{inX(N)} :- node(N).
%@forall // Program P2
{inS(N)} :- inX(N).
%@exists // Program P3
{inPath(X,Y)} :- edge(X,Y).
:- inPath(X,Y1), inPath(X,Y2), Y1!=Y2.
:- inPath(X1,Y), inPath(X2,Y), X1!=X2.

{start(N)} :- node(N).
foundStart :- start(N).
:- start(N1), start(N2), N1!=N2.

reach(Y) :- inpath(X,Y), start(X).
reach(Y) :- reach(X), inpath(X,Y).
%@constraint // Program C
:- inS(N), not reach(N).
:- inX(N), not inS(N), reach(N).
%@global // Global weak constraints C^w
:~ inX(N). [-1@1,N]
```

Intuitively, (optimal) answer sets of the program $P1$ are in one-to-one correspondence with possible $X \subseteq V$. Similarly, (optimal) answer sets of the program $P_2$ correspond to possible set $S \subseteq X$. Then, the program $P_3$ allow to compute possible subpath of $G$. Basically, choice rules of $P_3$ guess a subset of edges and a possible starting node $N$. Then, remaining rules ensure that $(i)$ for each node at most one outgoing (resp. incoming) has been selected; and $(ii)$ exactly one starting node has been selected. Finally, nodes that are reachable from the

starting node correspond to nodes visited by the guessed subpath of $G$. The constraint program $C$, instead, imposes that all nodes in $S$ must be reached and none of the nodes in $X \setminus S$ must be reached. Thus, by following the ASP(Q) semantics, a quantified answer set is a (optimal) answer set $M_1$ of $P_1$ (i.e. a set $X \subseteq V$) such that for each (optimal) answer set $M_2$ of $P_2' = P_2 \cup \mathit{fix}_{P_1}(M_1)$ (i.e. for each set $S \subseteq X$), there exists an (optimal) answer set $M_1$ of the program $P_3' = P_3 \cup \mathit{fix}_{P_2'}(M_2)$ (i.e., a subpath of $G$) such that $C \cup \mathit{fix}_{P_3'}(M_3)$ is coherent (i.e., all node in $S$ are reached and none of the node in $X \setminus S$ is reached). Thus, each quantified answer set corresponds to a set $X \subseteq V$ that is shattered by subpaths of $G$.

Finally, global weak constraints in $C^\omega$ assign a cost to each quantified answer set. In particular, the only global weak constraint is $:\sim in(N).\ [-1@1, N]$, and so it adds a cost of $-1$ for each node $N$ which is included in $X$. Thus, the cost of each quantified answer set is equal to $-|X|$, for some $X \subseteq V$. In this case, minimizing the cost of the quantified answer set corresponds to maximize the cardinality of the set $X$ and so an optimal quantified answer set corresponds to one of the largest $X$ that is shattered by subpaths of $G$.

## 6 Large Language Models and Answer Set Programming: A Promising Research Direction

Natural Language Processing (NLP) techniques [45] have profoundly transformed human–computer interaction, enabling users to accomplish tasks that once demanded substantial expertise and effort. Among the most impactful advances, Large Language Models (LLMs)[70] have achieved remarkable performance across a wide spectrum of tasks, including natural language understanding[59], dialogue systems [58], and code generation [44].

The combination of LLMs with Answer Set Programming (ASP) can be fruitfully explored in several directions. Two of them have gained attention recently, namely: (*i*) the study of methods for automatic composition of ASP programs from natural language specifications, and (*ii*) the development of neurosymbolic AI systems that exploit ASP as a reasoning component. Recent efforts have indeed expanded LLM research toward advanced reasoning, knowledge representation, and logical formalisms [12, 50, 66]. Yet, despite such progress, significant evaluation gaps persist. While numerous benchmarks assess LLMs on imperative and web-oriented languages (e.g., C++, Java, HTML)[31], attempts to systematically evaluate declarative paradigms like ASP remain relatively scarce [12, 43, 18]. In particular LLASP is a fine-tuned and lightweight model specifically designed to capture fundamental ASP program patterns. It is based on a dedicated dataset covering a broad range of core problem specifications expressible in ASP. LLASP produces ASP programs of good outperforming its non-fine-tuned counterpart and the majority of competitive large language model baselines, particularly in terms of semantic accuracy. LLASP supports a limited pattern library. On the other hand, Ricca et al. [12] moved the first step towards automating the composition of Answer Set Programming (ASP) specifications. They proposed a two-step architecture, implemented in the NL2ASP tool, which generates ASP programs from natural language descriptions. The approach leverages neural machine translation to convert natural language into Controlled Natural Language (CNL) statements, which are subsequently translated into ASP code by the CNL2ASP component. An experimental evaluation confirmed the feasibility and promise of this method, which however is limited to graph problems and to some extend to the limits of the underlying CNL system. Fully automatic ASP code generation remains difficult, although important initial steps have been taken.

Concerning neurosymbolic AI, several works have already demonstrated the utility of integrating LLMs with ASP. Nye et al.[56] proposed a dual-system model based on GPT-3 that combines semantic parsing with reasoning modules. Yang et al.[69] showed that

LLMs can act as few-shot semantic parsers, producing ASP logical forms without retraining. Other efforts exploited prompt engineering to encode and solve logic puzzles in ASP [42], or introduced hybrid systems such as LLM2LAS [46], which combines LLMs with ILASP [49] to acquire commonsense knowledge from narrative QA. ASP has also been integrated with LLMs for natural language understanding in the STAR framework [60]. More recently, new benchmarks have been proposed to evaluate ASP-specific reasoning abilities of LLMs, including entailment, verification, and answer-set computation [61].

In summary, the intersection of LLMs and ASP represents a highly promising research direction. On one side, it holds the potential to make ASP development more accessible and automated; on the other, it paves the way toward neurosymbolic systems that combine the expressive reasoning power of ASP with the generative and interpretative capabilities of LLMs.

## 7 Conclusion

ASP is a powerful paradigm for Knowledge Representation and Reasoning, offering a declarative framework that combines intuitive problem modeling with efficient computational techniques.

This paper has provided an overview of ASP, moving from the fundamental aspects of the language to programming methodologies and the computation of answer sets. It has also outlined some recent extensions that expand ASP expressive power and enhance ASP program evaluation. Finally, attention was drawn to emerging tools that leverage natural language to generate ASP programs.

Taken together, these developments highlight both the maturity and the continuing evolution of ASP. While the paradigm already provides robust solutions in a range of application domains, ongoing research on efficiency, usability, and integration with other AI technologies will further broaden its impact. In this light, ASP remains not only a valuable tool for today's reasoning and decision-making problems but also a fertile ground for future innovations in declarative AI.

## References


1 Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017. doi:10.1007/978-3-319-61660-5_19.

2 Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015. doi:10.1007/978-3-319-23264-5_5.

3 Giovanni Amendola, Bernardo Cuteri, Francesco Ricca, and Mirek Truszczynski. Solving problems in the polynomial hierarchy with ASP(Q). In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, pages 373–386. Springer, 2022. doi:10.1007/978-3-031-15707-3_29.

4 Giovanni Amendola, Carmine Dodaro, Nicola Leone, and Francesco Ricca. On the application of answer set programming to the conference paper assignment problem. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea, editors, *AI*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings*, volume 10037 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2016. doi:10.1007/978-3-319-49130-1_13.

5 Giovanni Amendola, Francesco Ricca, and Miroslaw Truszczynski. Beyond NP: quantifying over answer sets. *Theory Pract. Log. Program.*, 19(5-6):705–721, 2019. doi:10.1017/S1471068419000140.

6 Damiano Azzolini, Giuseppe Mazzotta, Francesco Ricca, and Fabrizio Riguzzi. Most probable explanation in probabilistic answer set programming. In *IJCAI*. ijcai.org, 2025. doi:10.24963/IJCAI.2025/1006.

7 Damiano Azzolini, Giuseppe Mazzotta, Francesco Ricca, and Fabrizio Riguzzi. A novel framework for reasoning over optimization problems in probabilistic answer set programming. In *Proceedings of the 22st International Conference on Principles of Knowledge Representation and Reasoning, KR*, 2025.

8 Damiano Azzolini and Fabrizio Riguzzi. Probabilistic answer set programming with discrete and continuous random variables. *Theory Pract. Log. Program.*, 25(1):1–32, 2025. doi:10.1017/S1471068424000437.

9 Pierpaolo Bellusci, Giuseppe Mazzotta, and Francesco Ricca. Modelling the outlier detection problem in ASP(Q). In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, pages 15–23. Springer, 2022. doi:10.1007/978-3-030-94479-7_2.

10 Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the vapnik-chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989. doi:10.1145/76359.76371.

11 Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP*, 16(5-6):570–586, 2016. doi:10.1017/S1471068416000387.

12 Manuel Borroto, Irfan Kareem, and Francesco Ricca. Towards automatic composition of ASP programs from natural language specifications. In *IJCAI*, volume abs/2403.04541, page to appear. ijcai.org, 2024. doi:10.48550/arXiv.2403.04541.

13 Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

14 Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017. doi:10.3233/IA-170104.

15 Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016. doi:10.1016/J.ARTINT.2015.09.008.

16 Matteo Cardellini, Paolo De Nardi, Carmine Dodaro, Giuseppe Galatà, Anna Giardini, Marco Maratea, and Ivan Porro. A two-phase ASP encoding for solving rehabilitation scheduling. In *RuleML+RR*, volume 12851 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2021. doi:10.1007/978-3-030-91167-6_8.

17 Keith L. Clark. Negation as failure. In *Logic and Data Bases*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press. doi:10.1007/978-1-4684-3384-5_11.

18 Erica Coppolillo, Francesco Calimeri, Giuseppe Manco, Simona Perri, and Francesco Ricca. Llasp: fine-tuning large language models for answer set programming. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, KR '24, 2024. doi:10.24963/kr.2024/78.

19 Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *IJCAI*, pages 1688–1694. ijcai.org, 2020. doi:10.24963/IJCAI.2020/234.

20 Bernardo Cuteri, Kristian Reale, and Francesco Ricca. A logic-based question answering system for cultural heritage. In *JELIA*, volume 11468 of *LNCS*, pages 526–541. Springer, 2019. doi:10.1007/978-3-030-19570-0_35.

21 Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. doi:10.1145/502807.502810.

22 Carmine Dodaro, Giuseppe Galatà, Marco Maratea, and Ivan Porro. Operating room scheduling via answer set programming. In *AI*IA*, volume 11298 of *Lecture Notes in Computer Science*, pages 445–459. Springer, 2018. doi:10.1007/978-3-030-03840-3_33.

**23** Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Compilation of tight ASP programs. In Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Radulescu, editors, *ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023)*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 557–564. IOS Press, 2023. doi:10.3233/FAIA230316.

**24** Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Blending grounding and compilation for efficient ASP solving. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR*, 2024.

**25** Thomas Eiter, Michael Fink, Gianluigi Greco, and Domenico Lembo. Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.*, 33(2):10:1–10:51, 2008. doi:10.1145/1366102.1366107.

**26** Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995. doi:10.1145/200836.200838.

**27** Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995. doi:10.1007/BF01536399.

**28** Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016. doi:10.1609/AIMAG.V37I3.2678.

**29** Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory Pract. Log. Program.*, 3(4-5):499–518, 2003. doi:10.1017/S1471068403001765.

**30** Esra Erdem and Volkan Patoglu. Applications of ASP in robotics. *Künstliche Intell.*, 32(2-3):143–149, 2018. doi:10.1007/S13218-018-0544-X.

**31** Neil A. Ernst and Gabriele Bavota. Ai-driven development is here: Should you worry? *IEEE Softw.*, 39(2):106–110, 2022. doi:10.1109/MS.2021.3133805.

**32** Wolfgang Faber, Giuseppe Mazzotta, and Francesco Ricca. An efficient solver for ASP(Q). *Theory Pract. Log. Program.*, 23(4):948–964, 2023. doi:10.1017/S1471068423000121.

**33** Wolfgang Faber and Michael Morak. Evaluating epistemic logic programs via answer set programming with quantifiers. In *HYDRA/RCRA@LPNMR*, volume 3281 of *CEUR Workshop Proceedings*, pages 78–89. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3281/paper7.pdf.

**34** Wolfgang Faber, Michael Morak, and Lukás Chrpa. Determining action reversibility in STRIPS using answer set programming with quantifiers. In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2022. doi:10.1007/978-3-030-94479-7_4.

**35** Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018. doi:10.1007/S13218-018-0548-6.

**36** Jorge Fandinno, François Laferrière, Javier Romero, Torsten Schaub, and Tran Cao Son. Planning with incomplete information in quantified answer set programming. *Theory Pract. Log. Program.*, 21(5):663–679, 2021. doi:10.1017/S1471068421000259.

**37** Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl, 2016. doi:10.4230/OASICS.ICLP.2016.2.

**38** Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, pages 5450–5456. ijcai.org, 2018. doi:10.24963/IJCAI.2018/769.

**39** Martin Gebser, Marco Maratea, and Francesco Ricca. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60:41–95, 2017. doi:10.1613/jair.5373.

**40** Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.*, 9(3/4):365–386, 1991. doi:10.1007/BF03037169.

41 Giovanni Grasso, Salvatore Iiritano, Nicola Leone, Vincenzino Lio, Francesco Ricca, and Francesco Scalise. An asp-based system for team-building in the gioia-tauro seaport. In *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 40–42. Springer, 2010. doi:10.1007/978-3-642-11503-5_5.

42 Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. In *KR*, pages 374–383, 2023. doi:10.24963/KR.2023/37.

43 Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, KR '23, 2023. doi:10.24963/kr.2023/37.

44 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. doi:10.48550/arXiv.2406.00515.

45 Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

46 Irfan Kareem, Katie Gallagher, Manuel A. Borroto, Francesco Ricca, and Alessandra Russo. Using learning from answer sets for robust question answering with LLM. In *LPNMR*, volume 15245 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2024. doi:10.1007/978-3-031-74209-5_9.

47 Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Mag.*, 37(3):25–32, 2016. doi:10.1609/AIMAG.V37I3.2672.

48 Evangelos Kranakis, Danny Krizanc, Berthold Ruf, Jorge Urrutia, and Gerhard J. Woeginger. The vc-dimension of set systems defined by graphs. *Discret. Appl. Math.*, 77(3):237–257, 1997. doi:10.1016/S0166-218X(96)00137-0.

49 Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for inductive learning of answer set programs. *CoRR*, abs/2005.00904, 2020. arXiv:2005.00904.

50 Anna Sofia Lippolis, Mohammad Javad Saeedizade, Robin Keskisärkkä, Sara Zuppiroli, Miguel Ceriani, Aldo Gangemi, Eva Blomqvist, and Andrea Giovanni Nuzzolese. Ontology generation using large language models. In *ESWC (1)*, volume 15718 of *Lecture Notes in Computer Science*, pages 321–341. Springer, 2025. doi:10.1007/978-3-031-94575-5_18.

51 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

52 Dániel Marx. Complexity of clique coloring and related problems. *Theor. Comput. Sci.*, 412(29):3487–3500, 2011. doi:10.1016/J.TCS.2011.02.038.

53 Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro. Compilation of aggregates in ASP systems. In *AAAI*, pages 5834–5841. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20527.

54 Giuseppe Mazzotta, Francesco Ricca, and Mirek Truszczynski. Quantifying over optimum answer sets. *Theory Pract. Log. Program.*, 24(4):716–736, 2024. doi:10.1017/S1471068424000395.

55 Arindam Mitra, Peter Clark, Oyvind Tafjord, and Chitta Baral. Declarative question answering over knowledge bases containing natural language text with answer set programming. In *AAAI*, pages 3003–3010. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33013003.

56 Maxwell I. Nye, Michael Henry Tessler, Joshua B. Tenenbaum, and Brenden M. Lake. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In *NeurIPS*, pages 25192–25204, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/d3e2e8f631bd9336ed25b8162aef8782-Abstract.html.

57 Max Ostrowski and Torsten Schaub. ASP modulo CSP: the clingcon system. *Theory Pract. Log. Program.*, 12(4-5):485–503, 2012. doi:10.1017/S1471068412000142.

58 Jiao Ou, Junda Lu, Che Liu, Yihong Tang, Fuzheng Zhang, Di Zhang, and Kun Gai. DialogBench: Evaluating LLMs as human-like dialogue systems. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics:*

*Human Language Technologies (Volume 1: Long Papers)*, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.naacl-long.341.

59 Libo Qin, Qiguang Chen, Xiachong Feng, Yang Wu, Yongheng Zhang, Yinghui Li, Min Li, Wanxiang Che, and Philip S. Yu. Large language models meet nlp: A survey, 2024. doi:10.48550/arXiv.2405.12819.

60 Abhiramon Rajasekharan, Yankai Zeng, Parth Padalkar, and Gopal Gupta. Reliable natural language understanding with large language models and answer set programming. *Electronic Proceedings in Theoretical Computer Science*, 385:274–287, September 2023. doi:10.4204/eptcs.385.27.

61 Lin Ren, Guohui Xiao, Guilin Qi, Yishuai Geng, and Haohan Xue. Can llms solve asp problems? insights from a benchmarking study (extended version). *arXiv preprint arXiv:2507.19749*, 2025. doi:10.48550/arXiv.2507.19749.

62 Marcus Schaefer and Christopher Umans. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT news*, 33(3):32–49, 2002.

63 Peter Schüller. Modeling variations of first-order horn abduction in answer set programming. *Fundam. Informaticae*, 149(1-2):159–207, 2016. doi:10.3233/FI-2016-1446.

64 Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: A survey. *Theory Pract. Log. Program.*, 23(1):226–298, 2023. doi:10.1017/S1471068422000072.

65 Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976. doi:10.1016/0304-3975(76)90061-X.

66 Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. A systematic evaluation of the planning and scheduling abilities of the reasoning model o1. *Trans. Mach. Learn. Res.*, 2025, 2025. URL: https://openreview.net/forum?id=FkKBxp0FhR.

67 V. N. Vapnik. *Statistical learning theory*. Wiley, 1998.

68 V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In Vladimir Vovk, Harris Papadopoulos, and Alexander Gammerman, editors, *Measures of Complexity: Festschrift for Alexey Chervonenkis*, pages 11–30. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-21852-6_3.

69 Zhun Yang, Adam Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text. In *ACL (Findings)*, pages 5186–5219. Association for Computational Linguistics, 2023. doi:10.18653/V1/2023.FINDINGS-ACL.321.

70 Wayne Xin Zhao, Kun Zhou, and Junyi Li et al. A survey of large language models, 2025. arXiv:2303.18223.