Janhunen, Tomi; Niemelä, Ilkka

# The Answer Set Programming Paradigm

*Published in:*
AI Magazine

*DOI:*
10.1609/aimag.v37i3.2671

Published: 01/01/2016

*Document Version*
Peer reviewed version

*Please cite the original version:*
Janhunen, T., & Niemelä, I. (2016). The Answer Set Programming Paradigm. *AI Magazine*, *37*(3), 13-24. https://doi.org/10.1609/aimag.v37i3.2671

# The Answer Set Programming Paradigm

**Tomi Janhunen and Ilkka Niemelä**
Helsinki Institute for Information Technology HIIT
Aalto University School of Science
Department of Computer Science
PO Box 15400, FI-00076 Aalto, Finland

**Abstract**

In this paper, we give an overview of the answer set programming paradigm, explain its strengths, and illustrate its main features in terms of examples and an application problem.

## Introduction

*Answer set programming* (*ASP*, for short) is a declarative programming paradigm for solving search problems and their optimization variants. In ASP a search problem is modeled as a set of statements (a program) in a logic programming type of a language in such a way that the answer sets (models) of the program correspond to the solutions of the problem. The paradigm was first formulated in these terms by Marek and Truszczyński (1999) and Niemelä (1999). The ASP paradigm has its roots in knowledge representation and nonmonotonic logics research as described by Marek et al. (2011) in a historic account on the development of the paradigm. A more recent and more technical overview of ASP has been contributed by Brewka et al. (2011).

The ASP paradigm is most widely used with the formalism of logic programming under the semantics given by *answer sets* (Gelfond and Lifschitz 1988; 1990). The term *answer sets* was proposed by Gelfond and Lifschitz (1991) for sets of literals, by which programs in an extended syntax are to be interpreted where the classical *negation* operator and *disjunctions* of literals are allowed in the heads of program rules. Lifschitz' article (2016) in this special issue gives an introduction to the notion of an answer set and the language of ASP, as well as a comparison to Prolog systems. An alternative approach to ASP has been to use directly first-order logic as the basis and extend it with inductive definitions. The details can be found in the articles by Denecker and Vennekens (2014), Denecker and Ternovska (2008), East and Truszczyński (2006), and the one by Bruynooghe et al. (2016) in this special issue.

A main reason for the increasing interest in ASP is the availability of fast software tools that makes it possible to tackle problems of practical importance. Most of the current software tools employ two steps commonly referred to as *grounding* and *solving*, reflecting the definition of answer sets for programs with variables (Lifschitz 2016). The idea is to separate concerns so that the grounding phase takes care of the evaluation of more complicated data structures and variable instantiations using logic programming and deductive database techniques, and then the solving phase focuses on search for answer sets for a much simpler type of programs by employing advanced search methods. The papers by Kaufmann et al. (2016) and by Gebser and Schaub (2016) in this special issue provide more information on the solving and grounding techniques.

There is a growing number of successful applications of ASP including molecular biology (Gebser et al. 2010a; 2010b), decision support system for space shuttle controllers (Balduccini, Gelfond, and Nogueira 2006), phylogenetic inference (Erdem 2011; Koponen et al. 2015), product configuration (Soininen and Niemelä 1998; Finkel and O'Sullivan 2011) and repair of web-service work flows (Friedrich et al. 2010). Erdem et al. (2016) give an account of the applications of ASP in this special issue.

On the one hand, ASP is closely related to logic programming and Prolog and, on the other hand, to constraint programming (CP), propositional satisfiability (SAT), and linear/integer programming (LP/IP). Unlike Prolog-like logic programming ASP is *fully declarative* and neither the order of rules in a program nor the order of literals in the rules matter. Moreover, Prolog systems are tailored to find proofs or answer substitutions to individual queries whereas ASP systems are finding answer sets corresponding to complete solutions to a problem instance. The basic idea in ASP is very close to the paradigm of CP, SAT, or LP/IP where problems are represented by constraints and where systems are tailored to find satisfying variable assignments corresponding to complete solutions.

However, there are significant differences. The ASP paradigm allows for a very systematic approach to problem representation through *uniform encodings* where the problem statement can be developed independently of data on a particular instance. This leads to a large degree of *elaboration tolerance*. The ASP approach enables *structured representation* of problems where more complicated constraints are composed of simpler ones using rules. On the other hand, rules enable one to encode conditions that are challenging (like representing *disjunctive constraints* or other basic relational operations on constraints) or not available at all (like *recursive constraints*) when comparing to CP or

LP/IP paradigms. Because of these properties ASP allows for incremental development of an application and supports well *rapid prototyping*.

The goal of this paper is to provide an up-to-date overview of the ASP paradigm and illustrate its usage with examples as well as a more comprehensive application problem. We proceed as follows. In the next section, we explain the fundamental ideas of the ASP paradigm. The use of the paradigm and its main features are then illustrated by developing ASP encodings for an application problem step by step. The application considered in this paper is about designing a locking scheme for a building so that certain safety requirements are met. Having introduced the basic paradigm, we briefly address main ways to implement ASP—either using native answer-set solvers or translators enabling the use of solver technology from neighboring disciplines. The paper ends with a summary and discussion of future prospects. In addition, we illustrate the potential computational hardness of our application problem by explaining its connection to the NP-complete decision problem Exact-3-SAT.

## Basic ASP Paradigm

The conceptual model of the ASP paradigm is depicted in Figure 1. We start by explaining how to understand search problems at an abstract level and then illustrate how ASP is typically employed to solve such problems using the approach illustrated in the figure. Finally, we address a number of features and attractive properties of the paradigm.

**Problem Solving.** The ASP paradigm provides a general purpose methodology for solving search and optimization problems encountered in many real world applications. To get started, the key step is to identify and formalize the problem to be solved, i.e., to work out a **problem statement**. Typically this consists of clarifying what the potential **solutions** of the problem are like and then setting the conditions that solutions should satisfy. Solving the problem means that given the data on an instance of the problem we should find one or more solutions satisfying the given conditions (see the topmost arrow in Figure 1). For illustration, we use the task of finding a seating arrangement for a dinner as the first simple example. The respective problem statement could read as formulated below.

**Example 1 (Seating Arrangement Problem)**
*A certain group of people, say persons $p_1, \ldots, p_n$, are invited for dinner. There are tables $t_1, \ldots, t_k$ with the respective capacities $c_1, \ldots, c_k$ available for seating such that $c_1 + \cdots + c_k \geq n$. The host has some prior knowledge about the relationships of the guests: there are both friends and enemies among the invitees. This information should be taken into account when designing the arrangement. A solution to this problem is a mapping $s(p_i) = t_j$ of persons $p_i$ to tables $t_j$ so that the mutual relationships are respected.*

The problem statement above uses mathematical symbols to abstract the details of the problem such as the number and the identity of persons involved and the collection of tables available for seating. This reflects an important methodological feature, namely the separation of **instance data** from the actual problem statement. The point is that the problem can be stated without listing all details for a particular instance of the problem. In case of the seating arrangement problem, the instance data would consist of the names of invitees together with lists of tables and their capacities, and the pairs of persons who are known to be either friends or enemies. More concretely put, suppose that we have a group of 20 people: Alice, Bob, John, etc. There are four tables, seating 7, 6, 5, and 4 people, respectively. Moreover, we know that Alice likes Bob, Bob likes John and so on. Given all such pieces of information, the goal is

— to find at least one solution that fulfills the criteria set in the problem statement of Example 1, or

— to show that no solution exists.

Given what we know so far, we can expect solutions where Alice, Bob, and John are seated together at one of the four tables available. However, if we state additionally that Alice and John dislike each other, for instance, the seating problem instance under consideration has no solutions.

**ASP Encoding.** But how do we achieve the goal stated above using ASP and get the problem solved? As suggested by Figure 1, we should **formalize** the problem statement by writing down a (logic) **program**. Before we can really do this, we should have a basic understanding of syntax, also introduced in the article by Lifschitz (2016) in this issue. In ASP, programs consist of *rules*, i.e., statements of the form

```
head :- body1, body2, ..., bodyn.
```

The intuitive reading of the rule above is that the `head` can be inferred *if* (and only if) the body conditions `body1, body2, ..., bodyn` have been inferred by any other rules in the program. The conditions in the rule are either *atomic statements* (a.k.a. *atoms*) like `seat(a,1)` for Alice being seated at Table 1, or *count-bounded* sets of atoms

```
l { atom1; ...; atomk } u
```

where at least `l` but at most `u` atoms among `atom1, ..., atomk` should be inferable. The cardinality constraint above can also be expressed in terms of a *counting aggregate*

```
#count{atom1; ...; atomk}
```

where appropriate bounds can be incorporated using relation symbols <, <=, >, >=, and =. Atoms can also be negated using the operator `not` for *default negation*. A rule with an empty body (`n=0`) stands for a *fact* whose `head` holds unconditionally. As a further special case, a rule without a head stands for a *constraint* whose body `body1, body2, ..., bodyn` must not be satisfied. In this article, we do not consider extensions of rules by classical negation nor disjunctions in rule heads (Gelfond and Lifschitz 1991).

We are now ready to describe typical steps in writing down a program in ASP, resulting in an *encoding*[1] given as

[1]The encodings presented in this paper are directly executable using contemporary ASP grounders and solvers compatible with the ASP-core-2 language specification (Calimeri et al. 2012).

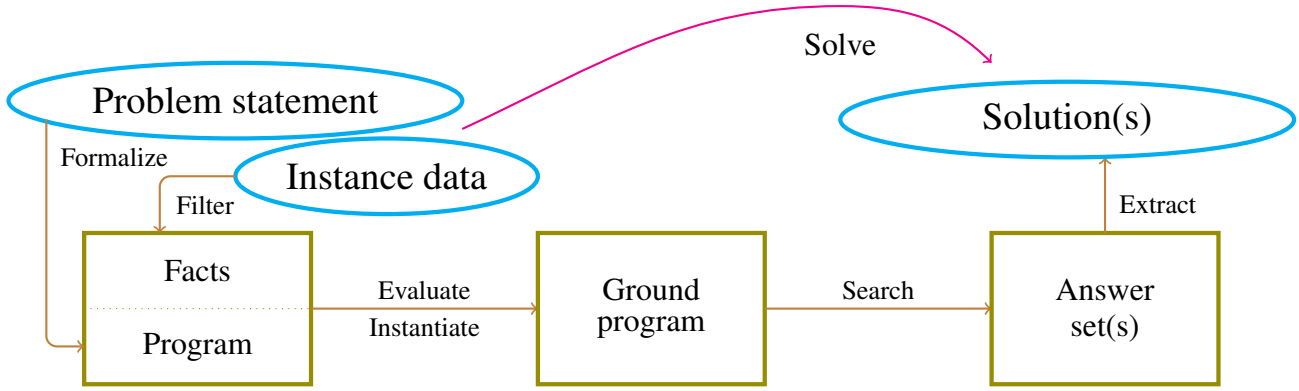


Figure 1: Conceptual Model of the ASP Paradigm

Listing 1: Encoding the Seating Problem in ASP

```
1  % Instance
2  person(a). person(b). person(j).
3  likes(a,b). likes(b,j). ...
4  dislikes(a,j). dislikes(j,a). ...
5  tbl(1,7). tbl(2,6). tbl(3,5). tbl(4,4).
6
7  % Rules and constraints
8  1 { seat(P,T): tbl(T,_) } 1 :- person(P).
9  :- #count{seat(P,T): person(P)}>C, tbl(T,C).
10 :- likes(P1,P2), seat(P1,T1), seat(P2,T2),
11    person(P1), person(P2),
12    tbl(T1,_), tbl(T2,_), T1 != T2.
13 :- dislikes(P1,P2), seat(P1,T), seat(P2,T),
14    person(P1), person(P2), tbl(T,_).
```

Listing 1. First, we have to decide how to represent the instance data. Sometimes this requires some form of **filtering** in order to identify which pieces of information are relevant in view of solving the problem. This is easy for the seating problem. The persons involved are listed in line 2 using predicate symbol `person/1` and constant symbols `a`, `b`, `j`, ... as abbreviations for the names of persons in question. Predicates `likes/2` and `dislikes/2` are used in lines 3–4 to represent (potentially incomplete)[2] information concerning friendship and dislike, respectively. Finally, the identities and capacities of tables are declared by the facts listed in line 5 using predicate `tbl/2`. Overall, we have obtained a set of **facts** as the representation of instance data.

The second step concerns the actual program formalizing the problem statement. Writing down the rules is of course a creative activity, which one learns best by doing, but in ASP one can concentrate on defining the relevant concepts (relations) in terms of rules, as well as thinking about conditions on which certain relations should hold. To understand the outcome of the formalization in Listing 1, let us give the intuitive readings for the rules involved. The rule in line 8 stipulates that every person `P` must be seated at exactly one table `T`. A few constraints follow. The capacities of tables are enforced in line 9: it is unacceptable if more than `C` persons are seated at table `T` which seats at most `C` persons. Moreover, if person `P1` likes person `P2`, they should not be seated at different tables `T1` and `T2`. This constraint is expressed in lines 10–12. The other way around, if `P1` does not like `P2`, then they should not be seated at the same table `T`. The respective rule is given in lines 13–14. The rules and constraints in lines 8–14 explained so far form a *uniform* encoding of the seating problem, as the representation is independent of any problem instance described by facts of the type in lines 2–5.

So far, we have demonstrated the modeling philosophy of ASP in terms of a simple application. The later section on locking design provides further insights into modeling and typical design decisions made. Yet further information is available in the articles of Bruynooghe et al. (2016) and Gebser and Schaub (2016) in this special issue.

**ASP Solving.** It remains to explain how the encoding from Listing 1 solves the problem instance in practice. First, the rules of the program have to be **instantiated** and **evaluated** with respect to the present facts. This means, e.g., that the rule in line 8 yields an instance

```
1 { seat(a,1); seat(a,2);
    seat(a,3); seat(a,4) } 1.
```

when `P` is replaced by `a` and `T` ranges over the available tables `1`, `2`, `3`, and `4`. This particular instance concerns the seating of Alice. While instantiating the rules also some evaluations take place. For example, when handling the rule in line 9 for table `1` with capacity `7` the lower bound `C` of the constraint is substituted by the value `7`. The **ground program**, also indicated in Figure 1, is typically generated by running a dedicated tool, i.e., a *grounder*, on the input. After that the **search** for **answer sets** can be performed by invoking an answer set solver. Finally, the solution(s) of the original problem instance are obtained by **extracting** relevant part(s) from the answer set(s) found. For the encoding under consideration, this means that whenever an occurrence of `seat(P,T)` is contained in an answer set, then person `P`

[2]However, ASP builds on the *closed world assumption* (CWA): the given information is treated as complete information and the problem is solved under this assumption.

is supposed to be seated at table T. Using the notions from Example 1, we would have the required mapping $s : \mathtt{P} \mapsto \mathtt{T}$ from persons to tables. If no answer set can be found, then a problem instance has no solutions. This is actually the case for the instance described by lines 2–5 in Listing 1, since it is impossible to place Alice, Bob, and John at the same table due to their relations. However, if the facts in line 4 are removed, obtaining answer sets is still feasible—the relationships of other guests permitting.

**Beyond Basic ASP.** The basic paradigm illustrated in Figure 1 solves the problem at hand by eventually finding one or more solutions to the problem, or by showing that no solution exists. If there are multiple solutions to the problem, then it may be desirable to select the best solution among the alternatives using some criterion such as *price*, *capacity*, etc. This turns the problem into an *optimization problem*. In ASP, objective functions for such problems can be defined in terms of *optimization statements* like

$$\texttt{\#minimize}\{\,\mathtt{w_1,1:atom_1;}\ \ldots;\ \mathtt{w_n,n:atom_n}\,\}.$$

The statement above assigns weights $\mathtt{w_1}, \ldots, \mathtt{w_n}$ to atoms $\mathtt{atom_1}, \ldots, \mathtt{atom_n}$, respectively, and the goal is to minimize the sum of weights for atoms contained in an answer set—when evaluated over all answer sets. As regards the seating arrangement problem, the respective optimization problem could deal with obviously inconsistent settings like the one described above. Rather than satisfying all constraints resulting from the mutual relations of persons, the goal would be to satisfy as many as possible. In the preceding example, this would mean that either Alice is seated at the same table as Bob, or Bob is seated with John, but Alice and John are placed at different tables.

Besides the optimization of solutions, there are also other *reasoning modes* of interest. It is sometimes interesting to see how much the solutions are alike. In *cautions reasoning*, the idea is to check whether a certain atom is present in all or absent from some answer set. For instance, if `seat(a,1)` is for some reason contained in all answer sets, then Alice will be unconditionally seated at the first table and no options remain to this end. Cautious reasoning corresponds to basic *query evaluation* over answer sets and it can be implemented by adding a constraint to the program. In the case of our example, the constraint would read `:- seat(a,1).` indicating that we would like to find any *counter-example*, i.e., an answer set *not* containing `seat(a,1)`. Alternatively, cautious reasoning can be implemented by solvers as a special reasoning mode while searching for answer sets. *Brave reasoning* is the dual of cautious reasoning and then the presence in some or absence from all answer sets is required. Again, this can be implemented by adding a constraint or as a special reasoning mode.

It is also possible to *enumerate* answer sets and, hence, *count* their number. For certain applications, the number of solutions could actually be an interesting piece of information. In product configuration (see, e.g., (Soininen and Niemelä 1998)), this could be the number of variants that a production line should be able to produce. There are also complex use cases of ASP. In *incremental solving*, the idea is to compute partial solutions to a problem (or show their non-existence) by calling an ASP solver several times and by extending the instance data on the fly. Various kinds of planning problems (with an increasing plan length) typically fall into this category. The latest developments even suggest *multi-shot solving* (Gebser et al. 2014) where solver calls are freely mixed and the ground programs used upon solver calls may evolve in more complex ways.

Listing 2: Examples of difference constraints

```
1 :- l(P)-e(P)<5, person(P).
2 :- l(P)-e(P)>90, person(P).
3 :- l(P1)-e(P2)>0, l(P2)-e(P1)>0,
4    dislikes(P1,P2), person(P1), person(P2),
5    seat(P1,T), seat(P2,T), tbl(T,_).
```

**Constraints over Infinite Domains.** Since grounding is an inherent part of ASP work flow, the basic paradigm is based on Boolean or finite-domain variables only. However, certain applications call for variables over infinite domains such as integers and reals. For instance, there have been proposals to extend ASP rules by *linear inequalities* (Gebser, Ostrowski, and Schaub 2009; Liu, Janhunen, and Niemelä 2012; Mellarkod, Gelfond, and Zhang 2008) as well as *difference constraints* (Janhunen, Liu, and Niemelä 2011). From the modeling perspective, the goal of such extensions is to increase the expressive power of ASP suitably so that new kinds of applications become feasible. For instance, referring back to the seating problem in Listing 1, we could refine the specification for each person `P` by introducing integer variables `e(P)` and `l(P)` denoting the points of time when `P` *enters* and *leaves* the table in question. Using difference constraints, we could state a specification given as Listing 2. Intuitively, the rules in lines 1 and 2 insist that person `P` stays at the table from `5` to `90` minutes. The constraint in lines 3–5 refines the last one from Listing 1. It is not allowed that any two persons `P1` and `P2` who dislike each other are seated at the same table at the same time. It is important to notice that when the constraint in line 1 is instantiated for Alice, the resulting constraint is `:- l(a)-e(a)<5.` Thus, the infinity of the underlying domain is not reflected to the size of the resulting ground program. Naturally, the interpretation of `l(a)` and `e(a)` as integer variables must be dealt with by the implementation of such constraints.

## Application: Locking Design

Having introduced the ASP paradigm on a general level, we now illustrate its main features in terms of an application problem where the goal is to design a locking scheme for a building. This is to be understood comprehensively, i.e., we are not just interested in locks but also anything else that can affect accessibility in a building. For simplicity, we consider a single floor. A sample floor plan of such a building is depicted in Figure 2. There are 12 rooms altogether, numbered from 1 to 12 in the figure. Given this domain, our objectives

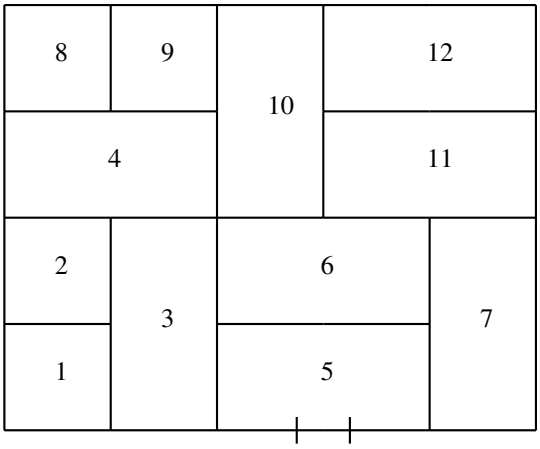


Figure 2: Floor plan for the rooms 1–12

are as follows. First, we describe the domain in a uniform way by selecting adequate predicates for the representation of domain information. Second, we take one concrete design goal from this domain into consideration. To this end, we concentrate on the configuration of locks installed on (potential) doors between the rooms in such a way that certain accessibility criteria are met. A particular safety requirement is that the floor can be effectively evacuated in case of an emergency. The idea is to develop ASP encodings for a design problem like this and, at the same time, illuminate the basic line of thinking and typical primitives used when modeling in ASP.

**Uniform Encoding.** The goal is to choose predicate symbols and the respective relations that are needed to represent an instance of the application problem at hand. To abstract the physical coordinates of the rooms, we rather represent the *adjacency relation* of rooms in terms of a predicate `adj/2`. For simplicity, we also assume that this relation captures the potential of installing doors between any adjacent rooms. The floor plan of Figure 2 can be represented by constants `1..12` for the rooms and the following facts:

```
adj(1,2).  adj(1,3).  adj(2,3).
adj(2,4).  ...        adj(11,12).
```

In total, there are 21 such facts and they are sufficient for the purposes of our examples to describe the interconnections of the rooms. For space efficiency, the adjacency information is represented asymmetrically, i.e., `adj(X,Y)` is reported only if `X<Y`. In addition, the rooms having exits are reported using a unary predicate `exit/1`. For the running example in Figure 2, this is captured by the fact `exit(5).` Now, if the given floor plan were changed in one way or another, or a completely different floor plan were taken into consideration, this should be reflected in the facts describing the problem instance. The other rules describing the application problem are based on these two predicates, hence making the encoding uniform. As typical in ASP encodings, some subsidiary domain predicates are defined in order to make the description of the actual problem easier. Some domain

Listing 3: Domain rules for locking design

```
1 room(R1) :- adj(R1,R2).
2 room(R2) :- adj(R1,R2).
3 pot(R1,R2) :- adj(R1,R2).
4 pot(R1,R2) :- adj(R2,R1).
5 otherexit :- exit(X), X>1.
6 exit(1) :- not otherexit.
```

rules for the locking design problem are collected in Listing 3 and explained below.

**Relational Operations.** The rules in lines 1–2 of Listing 3 are used to extract room information from the adjacency information by a simple *projection* operation. As a result `room(R)` is true for only those values of `R` that actually appear in the adjacency information. In principle, a door between two rooms provides symmetric access from a room to another. Thus, the adjacency relation is not well-suited as such for the description of accessibility and we form the *union* of the accessibility relation with its reverse relation using rules in lines 3–4. The relation `pot/2` stands for potential access depending on instrumentation such as locks, handles, press buttons, etc.

**Defaults.** To illustrate the use of defaults in encodings, we have included the rules in lines 5–6 of Listing 3. The rule in line 5 defines the condition `otherexit/0` meaning that some other room than the room `1` has an exit. The rule in line 6 ensures that, *by default*, there is an exit at room `1`. This is to hold unless another exit has been declared for the particular problem instance. There can be multiple exits. For instance, if there are two exits at rooms `1` and `5`, this can be stated explicitly using facts `exit(1)` and `exit(5)`. Adding these facts overrules the default in line 6 because `otherexit` can be inferred by the rule in line 5.

**Defining the Search Space.** Typical ASP encodings include a part where the solution candidates for the problem being formalized are generated. This can be achieved by expressing a number of *choices* that aim at capturing the varying aspects of solutions. As regards syntax, such choices can be expressed in terms of *choice rules* whose heads are count-bounded sets of atoms. Bounds can also be omitted if an arbitrary choice is of interest. As explained above, the access from a room to another can be asymmetric due to physical constructions. In particular, this is true for emergency situations where persons try to leave the building as soon as possible but might have no keys to unlock any door. For simplicity, we introduce a two-argument predicate `evac/2` that is used to express the existence of an evacuation route from a room to another. Given adjacent rooms `R1` and `R2`, such a design choice can be made in terms of a choice rule

```
{ evac(R1,R2) } :- pot(R1,R2).
```

The intuitive reading is that if `pot(R1,R2)` is true, then the truth value of `evac(R1,R2)` is subject to a *choice*. Hence, the selection of evacuation routes between rooms is formalized. Note that the analogous normal rule

Listing 4: ASP Encoding of the Evacuation Plan

```
1 reach(R,R) :- room(R).
2 reach(R1,R2) :-
3   reach(R1,R3), evac(R3,R2),
4   room(R1), pot(R3,R2).
5
6 ok(R) :- room(R), reach(R,X), exit(X).
7 :- not ok(R), room(R).
8
9 #minimize{1,R1,R2: evac(R1,R2), pot(R1,R2)}.
```

Listing 5: Revised ASP Encoding of the Evacuation Plan

```
1 step(0..s).
2
3 reach(R,R,0) :- room(R).
4 reach(R1,R2,S+1) :-
5   reach(R1,R3,S), evac(R3,R2),
6   room(R1), pot(R3,R2), step(S), step(S+1).
7
8 ok(R) :- room(R), reach(R,X,S),
9   exit(X), step(S).
```

```
evac(R1,R2) :- pot(R1,R2).
```

would falsify `evac(R1,R2)` by default if `pot(R1,R2)` were false, e.g., rooms `R1` and `R2` were not adjacent. Since the relation `pot/2` is symmetric, this gives rise to four different scenarios if `pot(R1,R2)` and thus also `pot(R2,R1)` is true. Evacuation in one direction is possible if either `evac(R1,R2)` or `evac(R2,R1)` holds. If they are both true, this allows for bidirectional evacuation between `R1` and `R2`. If such an option is not considered safe, it is easy to introduce an *integrity constraint* to exclude such a possibility in general:

```
:- evac(R1,R2), evac(R2,R1), pot(R1,R2).
```

If both `evac(R1,R2)` and `evac(R1,R2)` are false, then there is no connection between rooms `R1` and `R2` in case of an emergency. It remains to ensure that there exists an overall *evacuation plan*, i.e., it is possible to reach at least one exit of the building from every room.

**Recursive Definitions.** The existence of an evacuation plan is governed by constraints that concern the mutual reachability of rooms, to be formalized using a predicate `reach/2`. The first two rules of Listing 4 give a *recursive definition* for this predicate. Every room `R` is reachable from itself: the corresponding base case is given in line 1. The recursive case is formulated in lines 2–4: the reachability of `R2` from `R1` builds on the reachability of an intermediate room `R3` from `R1` and the condition that `R3` can be evacuated to `R2` (cf. line 3).

**Constraining Solutions.** The essential constraint on the evacuation plan is given in lines 6–7 of Listing 4. Any given room `R` is considered to be OK, if some exit `X` is reachable from it (line 6). The auxiliary predicate `ok/1` is defined in order to detect this aspect for each room. The actual constraint (line 7) excludes scenarios where some of the rooms would not be OK. Last, we want to minimize the number of evacuation connections by the objective function given in line 9. Using the encoding devised so far and an ASP solver, it is possible to check for the floor plan of Figure 2 that the minimum number of connections is 11. This is clear since there are 12 rooms in total each of which (except room `5`) must be connected to some other room for the purpose of evacuation. But ASP solvers can find out more for our running example. For instance, it is possible to enumerate and count all possible evacuation plans with 11 connections. In fact, there are 22 020 such plans and further constraints can be introduced to identify the most suitable ones. It is indeed the case that the current requirements allow for very long evacuation routes through the building of Figure 2 such as

$$7 \to 6 \to 11 \to 12 \to 10 \to 9 \to 8 \to 4 \to 2 \to 1 \to 3 \to 5.$$

Given this observation, the lengths of routes seem important. Thus, we now pay special attention to the number of evacuation *steps*, i.e., moves from a room to another, and from the room perspective. The number of steps ought to be limited.

**Elaboration Tolerance.** It is straightforward to modify the recursive encoding so that the number of steps is reflected. The revised encoding is presented as Listing 5. The domain for steps is first declared by the rule in line 1 where the maximum number of steps `s` is determined from the command line of the grounder. The base case in line 3 simply states that each room `R` is reachable from itself in zero steps. The main modification in the recursive case (lines 4–5) concerns counting: the number of steps `S` is increased by one to `S+1` whenever a further step is made. However, since both `S` and `S+1` must be members of the domain of steps, the maximum value is effectively determined by the constant `s` in line 1. Given the floor plan of Figure 2 and `s=2`, no evacuation plans can be found. By increasing `s` by one, solutions with 11 connections are found again and there are only 152 plans where the number of evacuation steps is at most three.

In summary, we have now tackled one particular aspect of locking design, i.e., ensuring that an evacuation plan exists for a building. In reality further requirements are imposed on evacuation plans making the problem computationally more and more challenging. For instance, it can be shown that if we incorporate conditions which can make rooms along an evacuation route mutually exclusive, e.g., for certain security reasons, it is unlikely that we are able to find a polynomial time algorithm for solving the problem (mathematically expressed the problem becomes NP-complete). This justifies well the use of powerful search methods like ASP for tackling the problem. For readers interested in computational complexity, we sketch the justifications of computational hardness in the sidebar.

## Computing Answer Sets

So far, we have concentrated on the conceptual model of Figure 1 with an emphasis on the modeling side. As regards the actual computation of answer sets, grounding and

solving were also identified as the main steps involved. Grounders are implemented either as stand-alone tools, such as the state-of-the-art grounder GRINGO[3], or integrated as a front-end of the solver. Native answer-set solvers are able to handle ground logic programs directly and, hence, truly implement the search step illustrated in the figure. Typically, this step is the most demanding one from the computational perspective. A number of answer-set solvers have been developed in the history of ASP and we mention here DLV[4], CLASP[3], and WASP[5] since they are actively maintained and developed at the moment. The article by Kaufmann et al. (2016) in this special issue gives a more detailed account of grounding and solving. If ASP is extended by constraints which cannot be directly handled by the ASP solver being used, the typical solution is to isolate extensions from rules themselves and to treat them by appropriate solvers externally. This leads to an architecture where two or more solvers are cooperating and interacting in analogy to SAT *modulo theories* (SMT) solvers. Then each sort of constraints can be handled by native algorithms.

**Translation-Based ASP.** The other constraint-based disciplines discussed in the introduction offer similar solver technology at the user's disposal for handling, in particular, the search phase. However, they cannot be used straightforwardly, as ground programs are not directly understood by such solvers and certain kinds of transformations become indispensable. The idea of *translation-based* ASP is to translate (ground) logic programs into other formalisms so that a variety of solvers can be harnessed to the task of computing answer sets. Such an approach can be understood as a refinement of the search step in Figure 1. There are existing translations from ASP, e.g., to SAT (Janhunen 2004), and its extension as SMT (Niemelä 2008), and *mixed integer programming* (MIP) (Liu, Janhunen, and Niemelä 2012). These translations indicate the *realizability* of ASP in other formalisms and they have all been implemented by translators in the ASPTOOLS[6] collection. They offer another way of implementing the search phase in ASP using off-the-shelf solvers as black boxes. This approach is already competitive in certain application problems and it can be seen as an effort to combine the expressive power of the modeling language offered by ASP with the high performance of existing solvers. Translations are also useful when implementing language extensions in a single target language. For instance, the idea of (Janhunen, Liu, and Niemelä 2011) is to translate programs enriched by difference constraints into difference logic altogether. The strength is that a single solver is sufficient for the search phase, but on the other hand, the original structure of constraints may be lost.

**Cross Translation.** The translations mentioned above are based on very similar technical ideas but yield representations of the ground program in completely different formats. Since the development of several translators brings about extra programming work, it would be highly desirable to integrate the variety of translators in a single tool—having options for different back-end formats. This is not as simple as that due to the wide variety of formats under consideration. However, this issue is partly solved by a recent translation from ASP to SAT *modulo acyclicity* (Gebser, Janhunen, and Rintanen 2014) where graph-based constraints are interconnected with ordinary logical constraints (i.e., clauses). The translation can be implemented by instrumenting a ground logic program with certain additional rules and meta information formalizing the underlying recursion mechanism in terms of the acyclicity constraint. This leads to a new implementation strategy for translation-based ASP: the choice of the target formalism can be postponed until the last step of translation where the constraints are output in a particular solver format. This idea is analogous to *cross compilation* in the context of compiling conventional programming languages and hence we coin the term *cross translation* for ASP. In the current implementation of this idea, a back-end translator transforms the instrumented program into other kinds of constraints understood by SMT, MIP, and *pseudo-Boolean* (PB) solvers, for instance. Interestingly, by implementing an additional acyclicity check inside a native ASP solver, the instrumented program can also be processed directly by the solver (Bomanson et al. 2015), which offers yet another approach to answer set computation.

## Summary and Future Prospects

This paper provides an introduction to the ASP paradigm as well as explains its main features—first generally, but also in terms of examples. We also discuss the two mainstream approaches to implementing the search for answer sets using either native solvers, or translators combined with solver technology offered by neighboring disciplines.

**Towards Universal Modeling.** There is a clear trend in the area of constraint-based modeling where methods and techniques are being transferred from one discipline to another. Various ideas from knowledge representation, logic programming, databases, and Boolean satisfiability served as a starting point for the ASP paradigm. But there are signs of knowledge transfer in the other direction as well. For instance, ASP solvers have been integrated into logic programming systems such as XSB (Rao et al. 1997). Advanced query evaluation mechanisms of ASP (Faber, Greco, and Leone 2007) are also relevant for deductive databases. The very idea of answer sets has been brought to the context of CP by introducing so-called *bound-founded* variables (Aziz, Chu, and Stuckey 2013). Quite recently, the algorithms for *projected answer set enumeration* have been exported for model counting in the context of SAT (Aziz et al. 2015).

We foresee that the exchange and incorporation of ideas and technologies in this way is gradually leading towards a universal approach where the user may rather freely pick the right language for expressing constraints of his or her interest. The underlying reasoning system is then supposed to (i) take care of required translations transparently and (ii) forward the resulting constraints for a solver architec-

[3] potassco.sourceforge.net/
[4] www.dlvsystem.com/
[5] github.com/alviano/wasp.git
[6] research.ics.aalto.fi/software/asp/

ture that can realize the search for answers. The first attempts to define a modular framework for multi-language modeling have already been made (Järvisalo et al. 2009; Lierler and Truszczyński 2014; Tasharrofi and Ternovska 2011). However, a lot of work remains to be done in order to realize the universal modeling scenario. Our experience from integrating various kinds of tools suggests that finding a universal format for the constraints of interest is one of the key issues for tool interoperability. There are existing formats such as the DIMACS format in SAT, the Smodels format in ASP, and the FlatZinc format in CP, that can be used as starting points for designing the universal format.

**Acknowledgments.** The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged. The authors thank Martin Gebser, Michael Gelfond, Torsten Schaub, and Mirek Truszczyński for their comments on a preliminary draft of this article.

## References

Aziz, R.; Chu, G.; Muise, C.; and Stuckey, P. 2015. #∃SAT: Projected model counting. In *Proceedings of SAT 2015*, 121–137.

Aziz, R.; Chu, G.; and Stuckey, P. 2013. Stable model semantics for founded bounds. *TPLP* 13(4-5):517–532.

Balduccini, M.; Gelfond, M.; and Nogueira, M. 2006. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence* 47(1-2):183–219.

Bomanson, J.; Gebser, M.; Janhunen, T.; Kaufmann, B.; and Schaub, T. 2015. Answer set programming modulo acyclicity. In *Proceedings of LPNMR 2015*, 143–150.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Bruynooghe, M.; Denecker, M.; and Truszczyński, M. 2016. First order logic with inductive definitions for model-based problem solving. *AI Magazine* (this number).

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Ricca, F.; and Schaub, T. 2012. ASP-CORE-2 input language format.

Denecker, M., and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.* 9(2).

Denecker, M., and Vennekens, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In *Proceedings of KR 2014*.

East, D., and Truszczyński, M. 2006. Predicate-calculus-based logics for modeling and solving search problems. *ACM Trans. Comput. Log.* 7(1):38–83.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine* (this number).

Erdem, E. 2011. Applications of answer set programming in phylogenetic systematics. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *LNCS*, 415–431.

Faber, W.; Greco, G.; and Leone, N. 2007. Magic sets and their application to data integration. *J. Comput. Syst. Sci.* 73(4):584–609.

Finkel, R., and O'Sullivan, B. 2011. Reasoning about conditional constraint specification problems and feature models. *AI EDAM* 25(2):163–174.

Friedrich, G.; Fugini, M.; Mussi, E.; Pernici, B.; and Tagni, G. 2010. Exception handling for repair in service-based processes. *IEEE Trans. Software Eng.* 36(2):198–215.

Gebser, M., and Schaub, T. 2016. Modeling and language extensions. *AI Magazine* (this number).

Gebser, M.; Guziolowski, C.; Ivanchev, M.; Schaub, T.; Siegel, A.; Thiele, S.; and Veber, P. 2010a. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proceedings of KR 2010*.

Gebser, M.; König, A.; Schaub, T.; Thiele, S.; and Veber, P. 2010b. The BioASP library: ASP solutions for systems biology. In *Proceedings of ICTAI 2010*, 383–389.

Gebser, M.; Kaminski, R.; Obermeier, P.; and Schaub, T. 2014. Ricochet robots reloaded: A case-study in multi-shot ASP solving. In *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation - Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, 17–32.

Gebser, M.; Janhunen, T.; and Rintanen, J. 2014. Answer set programming as SAT modulo acyclicity. In *Proceedings of ECAI 2014*, 351–356. IOS Press.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint answer set solving. In *Proceedings of ICLP 2009*, 235–249.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 6th International Conference on Logic Programming, ICLP'88*, 1070–1080.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In *Proceedings of ICLP'90*, 579–597.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9:365–385.

Janhunen, T.; Liu, G.; and Niemelä, I. 2011. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables, GTTV 2011*, 1–14.

Janhunen, T. 2004. Representing normal programs with clauses. In *Proceedings of ECAI'04*, 358–362.

Järvisalo, M.; Oikarinen, E.; Janhunen, T.; and Niemelä, T. 2009. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR 2009*, 155–168.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine* (this number).

Koponen, L.; Oikarinen, E.; Janhunen, T.; and Säilä, L.

2015. Optimizing phylogenetic supertrees using answer set programming. *TPLP* 15(4-5):604–619.

Lierler, Y., and Truszczyński, M. 2014. Abstract modular inference systems and solvers. In *Proceedings of PADL 2014*, 49–64.

Lifschitz, V. 2016. Answer sets and the language of answer set programming. *AI Magazine* (this number).

Liu, G.; Janhunen, T.; and Niemelä, I. 2012. Answer set programming via mixed integer programming. In *Proceedings of KR 2012*, 32–42.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer. 375–398.

Marek, V.; Niemelä, I.; and Truszczyński, M. 2011. Origins of answer-set programming - some background and two personal accounts. In *Nonmonotonic Reasoning: Essays Celebrating its 30th Anniversary*. College Publications. 233–258. Also available as CoRR `abs/1108.3281`.

Mellarkod, V.; Gelfond, M.; and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Math. and AI* 53(1-4):251–287.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Niemelä, I. 2008. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence* 53(1-4):313–329.

Papadimitriou, C. 1994. *Computational Complexity*. Addison-Wesley.

Rao, P.; Sagonas, K.; Swift, T.; Warren, D.; and Freire, J. 1997. XSB: A system for effciently computing WFS. In *Proceedings of LPNMR'97*, 431–441.

Soininen, T., and Niemelä, I. 1998. Developing a declarative rule language for applications in product configuration. In *Proceedings of PADL'99*, 305–319.

Tasharrofi, S., and Ternovska, E. 2011. A semantic account for modularity in multi-language modelling of search problems. In *Proceedings of FroCoS 2011*, 259–274.

## Sidebar: Locking Design Can Be Computationally Challenging

It is not surprising that finding a locking scheme satisfying given conditions can become computationally challenging when more involved conditions need to be satisfied. Here we consider the problem of finding a locking scheme that allows an evacuation plan such that for each room there is exactly one evacuation direction and the evacuation routes respect a given set of room conflicts, i.e., a set of pairs of rooms $(R_1, R_2)$ such that when following the evacuation routes if you enter room $R_1$, then you cannot enter room $R_2$. We show that this locking design problem is NP-complete indicating that it is unlikely that a polynomial time algorithm for solving this problem can be found. See, for example, (Papadimitriou 1994) for an introduction to computational complexity and the required concepts used below.

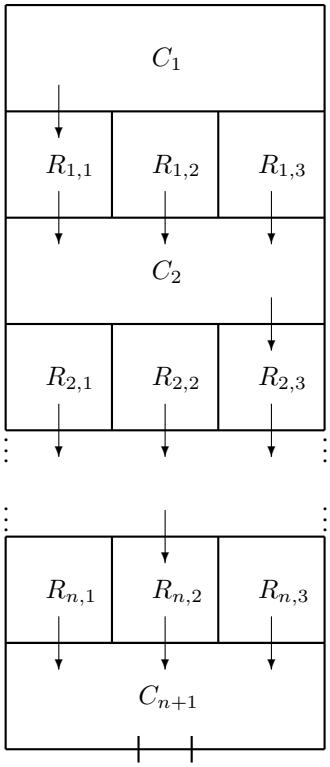


Figure 3: Floor Plan and Evacuation Routes for the NP Completeness Proof

Technically, the NP-completeness of a problem can be shown by establishing a reduction computable in polynomial time from a known NP-complete problem to the problem and showing that it can be checked in polynomial time that a potential solution satisfies the required conditions for the problem. As such a known NP-complete problem we use the Exact-3-SAT problem where we are given a conjunction of 3-literal clauses and the problem is to find a truth assignment that satisfies exactly one literal in each of the clauses.

**Reduction from Exact-3-SAT.** Any given 3-SAT instance $C_1 \wedge \ldots \wedge C_n$ can be transformed into a floor plan illustrated in Figure 3. For each 3-literal clause $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, we introduce a corridor $C_i$ connected to rooms $R_{i,1}$, $R_{i,2}$, and $R_{i,3}$ that are connected to corridor $C_{i+1}$. Moreover, rooms $R_{i,1}$, $R_{i,2}$, and $R_{i,3}$ do not have doors in-between. The (only) exit is located next to corridor $C_{n+1}$ which means that all corridors and rooms must be eventually evacuated through it. Moreover, each room $R_{i,j}$ is labeled by the respective literal $l_{i,j}$, the idea being that $l_{i,j}$ is satisfied if $C_i$ is evacuated via the room $R_{i,j}$. Consequently, if there are two rooms labeled by *complementary* literals (i.e., a Boolean variable $x$ and its negation $\neg x$), then those rooms are in conflict. This means that evacuation routes involving any pair of conflicting rooms are not feasible. It is also easy to see that the floor plan in Figure 3 and the associated set of conflicts can be computed in polynomial time.

It can be shown that a 3-SAT instance $C_1 \wedge \ldots \wedge C_n$ has

a satisfying truth assignment such that each clause has exactly one literal satisfied if and only if for the corresponding floor plan there is a locking scheme that allows an evacuation plan such that (i) for each room there is exactly one evacuation direction and (ii) the evacuation routes respect the set of room conflicts arising from the complementary literals. The key observation is that for the corresponding floor plan evacuation is possible only if there is a route from $C_1$ to $C_{n+1}$ such that for each $i = 1, \ldots, n$ the route visits exactly one of the rooms $R_{i,1}$, $R_{i,2}$, and $R_{i,3}$ and all room conflicts are respected. A satisfying truth assignment such that each clause has exactly one literal satisfied gives directly such a route and if such a route is available, it gives directly an appropriate truth assignment where literals corresponding to the visited rooms in the route are satisfied.

Moreover, it is clear that given a locking scheme with exactly one evacuation direction for each room, it can be checked in polynomial time that evacuation is possible and that all room conflicts are respected.