Modeling and Language Extensions

Martin Gebser
University of Potsdam, Germany
gebser@cs.uni-potsdam.de

Torsten Schaub[*]
University of Potsdam, Germany
INRIA Rennes, France
torsten@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP) has emerged as an approach to declarative problem solving based on the stable model semantics for logic programs. The basic idea is to represent a computational problem by a logic program, formulating constraints in terms of rules, such that its answer sets correspond to problem solutions. To this end, ASP combines an expressive language for high-level modeling with powerful low-level reasoning capacities, provided by off-the-shelf tools. Compact problem representations take advantage of genuine modeling features of ASP, including (first-order) variables, negation by default, and recursion. In this article, we demonstrate the ASP methodology on two example scenarios, illustrating basic as well as advanced modeling and solving concepts. We also discuss mechanisms to represent and implement extended kinds of preferences and optimization. An overview of further available extensions concludes the article.

Introduction

Answer Set Programming (ASP; (Brewka, Eiter, and Truszczyński 2011)) is a paradigm of declarative problem solving with roots in knowledge representation, logic programming, constraint satisfaction and optimization. Formally, ASP is based on the stable model semantics for logic programs (Gelfond and Lifschitz 1991), detailed by Lifschitz (2016) in this issue. As illustrated by Janhunen and Niemelä (2016), also in this issue, logic programs can be used to compactly represent search and optimization problems within the first two levels of the polynomial time hierarchy (Faber, Pfeifer, and Leone 2011; Ferraris 2011).[1] On the one hand, the attractiveness of ASP is due to an expressive modeling language, where concepts like (first-order) variables, negation by default, and recursion enable uniform problem representations in terms of facts specifying an instance along with a general problem encoding (Schlipf 1995). On the other hand, powerful ASP systems, described by Kaufmann et al. (2016) in this issue, are available off-the-shelf and automate the grounding of an encoding relative to a problem instance as well as the search for answer sets corresponding to problem solutions.

In this article, we detail the ASP modeling methodology on two example scenarios. To begin with, we elaborate on the use of traditional one-shot solving, where a problem is tackled by means of singular grounding and search processes. We particularly focus on a conceptual Generate-and-Test pattern (Eiter, Ianni, and Krennwallner 2009; Leone et al. 2006; Lifschitz 2002) as a best practice method to conceive legible yet efficient problem encodings. Further information regarding, amongst others, tool support for logic program development, elaboration tolerant ways to represent extensive application domains, and alternative modeling languages is provided by Lierler, Maratea, and Ricca (2016), Erdem, Gelfond, and Leone (2016), as well as Bruynooghe, Denecker, and Truszczyński (2016) in this issue.

In our second example scenario, we take advantage of multi-shot solving, a powerful extension of traditional ASP methods in which grounding and search are interleaved to process a series of evolving subtasks in an iterative manner. Rather than processing each subtask from scratch, multi-shot solving gradually expands the representation of a problem, where grounding instantiates novel problem parts and search can reuse conflict information. Such incremental reasoning fits the needs in dynamic domains like, e.g., logistics, policies, or robotics. In particular, we address a planning problem, where the (minimal) number of actions required to achieve a goal is usually not known a priori, while theoretical limits are prohibitively high as regards grounding.

The presentation of the two main approaches to modeling and solving is complemented by a survey of mechanisms to represent and implement extended kinds of preferences and optimization. An overview of further extensions conceived for demanding application problems concludes the article.

Modeling the Traveling Salesperson Problem

For illustrating the principal modeling concepts in ASP, let us consider the well-known *Traveling Salesperson Problem* (TSP). A TSP instance consists of a number of places, each of which must be included within a round trip visiting every place exactly once, as well as links between places, specifying potential successors along with associated costs.

For example, Figure 1 displays an instance with six places: the German cities Berlin, Dresden, Hamburg,

[*]Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

[1]See, e.g., (Papadimitriou 1994) for an introduction to computational complexity.

Figure 1: Places connected by links associated with costs

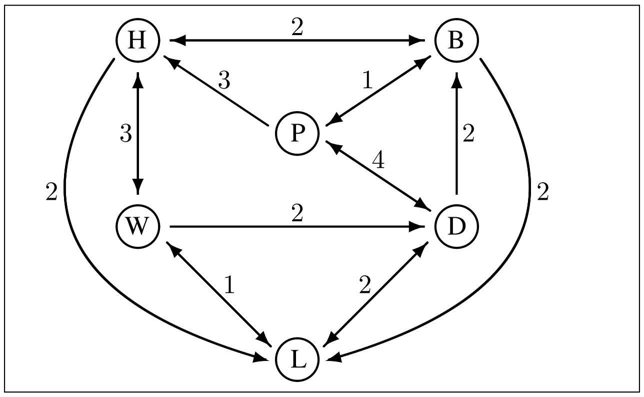
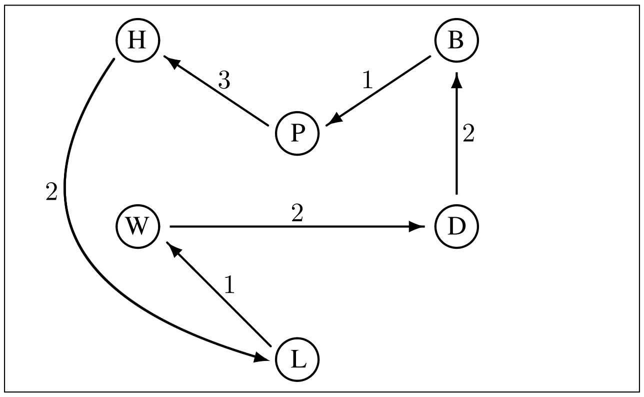


Figure 2: The shortest round trip for the places in Figure 1



Listing 1: Instance specifying the places in Figure 1 as facts

```
place(b). % Berlin
place(d). % Dresden
place(h). % Hamburg
place(l). % Leipzig
place(p). % Potsdam
place(w). % Wolfsburg
link(b,h,2). link(b,l,2). link(b,p,1).
link(d,b,2). link(d,l,2). link(d,p,4).
link(h,b,2). link(h,l,2). link(h,w,3).
link(l,d,2). link(l,w,1).
link(p,b,1). link(p,d,4). link(p,h,3).
link(w,d,2). link(w,h,3). link(w,l,1).
```

Leipzig, Potsdam, and Wolfsburg, each denoted by its first letter. The cities are linked by train connections, available in either one or both directions, and their respective durations in hours constitute the costs. E.g., Berlin and Potsdam are mutually linked, and it takes one hour to travel between the two neighboring cities, while four hours are needed from Potsdam to Dresden or vice versa. Moreover, a train connects Potsdam to Hamburg within three hours, but it does not operate the other way round. Further train connections interlink the other cities, and the question is how to arrange a shortest round trip visiting all cities.

When we construct a round trip manually, we may first fix some place to start the trip from, say Potsdam, and then proceed by opportunistically picking links to yet unvisited cities. E.g., we can find a round trip leading from Potsdam to Berlin, Hamburg, Leipzig, Wolfsburg, Dresden, and then back to Potsdam. The connections taken in this trip add up to a total duration of $1+2+2+1+2+4 = 12$ hours. However, the true shortest round trip shown in Figure 2 takes 11 hours only. To find such shortest round trips, also in case the train connections or cities to visit change, we aim at a general method for arbitrary places and links between them.[2] In what follows, we thus apply the ASP methodology to model shortest round trips for any TSP instance provided as input.

[2]Computing a shortest round trip is $\mathrm{FP}^{\mathrm{NP}}$-complete (Papadimitriou 1994); that is, it can be accomplished by means of a polynomial number of queries to an NP-oracle.

Problem Instance

The common practice in ASP is to represent a problem at hand *uniformly*, distinguishing between a particular instance and a general encoding (Marek and Truszczyński 1999; Niemelä 1999; Schlipf 1995). That is, we first need to fix a logical format for specifying places and links with associated costs. For example, the cities and connections displayed in Figure 1 are described in terms of the *facts* given in Listing 1. These facts are based on two predicates, `place`/1 and `link`/3, where 1 and 3 denote the arities of respective relations. The letters used as arguments of facts over `place`/1 stand for corresponding cities, e.g., *p* refers to Potsdam, and writing such constants in lower case follows logic programming conventions. Moreover, facts over `link`/3 specify the available connections, e.g., `link(p,b,1)`, `link(p,d,4)`, and `link(p,h,3)` provide those from Potsdam to Berlin, Dresden, and Hamburg along with their associated durations, as displayed in Figure 1. The durations are given by integers, on which ASP systems support arithmetic operations,[3] while the names used for cities and predicates have no particular meaning beyond identifying places or relations, respectively. Also note that the facts constitute a set, so that the order of writing them is immaterial, which distinguishes ASP from logic programming languages having a procedural flavor, such as Prolog.

Problem Encoding

The main modeling task consists of specifying the intended outcomes, viz. shortest round trips, in terms of the conditions they must fulfill. To this end, let us first formulate such requirements in natural language:

(a) Every place is linked to exactly one successor in a trip.

(b) Starting from an arbitrary place, a trip visits all places and then returns to its starting point.

(c) The sum of costs associated with the links in a trip ought to be minimal.

While these conditions are sufficient to characterize shortest round trips, the requirement in (b) further implies that every place has some predecessor. Given that (a) limits the number

[3]Extensions to real numbers are presented in (Bartholomew and Lee 2013; Liu, Janhunen, and Niemelä 2012).

of links in a round trip to the number of places, the following condition must hold as well:

(d) Every place is linked to exactly one predecessor in a trip.

In summary, trips meeting the requirements in (a) and (b) are subject to the optimality criterion in (c), and (d) expresses an implied property. The conditions at hand provide a mental model for the ASP encoding furnished in the following.

The *encoding* shown in Listing 2 is based on a conceptual Generate-and-Test pattern (Eiter, Ianni, and Krennwallner 2009; Leone et al. 2006; Lifschitz 2002). Accordingly, it is structured into several parts, distinguished by their concerns as well as typical constructs among those presented by Lifschitz (2016) in this issue. The purposes of the parts indicated by comments in lines beginning with '`%`' are:

- A `DOMAIN` part specifies auxiliary concepts that can be derived from facts and are shared by all answer sets.
- A `GENERATE` part includes non-deterministic constructs, usually choice or disjunctive rules, to provide solution candidates.
- A `DEFINE` part characterizes relevant properties of solution candidates, where the inherent features of fixpoint constructions and negation by default suppress "false positives" and enable a compact representation.
- A `TEST` part usually consists of integrity constraints that deny invalid candidates whose properties do not match the requirements on solutions.
- An `OPTIMIZE` part makes use of optimization statements or weak constraints to associate solutions with costs subject to minimization.
- A `DISPLAY` part declares output predicates to which the printing of answer sets ought to be restricted in order to make reading off solutions more convenient.[4]

In what follows, we elaborate on respective encoding parts.

DOMAIN: The first part, denoted by `DOMAIN`, includes the *rule* in Line 2 to determine the lexicographically smallest identifier among places in an instance as (arbitrary) starting point for the construction of a round trip. To this end, the identifiers given by facts over `place`/1 are taken as values for the variable `Y`, and the smallest value, selected via a **`#min`** aggregate, is used to instantiate the variable `X` recurring in the head `start(X)`. Note that, as usual in logic programming, variable names begin with upper-case letters, and recurrences within the same scope, i.e., a rule, are substituted with common values. Relative to the facts in Listing 1, the rule in Line 2 is thus instantiated to

```
start(b) :- b = #min{
  b : place(b); d : place(d); h : place(h);
  p : place(p); l : place(l); w : place(w)}.
```

Since the predicate `place`/1 is entirely determined by facts, the above rule can be simplified to a derived fact `start(b).`

[4]Apart from a system-specific **`#show`** directive for output projection, the encoding in Listing 2 is written in the syntax of the ASP-Core-2 standard language (`https://www.mat.unical.it/aspcomp2013/ASPStandardization/`).

Listing 2: Encoding of round trips w.r.t. facts as in Listing 1

```
 1 % DOMAIN
 2 start(X) :- X = #min{Y : place(Y)}.
 3 % GENERATE
 4 {travel(X,Y) : link(X,Y,C)} = 1 :- place(X).
 5 % DEFINE
 6 visit(X) :- start(X).
 7 visit(Y) :- visit(X), travel(X,Y).
 8 % TEST
 9 :- place(Y), not visit(Y).
10 :- start(Y), #count{X : travel(X,Y)} < 1.
11 :- place(Y), #count{X : travel(X,Y)} > 1.
12 % OPTIMIZE
13 :~ travel(X,Y), link(X,Y,C). [C,X]
14 % DISPLAY
15 #show travel/2.
```

In general, a `DOMAIN` part contains deterministic rules specifying relevant auxiliary concepts, so that they do not need to be provided per instance in a redundant fashion. Rather, including such rules in an encoding increases elaboration tolerance and exploits the capabilities of common grounders, which evaluate deterministic parts.

GENERATE: The second part, indicated by `GENERATE`, gathers non-deterministic constructs such that alternative selections among the derivable atoms provide distinct solution candidates. In Line 4, we use a *choice* rule (Simons, Niemelä, and Soininen 2002) to express that, for every place in an instance, exactly one link from the place must be picked for a round trip. The rule constitutes a schema that applies to each place identifier taken as value for the variable `X`. E.g., considering Potsdam and the three connections from there, it yields

```
{travel(p,b) : link(p,b,1);
 travel(p,d) : link(p,d,4);
 travel(p,h) : link(p,h,3)} = 1 :- place(p).
```

Further simplifying this rule in view of facts over `place`/1 and `link`/3 leads to

```
{travel(p,b); travel(p,d); travel(p,h)} = 1.
```

That is, any answer set must include exactly one of the options `travel(p,b)`, `travel(p,d)`, and `travel(p,h)`, reflecting that either Berlin, Dresden, or Hamburg has to succeed Potsdam in a round trip. As the same schema applies to other cities as well, atoms over the predicate `travel`/2 in an answer set represent a trip meeting the requirement in (a). However, the rule in Line 4 leaves open which successor per place shall be picked, and hence it is called choice rule.

DEFINE: While the predicate `travel`/2 provides sufficient information to reconstruct a trip from an answer set, the requirement in (b) that all places must be visited is yet unaddressed. In order to test this condition, the `DEFINE` part includes rules analyzing which places are visited from the starting point fixed in the `DOMAIN` part before. To begin with, the rule in Line 6 derives the starting point as visited, e.g., `visit(b)` follows from `start(b)` relative to the facts in Listing 1. The rule in Line 7 further collects places reachable via the connections indicated by `travel`/2. For example, the following derivation chain is activated by atoms over `travel`/2 that represent the connections shown in Figure 2:

```
visit(b) :- start(b).
visit(p) :- visit(b), travel(b,p).
visit(h) :- visit(p), travel(p,h).
visit(l) :- visit(h), travel(h,l).
visit(w) :- visit(l), travel(l,w).
visit(d) :- visit(w), travel(w,d).
```

Given that the involved connections form a round trip, all atoms over `visit`/1 follow via a sequence of rules rooted in `start(b)`. However, if Hamburg were linked to Berlin instead of Leipzig, no such sequence would yield `visit(l)`, `visit(w)`, and `visit(d)`. Atoms lacking a non-circular derivation are *unfounded* and exempt from answer sets (Van Gelder, Ross, and Schlipf 1991). In turn, answer sets encompass fixpoint constructions for expressing concepts like, e.g., induction and recursion. A `DEFINE` part makes use of this to derive predicates indicating relevant properties of a solution candidate at hand. As in `DOMAIN` parts, the contained rules are deterministic, yet their evaluation relies on non-deterministically generated solution candidates. In our case, `visit`/1 provides all places reached by taking connections in the trip from a fixed starting point.

`TEST`: The predicates characterizing solution candidates as well as their relevant properties are inspected in the `TEST` part in order to eliminate invalid candidates. This is accomplished by means of *integrity constraints*, i.e., rules of denial with an implicitly false head, written by leaving the left hand side of '`:-`' blank. Regarding the conditions for round trips, the `GENERATE` part already takes care of (a), while the requirement in (b) remains to be checked. To this end, the integrity constraint in Line 9 expresses that all places must be visited from the starting point given by `start`/1. E.g., if Leipzig were not reached, a contradiction would be indicated via

```
:- place(l), not visit(l).
```

Note that '`not visit(l)`' makes use of negation by default, which applies whenever `visit(l)` is unfounded. Importantly, negation by default does not offer any derivation (by contraposition). As a consequence, the above integrity constraint is not interchangeable with a rule like

```
visit(l) :- place(l).
```

If given such a rule, we could simply conclude `visit(l)`, regardless of reachability. Unlike that, integrity constraints do not modify solution candidates or predicates providing their properties, but merely deny unintended outcomes. The distinction between constructs for deriving and evaluating atoms is an important modeling concept, here used to check that all places are indeed reached from a fixed starting point.

For the requirement in (b), we still have to make sure that a trip at hand returns to its starting point. Since every place is linked to one successor only and all but one final connection are needed to visit places different from the starting point given by `start`/1, it is sufficient to check that a (final) connection returning to the starting point exists. This condition is imposed by the integrity constraint in Line 10, and relative to the facts in Listing 1 it is instantiated to

```
:- start(b), #count{d : travel(d,b);
   h : travel(h,b); p : travel(p,b)} < 1.
```

The `#count` aggregate provides the number of atoms among `travel(d,b)`, `travel(h,b)`, and `travel(p,b)`, representing connections returning to Berlin, included in an answer set. If neither connection is taken, this number is zero, in which case the success of the '`< 1`' comparison indicates a contradiction. In turn, some connection must lead back to Berlin, but it can only be taken once all places are visited.

The checks via the integrity constraints in Line 9 and 10 establish that answer sets represent round trips meeting the requirement in (b). Since (a) is handled in the `GENERATE` part, the rules up to Line 10 are already sufficient to characterize round trips. However, the implied property in (d) also states that a place cannot be linked to several predecessors. While this condition may seem apparent to humans, it relies on a counting argument taking the number of connections in a trip and the necessity that every place must be linked to some predecessor into account. Given that ASP solvers do not apply such reasoning, it can be beneficial to formulate non-trivial implied properties as *redundant constraints*. This is the motivation to include the integrity constraint in Line 11, making explicit that a place cannot be linked to several predecessors. E.g., regarding connections leading to Berlin, the schema yields

```
:- place(b), #count{d : travel(d,b);
   h : travel(h,b); p : travel(p,b)} > 1.
```

In view of the '`> 1`' comparison relative to the `#count` aggregate, a contradiction is indicated as soon as connections from two cities among Dresden, Hamburg, and Potsdam to Berlin are picked for a round trip. Respective restrictions to a single predecessor apply to cities other than Berlin as well.

`OPTIMIZE`: After specifying solution candidates and requirements, the `OPTIMIZE` part addresses the optimality criterion in (c). To this end, the *weak constraint* in Line 13 associates every place with the cost of the link to its successor in a round trip.[5] Focusing on the three connections from Berlin, we obtain

```
:~ travel(b,h), link(b,h,2). [2,b]
:~ travel(b,l), link(b,l,2). [2,b]
:~ travel(b,p), link(b,p,1). [1,b]
```

Weak constraints resemble integrity constraints, but rather than eliminating solution candidates to which the expressed conditions apply, the lists enclosed in square brackets are gathered in a set. The sum of integers included as their first elements constitutes the total cost associated with an answer

[5]The weak constraint corresponds to the optimization statement `#minimize{C,X : travel(X,Y), link(X,Y,C)}.`

Listing 3: `clingo` run on facts and encoding in Listing 1-2

```
$ clingo tsp-ins.lp tsp-enc.lp

Answer: 1
  travel(b,l) travel(l,w) travel(w,d)
  travel(d,p) travel(p,h) travel(h,b)
Optimization: 14

Answer: 2
  travel(b,p) travel(p,h) travel(h,w)
  travel(w,l) travel(l,d) travel(d,b)
Optimization: 12

Answer: 3
  travel(b,p) travel(p,h) travel(h,l)
  travel(l,w) travel(w,d) travel(d,b)
Optimization: 11

OPTIMUM FOUND
```

set and is subject to minimization. Regarding connections from Berlin, the fraction of the total cost is either 1 for Potsdam or 2 in case of Hamburg and Leipzig. Given that Hamburg and Leipzig cannot both succeed Berlin in a round trip, there is no urge to keep their respective lists distinct, e.g., by adding the identifier *h* or *l* as an element. By reusing the same list instead, we actually reduce the number of factors taken into account in the total cost calculation, which can in turn benefit the performance of ASP solvers.[6]

DISPLAY: The final part, denoted by `DISPLAY`, includes the **#show** directive in Line 15, declaring `travel`/2 as output predicate. This does not affect the meaning of the encoding, but instructs systems like `clingo` (Gebser et al. 2014) to restrict the printing of answer sets to atoms over `travel`/2. Indeed, facts as well as places given by `start`/1 and `visit`/1 are predetermined by an instance, and only the connections provided by `travel`/2 characterize a particular round trip.

Solution Computation

Assuming that the facts in Listing 1 and the encoding in Listing 2 are stored in text files called `tsp-ins.lp` and `tsp-enc.lp`, the output of a `clingo` run is given in Listing 3. We see that `clingo` finds three round trips of decreasing cost, listed in lines beginning with ‘`Optimization:`’ below the atoms over `travel`/2 in a corresponding answer set. While the first round trip is arbitrary and merely depends on heuristic aspects of the search in `clingo`, the second must be of smaller cost, and likewise the third. The latter cannot be improved any further, indicated by ‘`OPTIMUM FOUND`’ in the last line, as it represents the shortest round trip shown in Figure 2. For the instance at hand, this is the only round trip of cost 11, and some arbitrary witness among all optimal answer sets is determined in general.[7] Non-determinisms, such as the (optimal) answer set found, are thus left up to the search in an ASP solver, while an encoding merely specifies requirements on intended outcomes. This distinguishes ASP from traditional logic programming languages like Prolog, in which programs have a procedural semantics based on the order of writing rules.

[6]An even more elaborate penalization scheme based on relative cost differences is presented in (Gebser et al. 2012, Section 8.3).

Summary

While the well-known TSP is conceptually simple, it gives room for exploring diverse modeling concepts and designs. Let us recap the main principles of the above ASP method:

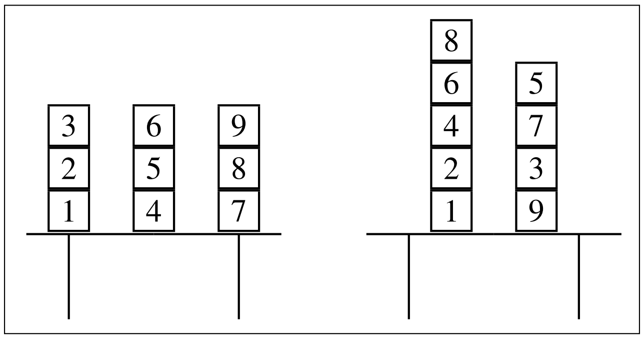
- A uniform problem representation separates facts describing an instance from a general problem encoding. The latter consists of schemata, expressed in terms of variables, that specify solutions for any problem instance. Such high-level modeling is crucial for elaboration tolerance, meaning that changes in a problem specification can be addressed by modest modifications of the representation. For example, when round trips shall be approximated for instances that have no solution otherwise, the integrity constraint requiring all places to be visited can easily be turned into a weak constraint for admitting exceptions.
- An ASP encoding is usually structured into parts addressing different concerns in a Generate-and-Test conception. The key parts, nicknamed `GENERATE`, `DEFINE`, `TEST`, and `OPTIMIZE`, provide solution candidates, analyze their relevant properties, eliminate invalid candidates, and evaluate solution quality.
- The typical constructs used within `GENERATE`, `DEFINE`, `TEST`, and `OPTIMIZE` parts are non-deterministic (choice) rules, deterministic rules, integrity constraints, or weak constraints, respectively. Deterministic rules make use of the expressivity of answer sets encompassing fixpoints, induction, and recursion. While `TEST` parts should typically stay compact regarding sufficient conditions, redundant constraints expressing non-trivial implied properties can benefit the search in an ASP solver. Weak constraints in an `OPTIMIZE` part can be made more effective by reducing the number of factors taken into account to evaluate solution quality.
- An ASP encoding merely specifies requirements, but not how answer sets representing (optimal) solutions shall be computed. While admissible outcomes are fixed by the semantics, the way to find them is left up to ASP solvers.

Modeling the Blocks World Planning Problem

Beyond traditional one-shot solving, where a problem instance is fed to an isolated search process, multi-shot solving addresses series of evolving subtasks in an iterative manner. This is of interest in dynamic domains, such as logistics, policies, or robotics, dealing with recurrent tasks in a changing environment. To illustrate respective scenarios, we con-

[7]Optimal answer sets can be enumerated using dedicated reasoning modes of `clingo` (Gebser et al. 2015).

Figure 3: Initial and goal situation for blocks world planning



Listing 4: Instance specifying situations in Figure 3 as facts

```
init(3,2). init(6,5). init(9,8).
init(2,1). init(5,4). init(8,7).
init(1,0). init(4,0). init(7,0). table(0).
goal(8,6).
goal(6,4). goal(5,7).
goal(4,2). goal(7,3).
goal(2,1). goal(3,9).
goal(1,0). goal(9,0).
```

sider *Blocks World Planning* (Slaney and Thiébaux 2001), where blocks must be restacked on a table to bring them from their initial positions into a goal configuration.

Figure 3 displays an example scenario with nine blocks. In the initial situation, shown on the left, the blocks are arranged in three stacks, and the two stacks on the right constitute the goal situation. To change the configuration, a free block at the top of some stack can be moved on top of another stack or to the table. That is, a block offers room for at most one other block on top of it, while any number of blocks can be put on the table. A naive strategy to establish the goal situation thus consists of successively moving all blocks to the table, and then build up required stacks from the bottom. For the displayed scenario, this results in six moves to the table plus seven moves to construct the stacks on the right. However, the interest is to perform as few moves as needed, and in the following we show how shortest plans can be found using multi-shot solving.

Problem Instance

Similar to one-shot solving, applied to the TSP before, a problem instance is described in terms of facts. Those representing the situations displayed in Figure 3 are given in Listing 4, where the predicates `init`/2 and `goal`/2 specify the respective positions of blocks. In addition, `table(0)` declares `0` as identifier for the table, which is at the bottom of stacks in both the initial and the goal configuration.

Problem Encoding

In order to exploit the multi-shot solving capacities provided by the `clingo` system (Gebser et al. 2014), the encoding given in Listing 5 is composed of three subprograms. Their

Listing 5: Blocks world encoding w.r.t. facts as in Listing 4

```
#program base.
% DOMAIN
do(X,Z) :- init(X,Y), not table(Y), table(Z).
do(X,Y) :- goal(X,Y), not table(Y).
on(X,Y,0) :- init(X,Y).

#program check(t).
% TEST
:- query(t), goal(X,Y), not on(X,Y,t).

#program step(t).
% GENERATE
{move(X,Y,t) : do(X,Y)} = 1.
% DEFINE
move(X,t) :- move(X,Y,t).
on(X,Y,t) :- move(X,Y,t).
on(X,Y,t) :- on(X,Y,t-1), not move(X,t).
lock(Y,t) :- on(X,Y,t-1), not table(Y).
firm(X,t) :- on(X,Y,t), table(Y).
firm(X,t) :- on(X,Y,t), firm(Y,t).
% TEST
:- lock(X,t), move(X,t).
:- lock(Y,t), move(X,Y,t).
:- init(Y,Z), #count{X : on(X,Y,t)} > 1.
:- init(X,Z), #count{Y : on(X,Y,t)} > 1.
:- init(X,Z), not firm(X,t).

% DISPLAY
#show move/3.
```

names and parameters are introduced by `#program` directives, and a subprogram includes the rules up to the next such directive (if any). In the context of planning, the subprograms are dedicated to the following concerns:

- A `base` subprogram is processed once for providing auxiliary concepts along with setting up an initial configuration.
- A `check(t)` subprogram is parametrized by a constant `t`, serving as a placeholder for successive integers starting from `0`. For each time point taken as value to replace `t` with, integrity constraints impose goal conditions. They include a dedicated atom `query(t)`, provided by `clingo` for the current last time point, while obsolete conditions are deactivated to reflect an increased plan length.
- A `step(t)` subprogram is likewise parametrized, yet `t` is replaced with successive integers starting from `1`. This subprogram specifies transitions in terms of rules for picking actions, deriving atoms that represent a successor configuration, and asserting the validity of a transition. In contrast to `check(t)`, such rules are not withdrawn but joined with others obtained at later time points.

The subprograms are further structured into conceptual `DOMAIN`, `GENERATE`, `DEFINE`, and `TEST` parts. Moreover, the `DISPLAY` part declares `move`/3 as output predicate (for all subprograms) via the **#show** directive in Line 29, while the solving process of `clingo` focuses on shortest plans without requiring any `OPTIMIZE` part.

base: The first subprogram, called `base`, contributes a `DOMAIN` part consisting of the rules from Line 3 to 5. The idea of the predicate `do`/2 is to provide moves that could be relevant to a shortest plan. In particular, the rule in Line 3 expresses that moving a block to the table can be useful for accessing the stack underneath, but only if such a stack exists and the block is not already on the table in the initial situation. Given the stacks on the left in Figure 3, we thus derive that the blocks numbered 2, 3, 5, 6, 8, and 9 may be moved to the table. In addition, the rule in Line 4 indicates moves to goal positions different from the table. Regarding the goal configuration on the right in Figure 3, we obtain corresponding moves for all blocks but those numbered 1 and 9. As a result, derived facts over `do`/2 yield at most two relevant moves per block, while other moves may be legal but cannot belong to shortest plans.[8] The remaining rule in Line 5 maps initial positions to derived facts over `on`/3, where the integer `0` denotes a time point associated with the initial configuration.

check(*t*): The subprogram `check(`*t*`)` is parametrized by a constant *t* that is handled by `clingo` and replaced with successive integers starting from `0`. It contributes a `TEST` part, including the integrity constraint in Line 9, to deny plans such that some goal position is not yet established at the last time point referred to by *t*. This is accomplished by means of a dedicated atom `query(`*t*`)`, provided by `clingo` for the current last time point and deactivated when proceeding to the next integer to replace *t* with. For example, the initial position of block 3 on the left in Figure 3 does not match its goal position on the right, and a contradiction is indicated via

```
:- query(0), goal(3,9), not on(3,9,0).
```

However, `query(0)` holds only as long as `0` is the last time point, while `query(1)` is used for `1` instead, and so on.

step(*t*): The third subprogram, denoted by `step(`*t*`)`, specifies transitions to time points referred to by its parameter *t*, serving as a placeholder for successive integers starting from `1`. To begin with, the choice rule in Line 13 constitutes the `GENERATE` part for picking one among the moves taken as relevant in the `base` subprogram. Note that the current time point is used as third argument in atoms over `move`/3, while `do`/2 remains fixed, regardless of the time point.

[8]More elaborate conditions to further restrict potential moves are provided in (Slaney and Thiébaux 2001), and respective ASP encodings are presented in (Gebser et al. 2012, Section 8.2). While such domain knowledge as well as the encoding in Listing 5 are specific to Blocks World Planning, domain-independent approaches to model actions and change are discussed by Erdem, Gelfond, and Leone (2016) in this issue.

Listing 6: `clingo` run on facts and encoding in Listing 4-5

```
$ clingo blocks-ins.lp blocks-enc.lp

Solving...
Solving...
Solving...
Solving...
Solving...
Solving...
Solving...
Solving...
Solving...
Solving...

Answer: 1
  move(9,0,1) move(6,0,2) move(3,9,3)
  move(8,0,4) move(7,3,5) move(5,7,6)
  move(4,2,7) move(6,4,8) move(8,6,9)
```

The deterministic rules in the `DEFINE` part from Line 15 to 20 derive further atoms characterizing a transition at hand. A block changing its position is extracted via projection to `move`/2. Atoms over `on`/3, representing a successor configuration, are derived from a move as well as inertia applying to all blocks but the one that is moved. Again harnessing projection, the predicate `lock`/2 indicates blocks that were not on top of a stack and can thus not participate in legal moves. Finally, the predicate `firm`/2 provides blocks rooted on the table in a successor configuration, where non-circular derivations similar to those for places reachable in the TSP have the table as their starting point.

The `TEST` part, including the integrity constraints from Line 22 to 26, then eliminates inexecutable plans. Moves involving inaccessible blocks are ruled out in Line 22 and 23, which is actually sufficient to check that a plan can be executed. Notably, the first of these integrity constraints reuses the projection to `move`/2, as only the moved block is of interest here. In addition, Line 24 to 26 impose redundant *state constraints*, making explicit that, in any configuration, no block is under or on several objects and all blocks are rooted on the table.[9] E.g., this expresses that block 3 cannot be at its goal position in between the blocks numbered 7 and 9 as long as the third stack displayed on the left in Figure 3 is intact, no matter the performed moves.

Solution Computation

The output of `clingo` run on the facts in Listing 4 and the encoding in Listing 5, stored in text files `blocks-ins.lp` and `blocks-enc.lp`, is given in Listing 6. The ten lines saying '`Solving...`' indicate that ten time points, viz. successive integers from `0` to `9`, have been used for the parameter of the `check(`*t*`)` subprogram. Apart from `0`, they are also applied to the `step(`*t*`)` subprogram describing transitions, while `base` is processed just once at the beginning.

[9]Similar constraints are also included in encodings presented in (Erdem and Lifschitz 2003; Gebser et al. 2012; Lifschitz 2002) and further pave the way to partially ordered plans with parallel actions.

Failed attempts to find an answer set for time points from `0` to `8` mean that there is no plan consisting of a respective number of moves. In turn, the plan found for time point `9` is shortest. The contained atoms over `move`/3 mainly convey that moving the blocks numbered 6, 8, and 9 to the table allows for building up the goal stacks. Alternative shortest plans, which can be obtained by enumerating answer sets, include a move of block 5, rather than block 8, to the table.

Summary

The blocks world is a dynamic domain, in which actions change the state of the environment over time. Shortest plans to progress from an initial to a goal situation can be found using multi-shot solving according to some basic principles:

- An instance is provided by facts specifying the objects of interest along with initial and goal conditions.
- A general problem encoding furnishes three subprograms, called `base`, `check(`*t*`)`, and `step(`*t*`)`. The latter are parametrized by a constant, here denoted *t*, serving as a placeholder for successive integers starting from `0` or `1`, respectively.
- The `base` subprogram is processed once at the beginning. It typically contributes a `DOMAIN` part setting up auxiliary concepts as well as atoms representing an initial configuration.
- Occurrences of parameter *t* in the `check(`*t*`)` subprogram are successively replaced with integers from `0`. The common purpose is to impose goal conditions by means of integrity constraints in a `TEST` part. By using a dedicated atom `query(`*t*`)` in integrity constraints, obsolete conditions are deactivated when proceeding to the next integer.
- The `step(`*t*`)` subprogram is processed analogously to `check(`*t*`)`, yet starting from integer `1` instead of `0`. This predestinates `step(`*t*`)` to specify the transition to a successor configuration associated with *t*. The constructs typical for `GENERATE`, `DEFINE`, and `TEST` parts are used to provide candidates, derive atoms characterizing them, and eliminate invalid transitions. Invariant properties can be expressed by incorporating redundant state constraints.
- While facts as well as the `base` subprogram are processed only at the beginning, multi-shot solving by `clingo` iteratively adds rules obtained by replacing the parameters of the `check(`*t*`)` and `step(`*t*`)` subprograms with successive integers. This corresponds to gradually increasing the plan length until an answer set representing a shortest plan is found. The required length is often not known a priori, and multi-shot solving allows for discovering it.

Preferences and Optimization

The identification of preferred, or optimal, solutions is often indispensable in real-world applications, as illustrated on the TSP and blocks world scenarios above. In many cases, this also involves the combination of various qualitative and quantitative preferences.

In fact, optimization statements representing objective functions based on summation or counting are integral concepts of ASP systems since their beginnings (manifested by **`#minimize`** and **`#maximize`** statements (Simons, Niemelä, and Soininen 2002) or weak constraints (Leone et al. 2006)). The built-in repertoire of current ASP systems also covers set-inclusion based optimization (Gebser et al. 2015).

Other approaches to optimizing relative to specific and often more complex types of preference are furnished by dedicated external systems. Such approaches can be categorized into two classes (cf. (Delgrande et al. 2004)). On the one hand, we find prescriptive approaches to preference that take an order on rules and then enforce this order during the construction of optimal answer sets (Brewka and Eiter 1999). Such prescriptive approaches do not lead to an increase in computational complexity, which makes them amenable to implementation by compilation (Delgrande, Schaub, and Tompits 2003) or meta-interpretation (Eiter et al. 2003). On the other hand, we have descriptive approaches that impose preferences among the answer sets of a program (Brewka, Niemelä, and Truszczyński 2003; Sakama and Inoue 2000; Son and Pontelli 2006). Unlike the former, these approaches typically lead to an elevated level of complexity, which makes their efficient implementation more challenging. The `asprin` system (Brewka et al. 2015) offers a general and flexible framework for computing optimal answer sets relative to preferences among them.[10] In particular, its library comprises all aforecited descriptive approaches and further allows for freely combining preferences of qualitative and quantitative nature.

Further Extensions

The previous sections presented some popular modeling features and extensions, e.g., relative to propositional Satisfiability (SAT), going along with the ASP methodology. These include uniform problem representations using (first-order) variables within encodings, aggregates expressing collective conditions on sets, optimization, and multi-shot solving capacities. While such concepts already provide rich facilities for modeling and solving complex computational problems, we conclude with a (non-exhaustive) overview of further extensions.

Similar to disjunctive rules, non-monotone recursive aggregates (Faber, Pfeifer, and Leone 2011; Ferraris 2011) allow for expressing problems at the second level of the polynomial time hierarchy. Finite-domain constraints specifying quantitative conditions can be addressed through dedicated back-ends (Aziz, Chu, and Stuckey 2013; Balduccini 2011; Mellarkod, Gelfond, and Zhang 2008; Ostrowski and Schaub 2012) or compilation (Banbara et al. 2015; Drescher and Walsh 2010). Moreover, translation approaches allow for handling real numbers (Bartholomew and Lee 2013; Liu, Janhunen, and Niemelä 2012). Extended functionalities

[10]The only requirement is that evaluating a preference must be encodable in ASP (and thus have a complexity not beyond the second level of the polynomial time hierarchy).

like multi-shot solving are realized by combining ASP systems with scripting languages (Gebser et al. 2014). Further details regarding the integration of ASP systems with imperative languages or external information sources are provided by Lierler, Maratea, and Ricca (2016) and Erdem, Gelfond, and Leone (2016) in this issue. As also discussed in the latter article, high-level problem representations, e.g., specified in terms of action languages, can in turn be mapped to ASP via corresponding front-ends.

Acknowledgments The authors were partially funded by DFG grants SCHA 550/8 and SCHA 550/9. We are grateful to Esra Erdem, Francesco Ricca, Gerhard Brewka, Mirek Truszczyński, and Wolfgang Faber for their comments on a preliminary draft of this article.

References

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

Balduccini, M. 2011. Industrial-size scheduling with ASP+CP. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, 284–296. Springer.

Banbara, M.; Gebser, M.; Inoue, K.; Ostrowski, M.; Peano, A.; Schaub, T.; Soh, T.; Tamura, N.; and Weise, M. 2015. aspartame: Solving constraint satisfaction problems with answer set programming. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 112–126. Springer.

Bartholomew, M., and Lee, J. 2013. Functional stable model semantics and answer set programming modulo theories. In *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*, 718–724. IJCAI/AAAI Press.

Brewka, G., and Eiter, T. 1999. Preferred answer sets for extended logic programs. *Artificial Intelligence* 109(1-2):297–356.

Brewka, G.; Delgrande, J.; Romero, J.; and Schaub, T. 2015. asprin: Customizing answer set preferences without a headache. In *Proceedings of the Twenty-Ninth National Conference on Artificial Intelligence (AAAI'15)*, 1467–1474. AAAI Press.

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

Brewka, G.; Niemelä, I.; and Truszczyński, M. 2003. Answer set optimization. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 867–872. Morgan Kaufmann Publishers.

Bruynooghe, M.; Denecker, M.; and Truszczyński, M. 2016. ASP with first-order logic and definitions. *AI Magazine*. This issue.

Delgrande, J.; Schaub, T.; Tompits, H.; and Wang, K. 2004. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence* 20(2):308–334.

Delgrande, J.; Schaub, T.; and Tompits, H. 2003. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming* 3(2):129–187.

Drescher, C., and Walsh, T. 2010. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming* 10(4-6):465–480.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 2003. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming* 3(4-5):463–498.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer set programming: A primer. In *Proceedings of the Fifth International Reasoning Web Summer School (RW'09)*, 40–110. Springer.

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming* 3(4-5):499–518.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine*. This issue.

Faber, W.; Pfeifer, G.; and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* 175(1):278–298.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic* 12(4):25.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan and Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Available at `http://arxiv.org/abs/1405.3694v1`.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Romero, J.; and Schaub, T. 2015. Progress in clasp series 3. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 368–383. Springer.

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

Janhunen, T., and Niemelä, I. 2016. The answer set programming paradigm. *AI Magazine*. This issue.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine*. This issue.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

Lierler, Y.; Maratea, M.; and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine*. This issue.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138(1-2):39–54.

Lifschitz, V. 2016. Answer sets and the language of answer set programming. *AI Magazine*. This issue.

Liu, G.; Janhunen, T.; and Niemelä, I. 2012. Answer set programming via mixed integer programming. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, 32–42. AAAI Press.

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

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

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

Ostrowski, M., and Schaub, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming* 12(4-5):485–503.

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

Sakama, C., and Inoue, K. 2000. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence* 123(1-2):185–222.

Schlipf, J. 1995. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences* 51:64–86.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

Son, T., and Pontelli, E. 2006. Planning with preferences using logic programming. *Theory and Practice of Logic Programming* 6(5):559–608.

Van Gelder, A.; Ross, K.; and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.