Applications of ASP

Esra Erdem
Sabanci University
Orhanli, Tuzla, 34956 Istanbul, Turkey

Michael Gelfond
Texas Tech University
Lubbock, Texas 79414, USA

Nicola Leone
University of Calabria
87030 Rende (CS), Italy

Abstract

ASP has been applied fruitfully to a wide range of areas in AI and in other fields, both in academia and in industry, thanks to the expressive representation languages of ASP and the continuous improvement of ASP solvers. We present some of these ASP applications, in particular, in knowledge representation and reasoning, robotics, bioinformatics and computational biology as well as some industrial applications. We discuss the challenges addressed by ASP in these applications and emphasize the strengths of ASP as a useful AI paradigm.

1 Introduction

Answer Set Programming (ASP) is a knowledge representation and reasoning paradigm. It has rich high-level representation languages that allow recursive definitions, aggregates, weight constraints, optimization statements, default negation, and external atoms. With such expressive languages, ASP can be used to declaratively represent knowledge (e.g., mathematical models of problems, behaviour of dynamic systems, beliefs and actions of agents) and solve combinatorial search problems (e.g., planning, diagnosis, phylogeny reconstruction) and knowledge-intensive problems (e.g., query answering, explanation generation). The idea is to represent a problem as a "program" whose models (called "answer sets" (Gelfond and Lifschitz 1988; 1991)) correspond to the solutions of the problem. The answer sets for the given program can then be computed by special software systems called answer set solvers, such as DLV, SMODELS or CLASP.

Due to the continuous improvement of ASP solvers and expressive representation languages of ASP, ASP has been applied fruitfully to a wide range of areas in AI and in other fields. Areas of AI that include applications of ASP are planning, probabilistic reasoning, data integration and query answering, multi-agent systems, natural language processing and understanding, learning, theory update/revision, preferences, diagnosis, description logics, semantic web, multi-context systems, and argumentation. Other areas that include applications of ASP are, for instance, computational biology, systems biology, bioinformatics, automatic music composition, assisted living, software engineering, bounded model checking, and robotics.

ASP has also been used in industry, for instance, for decision support systems (Nogueira et al. 2001) (used by United Space Alliance), automated product configuration (Tiihonen, Soininen, and Sulonen 2003) (used by Variantum Oy), intelligent call routing (Leone and Ricca 2015) (used by Italia Telecom) and (re)configuration of railway safety systems (Aschinger et al. 2011) (used by Siemens).

In the following, we will describe only some of these ASP applications, in particular, in knowledge representation and reasoning, robotics, bioinformatics, as well as some industrial applications. For a wide variety of ASP applications and relevant references, we refer the reader to the ASP Applications Table at `https://www.dropbox.com/s/pe261e4qi6bcyyh/aspAppTable.pdf?dl=0`.

2 ASP and Knowledge Representation

2.1 Reasoning with Defaults

One of the main goals of AI is to better understand how to build software components of agents capable of reasoning and acting in a changing environment. Most AI researchers agree that to exhibit such behavior the agent should have a mathematical model of its environment and its own capabilities and goals. A logic based approach to AI (McCarthy 1990) suggests that this model should contain a knowledge base (KB)—a collection of statements in some declarative language with precisely defined syntax and semantics. As a rule, such a KB should include commonsense knowledge—information an ordinary person is expected to know—as well as some specialized knowledge pertinent to a particular set of activities the agent is built to perform. Early proponents of the logic based AI believed that such a KB could be built in classical first-order logic (FOL) which, at the time, was commonly used for formalization of mathematical reasoning. It was quickly discovered, however, that this logic may not be a fully adequate tool for representing non-mathematical (especially commonsense) knowledge. The main problem was difficulty with using FOL for defeasible (or nonmonotonic) reasoning. In precise terms, a consequence relation $A \models F$ between statements of a declarative language is called *monotonic* if, for every A, B, and F, if

Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

$A \models F$ then $(A \wedge B) \models F$. This property guarantees that, once proven, a statement stays proven. If this condition is not satisfied, i.e., if addition of new information can force a reasoner to withdraw its previous conclusion, the consequence relation is called *nonmonotonic*. Though not encountered in mathematics, nonmonotonicity seems to be a prevailing feature of commonsense reasoning. It is especially relevant to reasoning with so called *defaults*—statements of the form *"Normally (typically, as a rule) elements of class C have property P."* We all learn rather early in life that parents normally love their children, citizens are normally required to pay taxes, etc. We also learn, however, that these rules are not absolute and allow various types of exceptions. It is natural to assume that these and other defaults should be included in a reasoner's KB. Learning correct ways to reason with defaults and their exceptions is necessary for building an agent capable of using such a KB. One of the best available solutions to this problem uses a knowledge representation language CR-Prolog (Balduccini and Gelfond 2003)—a simple extension of the original ASP language of logic programs with two types of negation and epistemic disjunction.

A program Π of CR-Prolog consists of a first-order signature, a collection Π^r of standard ASP rules of the form

$$l_0 \mid \ldots \mid l_k \leftarrow l_{k+1}, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$$

and a collection Π^{cr} of *consistency-restoring rules* (or simply cr-rules)[1] of the form:

$$l_0 \stackrel{+}{\leftarrow} l_{k+1}, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n.$$

Here each l_i is a *literal*, that is, an atom $p(\bar{t})$ or its negation $\neg p(\bar{t})$. The last statement says that $\bar{t}$ does not have property p. In contrast, default negation not has an epistemic character—*not* l is often read as "*it is not believed that l is true*". Similarly, the connective $\mid$ (also denoted by *or*) is often called *epistemic disjunction* with $l_1 \mid l_2$ being read as l_1 *is believed to be true or* l_2 *is believed to be true*. Intuitively, a regular ASP rule *Head* $\leftarrow$ *Body* says that if the body of the rule is believed then the reasoner *must* believe its head. A cr-rule says that if the body of the rule is believed, then the reasoner *may* possibly believe its head; however, this possibility may be used only if Π^r is inconsistent.

Informally, program Π can be viewed as a specification for answer sets—sets of beliefs that could be held by a rational reasoner associated with Π. Answer sets are represented by collections of ground literals. In forming such sets the reasoner must satisfy the rules of Π together with a so called *rationality principle*, which says that the reasoner associated with the program shall believe nothing that he is not forced to believe by the program's rules. In the absence of cr-rules this idea is captured by the standard answer set semantics.

The definition of an answer set for arbitrary CR-Prolog program is as follows. For a collection R of cr-rules, by $\alpha(R)$ we denote the collection of regular rules obtained by replacing labeled arrows in cr-rules of R by $\leftarrow$. A minimal (with respect to the preference relation of the program) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (that is has an answer set) is called an abductive support of Π. A set A is called an answer set of Π if it is an answer set of the regular ASP program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

In the following, we assume that the rules that are in teletype font are in the the ASP Core language ASP-Core-2,[2] whereas the rules in math font (e.g., cr-rules) are in languages (e.g., CR-Prolog) that extend the ASP Core language in different ways. The schematic variables (resp. object constants) in rules are denoted by strings whose first letters are in upper-case (resp. in lower-case).

Example 1 *[Representing Defaults]* A default *"parents normally love their children"* can be represented by the following rule:

```
(1) loves(P,C) :-
        parent(P,C), not -loves(P,C).
```

Consider a program P_1 consisting of this rule and a fact:

```
(2) parent(mary,john).
```

Since the answer set semantics of CR-Prolog incorporates the rationality principle and no rule of the program forces the reasoner to believe that Mary does not love John, the first rule allows it to conclude that she does. Additional information,

```
(3) -loves(mary,john)
```

will not lead to a contradiction. The new statement will render the first rule inapplicable and allow the reasoner to withdraw its earlier conclusion. Statement (3) is an example of so-called direct exceptions to defaults, i.e., exceptions which directly contradict the default conclusion. The situation is not always that neat. Let us now consider a program P_2 consisting of statements (1) and (2) above, together with a new rule:

```
(4) cares(P,C) :- loves(P,C) .
```

It is easy to see that P_2 entails $cares(mary, john)$. If, however, we were to learn that Mary does not care for John and expanded P_2 by

```
(5) -cares(mary,john)
```

the new program, P_3, would become inconsistent, i.e., will not have answer sets. In our everyday reasoning we do not seem to have difficulties in dealing with such *indirect* exceptions to defaults. We would avoid the contradiction by simply concluding that Mary and John constitute an (indirect) exception to default (1). So the fact (5) is the only conclusion which can be derived from the program. To model this type of reasoning, we should make it possible for our program to recognize that the relationship between parents and their children may be not that of love, but still be able to use our default whenever possible. This is done using a cr-rule.

Indirect exceptions to default (1) can be represented as follows:

$$\neg loves(P, C) \stackrel{+}{\leftarrow} parent(P, C). \qquad (6)$$

[1] The definition of a program also includes the fourth component—a preference relation on sets of cr-rules. In what follows, we assume that a set with the smaller number of rules is preferred to that with the larger one.

[2] https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf

The new program P_4 consisting of regular rules (1), (2) and (4) and cr-rule (6) entails that Mary cares about John. A consistent answer set of the program can be obtained from its regular rules only and cr-rule (6) is not used. If, however, we expand P_4 by statement (5), regular rules of the program are not sufficient to avoid the contradiction. Consistency restoring rule (6) will be activated and the reasoner will conclude that Mary does not love John.

The example above is rather general and allows for representation of different types of exceptions to defaults. More information on this can be found in (Gelfond and Kahl 2014).

2.2 Reasoning about Effects of Actions

Gaining better understanding of basic principles and mathematical models of default reasoning helped researchers to move forward in solving a number of other longstanding problems of AI and KR. In this section we briefly describe an ASP-based solution of one such problem—finding logical means for representing and reasoning about direct and indirect effects of actions.

To act in a changing (dynamic) domain, a rational agent should have a mathematical model of this domain allowing it to predict such effects. Here we limit ourselves to discrete dynamic domains represented by transition diagrams whose nodes correspond to possible physical states of the domain and whose arcs are labeled by actions.

A transition $\langle \sigma_1, a, \sigma_2 \rangle$ indicates that the execution of action a in state σ_1 may cause the domain to move to state σ_2. Due to the size of the diagram, the problem of finding a concise specification for it is not trivial and has been a subject of research for a comparatively long time. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents—propositions whose truth value may depend on the state of the domain.

An additional level of complexity is added by the need to specify what is not changed by actions in a concise, clear, and elaboration tolerant way. A seminal paper (Hayes and McCarthy 1969) in which the problem of finding such a specification (called the Frame Problem) was discussed also suggested a direction in which its possible solution could be found. The proposal was to reduce the solution of the Frame Problem to the problem of finding a concise, accurate and elaboration tolerant representation of the inertia axiom—a default which says that things normally stay as they are. The search for such a representation substantially influenced AI research during the next thirty years. An interesting account of the history of this research together with some possible solutions can be found in (Shanahan 1997). We have already discussed the ways of using ASP for representing defaults and their exceptions so it shall not come as a surprise that ASP provides a good solution to the Frame Problem. It also turned out that rules of ASP languages can nicely capture causal relations between fluents which led to the development of a powerful methodology for representing and reasoning about actions and their effects.

We illustrate this methodology by way of example—representation of a simple hydraulic system of a space shuttle. The example is taken from an actual decision support software system (Nogueira et al. 2001) developed to help shuttle controllers to deal with critical situations caused by multiple failures.

Example 2 *[Effects of Actions]* Consider a hydraulic system viewed as a graph whose nodes are labeled by tanks containing propellant, jets, junctions of pipes, etc. Arcs of the graph are labeled by valves which can be opened or closed by a collection of switches. The system is used to deliver propellant from tanks to a proper combination of jets.

To axiomatize the knowledge pertinent to this example, we describe the graph by a collection of statements of the form $connected(N_1, V, N_2)$—valve V labels the arc from N_1 to N_2, and $controls(S, V)$—switch S controls valve V. Fluents $pos(S, open)$ and $pos(S, closed)$ define positions of switch S. Fluents $pos(V, open)$ and $pos(V, closed)$, defining the position of a valve, and fluent $pressurized(N)$, which holds when node N is reached by propellant from some tank, will be defined in terms of positions of switches of the system.

The following axiom

```
(1) -h(pos(X,P1),I) :-
       h(pos(X,P2),I), P1 != P2
```

guarantees that positions of switches and valves are mutually exclusive, i.e., cannot both be true at the same time-step. Here relation $h(F, I)$, where h stands for $holds$, is true if a fluent F holds (is true) at time-step I of the system's trajectory.

Now we concentrate on the representation of action $flip(S)$ which flips the switch S from position $open$ to position $closed$ and vice versa. Note that this action has comparatively complex effects including the propagation of the delivery of propellent from tanks to other nodes of the system. The effects will be divided into direct and indirect.

The direct effect of flipping a switch S from $closed$ to $open$ will be given by the following axiom

```
(2) h(pos(S,open),I+1) :-
       occurs(flip(S),I), h(pos(S,closed),I)
```

where $occurs(A, I)$ is true if action A occurs (happened, is executed) at I. The rule states that if action $flip(S)$ occurred at a time-step I in which the fluent $pos(S, closed)$ was true then at the next step, $I + 1$, the fluent $pos(S, open)$ would become true. A similar axiom is needed for flipping a switch from the $open$ to $closed$ position.

To represent indirect effects we simply need to state the relations between fluents of the domain. The next rule describes a relationship between fluents representing positions of switches and valves.

```
(3) h(pos(V,P),I) :-
       controls(S,V), h(pos(S,P),I).
```

The rule states that if a switch is placed in a particular position, then so is the valve controlled by this switch.

The next rule describes the relationship between the values of fluent $pressurized(N)$ for neighboring nodes.

```
(4) h(pressurized(N2),I) :-
      connected(N1,V,N2),
      h(pressurized(N1),I),
      h(pos(V,open),I).
```

The rule says that if nodes N_1 and N_2 are connected by open valve V and node N_1 is pressurized then so is node N_2. We also assume that tanks are always pressurized and encode this as follows:

```
(5) h(pressurized(N),I) :- tank(N), step(I).
```

To complete the definition of this fluent we need to state that no other nodes except those defined by rules (4) and (5) are pressurized. This is done by the rule

```
(6) -h(pressurized(N),I) :- node(N), step(I),
      not h(pressurized(N),I).
```

Suppose now that the system contains nodes n_1, n_2, and n_3 where n_1 is a tank; n_1 and n_2 are connected by valve v_1; n_2 and n_3 are connected by valve v_2; v_1 and v_2 are controlled by switches s_1 and s_2, respectively. Assume also that initially, the switches are closed. One can see that at the initial step 0, node n_1 is pressurized (axiom (5)), and nodes n_2 and n_3 are not (axiom (6)). To compute the effects of flipping switch s_1 let us expand the program by statement

```
occurs(flip(s1),0).
```

The direct effect of this action, determined by axiom (2), is $h(pos(s_1, open), 1)$. There are also indirect effects which follow from axioms (3), (4), and (1): $h(pos(v_1, open), 1)$, $h(pressurized(n_2), 1)$, and $\neg h(pos(v_1, closed), 1)$. To complete our formalization we need to add our solution to the Frame Problem, which will allow us to conclude that flipping switch s_1 does not change the status of switch s_2 and valve v_2. As discussed earlier, this can be done by simply axiomatizing the default stating that normally the value of fluent $pos(S, Val)$ remains unchanged:

```
(7a) h(pos(S,Val),I+1) :-
       switch(S), h(pos(S,Val),I),
       not -h(pos(S,Val),I+1).
(7b) -h(pos(S,Val),I+1) :-
       switch(S), -h(pos(S,Val),I),
       not h(pos(S,Val),I+1).
```

This is, of course, a typical ASP representation of a default which provides the solution to the Frame Problem. It guarantees that at step 1 switch s_2 is still closed. Since positions of v_2 and the value of $pressurized(n_3)$ are fully determined by positions of the switches, nothing else is necessary—v_2 will remain closed and n_3 depressurized.

The ability of ASP languages to represent defaults and to express indirect effects of actions by a unidirectional implication made it a good tool for representing knowledge about dynamic domains. Nowadays, however, such knowledge is more frequently represented in so-called action languages (Gelfond and Lifschitz 1998) which are more specialized, higher-level languages designed for specifying state-action-state transition diagrams. Consider, for example, one of such languages, called $\mathcal{AL}$ (Gelfond and Kahl 2014). Axiom (2) from Example 2 may be written in $\mathcal{AL}$ as

$$\textit{flip}(S) \textbf{ causes } \textit{pos}(S, \textit{open}) \textbf{ if } \textit{pos}(S, \textit{closed})$$

which is a special case of a dynamic causal law of $\mathcal{AL}$—a statement of the form

$$A \textbf{ causes } F \textbf{ if } P.$$

The law says that *execution of action A in a state satisfying property P causes fluent F to become true in a resulting state*. Axiom (4) from Example 2 may be written as

$$\textit{pressurized}(N_2) \quad \textbf{if} \quad \begin{array}{l} \textit{pressurized}(N_1), \\ \textit{connected}(N_1, V, N_2), \\ \textit{pos}(V, \textit{open}) \end{array}$$

which is a special case of an action language $\mathcal{AL}$ statement

$$F \textbf{ if } P.$$

The statement guarantees that every state of the system satisfying property P also satisfies F. The inertia axioms (7a) and (7b) can be replaced by the simpler statement

$$fluent(\textit{inertial}, \textit{pos}(S, Val))$$

which indicates that the fluent $\textit{pos}(S, Val)$ is subject to the inertia axiom. Overall, use of action languages leads to substantially simpler representations allowing the system designer to avoid some ASP related details. ASP, however, continues to play an important role in reasoning about actions.

First, answer set semantics of logic programs is often used to define semantics of action languages. Natural translations from action languages to logic programs allow us to use the notion of answer set to precisely define the effects of executing an action A in a state σ. Rules (1)–(6) of Example 2 can be viewed as part of such translation from the description of our domain in an action language. The translation will also contain a more general version of axioms (7a) and (7b):

```
h(F,I+1) :-
    fluent(inertial,F), h(F,I), not -h(F,I+1)
-h(F,I+1) :-
    fluent(inertial,F), -h(F,I), not h(F,I+1)
```

which provide a rather general solution of the Frame Problem.

Second, translation from action languages to logic programs enables us to reduce classical reasoning tasks such as prediction, planning, and finding explanations of unexpected events to computing answer sets of logic programs. An interested reader may look into (Gelfond and Kahl 2014) for further details.

There are other interesting applications of ASP to classical KR problems. These include its early use for providing a declarative semantics to the negation as failure construct of the Prolog programming language (Gelfond and Lifschitz 1988) as well as comparatively recent work on combining subtle forms of logical and probabilistic reasoning (Baral, Gelfond, and Rushton 2009).

3 Applications of ASP to Robotics

ASP has been applied in various robotic applications, such as assembly planning, mobile manipulation, geometric rearrangement, multi-robot path finding, multi-robot coordination, multi-robot planning, plan execution and monitoring,

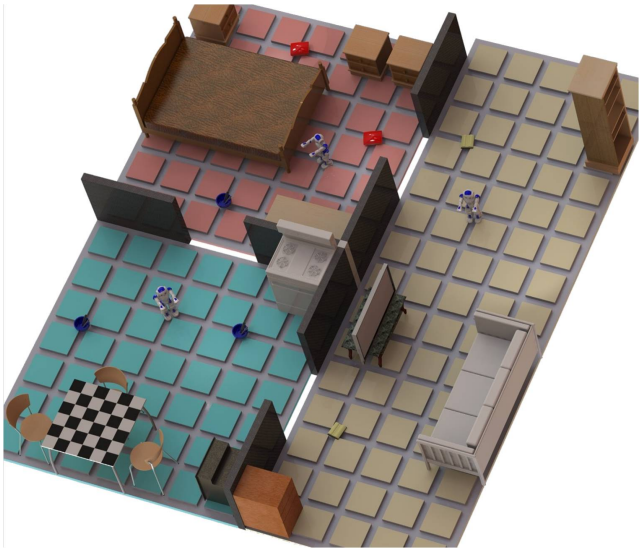

Figure 1: Multiple robots tidying up a house.

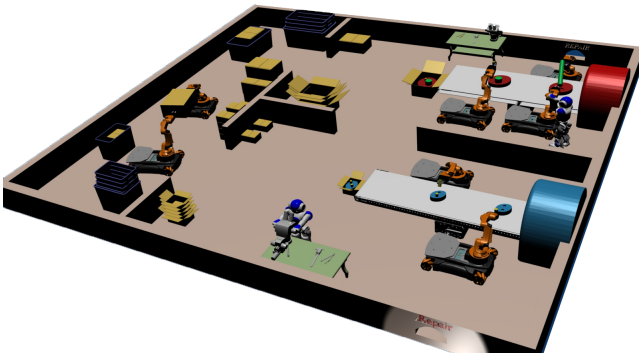

Figure 2: Multiple robots collaboratively working in a cognitive factory.

and human-robot interaction, to provide methods for high-level reasoning (like planning, hypothetical reasoning, diagnostic reasoning) and for declarative problem solving (like team coordination, gridization of continuous space).

For instance, Erdem et al. (2012) use ASP for planning of actions of multiple robots to collaboratively tidy up a house within a given time (Figure 1). They illustrate applications of their ASP-based planning, execution and monitoring approach with dynamic simulations with PR2 robots. Videos of these simulations can be seen at `http://cogrobo.sabanciuniv.edu/?p=214`.

In another study (Erdem et al. 2013), they use ASP to find an optimal global plan for multiple teams of heterogeneous robots in a cognitive factory to manufacture a given number of orders within a given time (Figure 2). They also use ASP for finding diagnosis of plan failures during plan execution monitoring (Erdem, Patoglu, and Saribatur 2015). They illustrate applications of their ASP-based planning with dynamic simulations and with an augmented reality physical implementation utilizing KuKa youBots and Lego NXT robots controlled over Robot Operating System (ROS). They show applications of their execution monitoring algorithm, in particular, the use of diagnostic reasoning for replanning and repairs, with dynamic simulations using Kuka youBots and a Nao humanoid robot. Videos of these simulations and implementations can be found at `http://cogrobo.sabanciuniv.edu/?cat=24`.

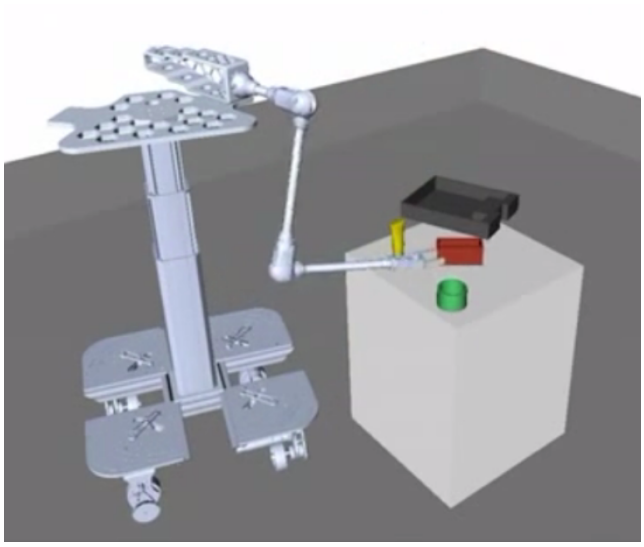

Figure 3: A mobile service robot rearranging objects on a cluttered table top.

Havur et al. (2014) use ASP for geometric rearrangement of multiple movable objects on a cluttered surface, where objects can change locations more than once by pick and/or push actions (Figure 3). They use ASP for gridization of the continuous plane for a discrete placement of the initial configurations and the tentative final configurations of objects on the cluttered surface, and for planning of robots' actions. The authors illustrate applications of their method with the CoCoA service robot, which features a holonomic mobile base and two 7 degrees of freedom (DoF) arms with grippers. Videos showing these simulations can be found at `http://cogrobo.sabanciuniv.edu/?p=762`.

Zhang et al. (2015) use ASP to describe objects and relations between them, and utilize this knowledge to improve localization of target objects in an indoor domain using (primarily) visual data. Such a use of ASP has been illustrated by a physical implementation with a wheeled robot navigating in an office building. Videos of these applications can be found at `http://youtu.be/CvKJyCI_YNE` and `http://youtu.be/2U6oOTuEd-Q`.

In these robotic applications, there are some important challenges from the point of view of robotic planning and diagnosis. The following discusses how ASP can handle them.

Hybrid reasoning One of the key challenges addressed in these robotic applications is hybrid reasoning, which can be understood as integrating high-level reasoning tasks, such as planning, hypothetical reasoning, and diagnosis, with low-level external computations. These external computations include, for instance, feasibility checks of robotic actions us-

ing probabilistic motion planning methods, as well as automatic extraction of relevant commonsense knowledge from the existing knowledge bases available on the web. Such a variety of hybrid reasoning is possible in ASP, thanks to "external atoms" (Eiter et al. 2006). These atoms provide a general interface between high-level reasoning and low-level external computations, in the spirit of semantic attachments in theorem proving. More precisely, an external atom is an expression of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called input and output lists, respectively), and g is an external predicate name. Intuitively, an external atom provides a way for deciding the values of an output tuple with respect to the values of an input tuple. External atoms allow us to embed results of external computations into ASP programs. Therefore, external atoms are usually implemented in a programming language of the user's choice.

Integrating high-level reasoning with low-level feasibility checks: Consider, for instance, multiple robots rearranging objects on a cluttered table (Havur et al. 2014). The objects can only move when picked up and placed, or pushed by robots, and the order of pick-and-place and push operations for rearranging objects may matter to obtain a feasible kinematic solution. Therefore, motion planning (e.g., finding a continuous trajectory from one configuration of the robot to another configuration) and other low-level feasibility checks alone are not sufficient to solve them. On the other hand, manipulation of objects also requires feasibility checks, such as whether the robot will be able to move an object from one location to another location without colliding with the other objects, or whether the robot will be able to reach the object on the table and grasp it without any collisions. Therefore, task planning only (e.g., finding a sequence of robotic actions from an initial state to a goal state) is not sufficient to solve the problem either. These examples illustrate the necessity for a hybrid approach to planning, that integrates task planning with feasibility checks.

One of the preconditions of the action *pickPlace*(R, O, C) of a robot R picking and placing an object O onto location C is that the object O is graspable by the end effector of the robot. This precondition can be formalized in ASP as follows:

$$\leftarrow \textit{occurs}(\textit{pickPlace}(R, O, C), I),$$
$$\textit{not}\ \ \&\textit{reachableGraspable}[O, R]().$$

Here *&reachableGraspable*$[O, R]()$ is an external atom; it returns true if and only if the end-effector of the manipulator R can successfully reach and grasp the given object O according to kinematics and force-closure calculations of OPENRAVE. Note that these calculations are done in a continuous configuration space using real-numbers; and thus are not possible in ASP.

One of the preconditions of the action *push*(R, O, C) of a robot R pushing an object O to location C is that the volume swept by the object O from its current configuration towards another configuration in C does not collide with other objects. This precondition can be described as follows:

$$\leftarrow \textit{occurs}(\textit{push}(R, O, C), I),$$
$$\textit{not}\ \ \&\textit{pushPossible}[\textit{location}, O, I]().$$

Here *&pushPossible* is an external predicate as well: it takes as input all locations of objects at time step I, and checks whether the swept volume of the object O collides with other objects using Open Dynamics Engine (ODE).

Embedding commonsense knowledge in high-level reasoning: Consider, for instance, the housekeeping domain with multiple robots (Erdem, Aker, and Patoglu 2012). The commonsense knowledge about expected locations Loc of objects Ep (e.g., dish in kitchen, bed in bedroom) can be extracted from the existing commonsense knowledge base CONCEPTNET (Liu and Singh 2004) by means of queries via its Python API; and can be defined by external atoms of the form $\&\,in\_place[Ep, Loc]()$. Then, one can represent the expected locations of objects Ep in the house by a new fluent of the form $at\_desired\_location(Ep)$ as follows:

$$h(at\_desired\_location(Ep), I) \leftarrow$$
$$h(at(Ep, Loc), I), \&\,in\_place[Ep, Loc]().$$

This rule formalizes that the object Ep is at its desired location if it is at some "appropriate" position Loc in the right room.

Another line of research that represents commonsense knowledge for service robotics applications is by Chen et al. (2010). In these applications, commonsense knowledge such as "a long-shape object B whose center-of-mass is on the table, is initially in balance if there is a can A on one end E_1 of it and another can B on the other end E_2 of it" is formulated in ASP:

```
h(balance(B,A,C),0) :-
   h(on(A,E_1),0), h(on(C,E_2),0),
   endof(E_1,B), endof(E_2,B).
```

Optimizations in planning and diagnosis There are various sorts of desired optimizations in robotic applications. For instance, in planning, an optimal plan can be understood as a plan with minimum makespan or a plan with minimum total cost of actions. In diagnostic reasoning, an optimal diagnosis can be understood as a hypothesis with a smallest number of broken parts of robots. Such optimizations are possible in ASP, thanks to "optimization statements".

For instance, consider the cognitive factories domain with multiple teams of heterogeneous robots (Erdem et al. 2013; Erdem, Patoglu, and Saribatur 2015). The following expression

```
#minimize {C,R,I : h(cost(R,C),I),
                   robot(R), step(I)}
```

is used to minimize the sum of all costs C of robotic actions performed in a local plan, where costs of actions performed by robot R at every time step are defined by fluents of the form *cost*(R, C).

The following statement minimizes the total number of the broken parts of robots while finding a diagnosis for a discrepancy:

```
#minimize {1,P,R: broken(R,P), comp(R,P)]
```

where atoms of the form $comp(R, P)$ describe robots and their parts, and atoms of the form $broken(R, P)$ describe that part P of the robot R is broken.

Complex constraints in replanning During plan execution, discrepancies between the observed state and the expected one may be detected that are relevant for the rest of the plan. These discrepancies may be due to an unexpected obstacle in the environment, change of locations of objects, broken robots, or failures of some actions. Once the cause of a discrepancy is detected, a new plan from the current state can be computed. While replanning, some guidance from earlier experiences and causes of discrepancies might be helpful to compute a better plan that does not fail due to the same reasons. ASP allows us to include such a guidance, by including constraints into the representation of the planning problem description. These constraints might express not only the new knowledge about the environment, robots and/or goals, but also what sort of actions should not be executed under what conditions and when. For instance, in the housekeeping domain (Erdem, Aker, and Patoglu 2012), if some robot's plan fails because the robot cannot pickup an object which turns out to be quite heavy, the robot might want to delay asking for help as much as possible so that the other robots are not distracted. Such a complex constraint (e.g., heavy objects can be picked with help only during the last three steps of the plan) can be represented in the planning problem description using ASP.

Commonsense knowledge and exceptions Consider, for instance, the housekeeping domain with multiple robots (Erdem, Aker, and Patoglu 2012). Normally, the movable objects in an untidy house are not at their desired locations. Such commonsense knowledge can be described by means of defaults, as in the following rules

```
-h(at_desired_location(Ep),I) :-
   endpoint(Ep), step(I),
   not h(at_desired_location(Ep),I).
```

In a similar way, the tidiness of a house is defined by means of defaults:

```
-h(tidy,I) :- -h(at_desired_location(Ep),I)
h(tidy,I) :- not -h(tidy,I), step(I).
```

The second rule above expresses that the house is normally tidy. The first rule above describes the exceptions: when an object is not at its expected location, the house is untidy.

Let us now consider diagnostic reasoning in cognitive factories with multiple teams of heterogeneous robots (Erdem, Patoglu, and Saribatur 2015). Normally, the robots and their parts run smoothly. However, there may be exceptions: some parts P of robots R that are not broken currently (at time step I) may get broken at the next state (at any time step I). This commonsense knowledge can be represented by means of defaults as well:

```
-h(broken(R,P),I) :- comp(R,P), step(I),
   not h(broken(R,P),I)
```

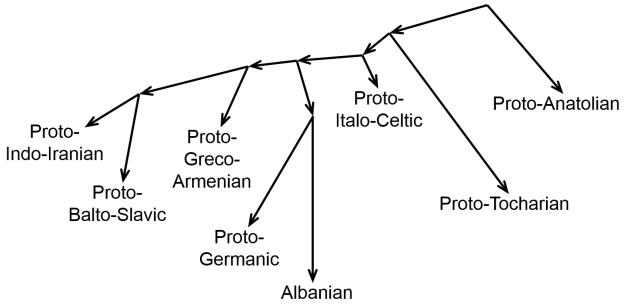

Figure 4: The most plausible phylogeny reconstructed for Indo-European languages.

```
h(broken(R,P),I+1) :- comp(R,P), step(I),
   -h(broken(R,P),I),
   not -h(broken(R,P),I+1).
```

Other examples of the use of ASP to represent expected locations of objects, by means of defaults, and to find diagnoses, by means of cr-rules, can be found in (Zhang et al. 2014).

4 Applications of ASP to Computational Biology and Bioinformatics

ASP has been applied in various computational biology and bioinformatics applications, providing a declarative problem solving framework for combinatorial search problems (e.g., haplotype inference, consistency checking in biological networks, phylogeny reconstruction) and providing a knowledge representation and reasoning framework for knowledge-intensive reasoning tasks (e.g., integrating, query answering and explanation generation over biomedical ontologies).

Tran and Baral (2009) introduce a method to model a biological signaling network as an action description in ASP to allow prediction, planning, and explanation generation about the network. They illustrate an application of their method to generate hypotheses about the various possible influences of a tumor suppressor gene on the p53 pathway. Gebser et al. (2011) introduce a method to model biochemical reactions and genetic regulations as influence graphs in ASP, to detect and explain inconsistencies between experimental profiles and influence graphs. With this method, they compare the yeast regulatory network with the genetic profile data of SNF2 knock-outs, and find out the data to be inconsistent with the network.

Brooks et al. (2007) use ASP to solve the problem of reconstructing phylogenies (i.e., evolutionary trees) for specified taxa, with a character-based cladistics approach. They apply their method to infer phylogenies for *Alcataenia* species as well as Indo-European languages (Figure 4) and Chinese dialects; these phylogenies are found plausible by the experts. Based on these phylogenies, phylogenetic networks are reconstructed as well (Erdem, Lifschitz, and Ringe 2006).

In the NMSU-PhyloWS project (Le et al. 2012), ASP is used to query the repository CDAOStore of phylogenies.

These queries are used, for instance, to find the trees satisfying a given property (e.g., whose size is smaller than a specified constant, with a specified ratio of internal nodes to external nodes), to find the similarity of two trees with respect to a distance measure (e.g., the Robinson-Foulds distance), to compute clades with specific properties (e.g., the minimum spanning clade for taxa in a specified tree).

Erdem et al. (2011) and Erdem and Oztok (2015) use ASP to answer complex queries over biomedical ontologies and databases considering the relevant parts of these knowledge resources, and to generate shortest explanations to justify these answers. They apply their methods to find answers and explanations to some complex queries related to drug discovery (e.g., "What are the genes that are targeted by the drug Epinephrine and that interact with the gene DLG4?", "What are the genes related to the gene ADRB1 via a gene-gene relation chain of length at most 3?" and "What are the most similar 3 genes that are targeted by the drug Epinephrine?") over the biomedical knowledge resources PHARMGKB, DRUGBANK, BIOGRID, CTD, SIDERand DISEASE ONTOLOGY.

Dovier et al. (2009) use ASP to study a variation of protein structure prediction problem: the 2D HP-protein structure prediction problem. The goal is find a folding in the 2D square lattice space, that maximizes the number of hydropic-hydropic contacts between given amino acids.

Erdem and Ture (2008) use ASP to solve the problem of Haplotype Inference by Pure Parsimony (HIPP) and its variations. Identifying maternal and paternal inheritance is essential for finding the set of genes responsible for a particular disease. However, due to technological limitations, we have access to genotype data (genetic makeup of an individual), and determining haplotypes (genetic makeup of the parents) experimentally is a costly and time consuming procedure. With these biological motivations, HIPP asks for the minimal number of haplotypes that form a given set of genotypes.

In these bioinformatics applications, one can identify some important challenges addressed by ASP; these challenges illustrate also the strengths of ASP.

Declarative problem solving The declarative representation formalism of ASP allows us to easily include domain specific information and constraints in the program, and thus to prevent the construction of implausible solutions. For instance, including some temporal and geographical constraints about Indo-European languages provided by historical linguists (e.g., "Albanian cannot be a sister of IndoIranian or BaltoSlavic") helps computing plausible phylogenies more efficiently.

Well-studied properties of programs in ASP allow us to easily prove the correctness of the formulation of the problem in ASP, as shown in (Erdem, Lifschitz, and Ringe 2006).

With a declarative representation of the problem in ASP, one can perform various reasoning tasks, such as ontological query answering and explanation generation (Le et al. 2012; Erdem et al. 2011; Erdem and Oztok 2015), planning and diagnosis (Tran and Baral 2009), consistency checking and explanation generation (Gebser et al. 2011), and repair and prediction (Gebser et al. 2010).

Integration of heterogeneous knowledge To answer complex queries over a variety of biomedical ontologies (Erdem et al. 2011), ASP allows us to extract relevant parts of them (thanks to external atoms) and integrate them by rules. For instance, the drug names can be extracted from a Drug Ontology, by first extracting the relevant triples from the ontology:

$$\textit{tripleD}(X,Y,Z) \leftarrow \\ \quad \&rdf["URI for Drug Ontology"](X,Y,Z)$$

and then extracting drug names from the triples:

```
drugName(A) :-
   tripleD(_,"drugproperties:name",A).
```

Then, to answer queries like "What are the drugs that treat the disease Depression and that do not target the gene ACYP1?", the extracted relevant knowledge can be integrated by rules as follows:

```
whatDrugs(DRG) :- cond1(DRG), cond2(DRG)
cond1(DRG) :- drugDisease(DRG,"Depression")
cond2(DRG) :- drugName(DRG),
   not drug_gene(DRG,"ACYP1").
```

Expressivity of representation ASP features rich, expressive formalisms (e.g., the support of recursive definitions and negation as failure), and efficient solvers that support special syntactic constructs (e.g., aggregates and optimization statements).

For instance, in (Gebser et al. 2011), candidates for minimal inconsistent components in an influence graph, where two distinct vertices are not reachable from each other by a cycle, can be eliminated by the following constraint:

```
:- active(U), active(V),
   not cycle(U,V), U<V.
```

where the definition of a cycle requires recursion:

```
reach(U,V) :- edge(U,V)
reach(U,V) :- edge(U,W),
   reach(W,V), vertex(V)
cycle(U,V) :- reach(U,V), reach(V,U), U!=V.
```

To answer queries like "What are the genes related to the gene ADRB1 via a gene-gene relation chain of length at most 3?", the auxiliary concept of reachability of a gene from another gene by means of a chain of gene-gene interactions is required (Erdem et al. 2011); this concept can be defined in ASP recursively as follows:

```
geneReachable(X,1) :-
   geneGene(X,Y), startGene(Y)
geneReachable(X,N+1) :-
   geneGene(X,Z), geneReachable(Z,N),
   max_chain_length(L), 0 < N, N < L.
```

Aggregates allow concise and easy-to understand formulations of problems. According to Le et al. (2012), we can identify phylogenies by the parsimony tree length, which is defined by the total number of characters of its taxa, by the following rules:

```
parsimonyLength(T,L) :- tree(T),
   L = #count {Char: belongsChar(_,Cell,Char),
                     belongsTU(_,Cell,TU),
                     representsTU(T,_,TU)}.
```

Negation as failure is useful to represent defaults (as seen in the examples of robotics applications) and the concept of unknown. For instance, we can define that some drugs' toxicity is unknown as follows (Erdem et al. 2011):

```
unknownToxicityDrug(X) :- drugSynonym(R,X),
   not drugToxic(R), not -drugToxic(R).
```

5 Industrial ASP Applications

As pointed out in the introduction, the availability of efficient ASP solvers has recently enabled the implementation of many advanced ASP applications, not only in academia but also in the industry. In this section, we briefly overview a number of real-world industrial applications of ASP. In particular, we will focus on ASP applications to *e-Tourism, Workforce management, Intelligent call routing,* and *e-Medicine*, that have been implemented by using the DLV system, and applications to *Products and services configuration* and *Decision support systems*, that have been implemented by using CLASP and SMODELS systems.

e-Tourism. ASP has been profitably applied in a couple of applications arising in the tourism industry. In the following, we overview an ASP-based application that has been integrated into an e-tourism portal, and implements an intelligent advisor that selects the most promising offers for customers of a travel agency (Ricca et al. 2010). The goal is to devise a tool that helps the employees of a travel agency in finding the best possible travel solution in a short time. It can be seen as a mediator system finding the best match between the offers of the tour operators and the requests of the tourists. The system improves the business of the travel agency by reducing the time needed to single out and sell the touristic offers, and increases the level of customer-satisfaction by suggesting the offers which match the user profile to the best. By analyzing the touristic domain in cooperation with the staff of a travel agency, a knowledge base has been specified which models the key entities that describe the process of organizing and selling a complete holiday package. In this framework, ASP has been first used as the intelligent engine of a semantic-based information-extractor (Manna, Scarcello, and Leone 2011), which analyzes the text files describing the touristic offers, extracts the relevant information (e.g., place, date, prize), and classifies them in an ontology. But the main usage of ASP in this application has been for developing several search modules that simplify the task of selecting the holiday packages that best fit the customer needs. As an example, we report (a simplified version of) a logic program that creates a selection of holiday packages in Figure 5.

Input predicate *askFor(TripKind,Period)* specifies the kind of trip requested by the customer and the period (s)he wants to travel. Predicate *touristicOffer(Offer, Place)* specifies, for each touristic offer available at the travel agency, what is the place it refers to. Predicates *placeOffer(Place, TripKind)*

```
%detect possible and suggested places
possiblePlace(Place) :- askFor(TripKind,_),
      placeOffer(Place, TripKind).
suggestPlace(Place) :- possiblePlace(Place),
      askFor(_,Period),
      suggestedPeriod(Place, Period),
      not badPeriod(Place, Period).

%select packages to suggest to the user
suggestOffer(O) :- touristicOffer(O, Place),
      suggestPlace(Place).
```

Figure 5: A program that creates a selection of holiday packages.

and *badPeriod(Place, Period)* are derived by other modules of the knowledge base and define, respectively, the places which are appropriate for a kind of trip, and the periods that should be avoided for a place (e.g., because of a bad weather). The first two rules select, respectively, possible places (i.e., the ones that offer the kind of holiday requested by the customer), and places to be suggested (because they offer the required kind of holiday in the specified period). The last rule selects, within the available holiday packages, the ones which offer a holiday that matches the original input (possible offer). This is one of the several reasoning modules that have been devised for implementing the intelligent search and integrated in the e-tourism portal (Ricca et al. 2010).

Workforce-management. In the framework of the efficient management of employees of the Gioia Tauro seaport—the largest transhipment port of the Mediterranean sea—an interesting ASP application has been developed. The problem that this application has dealt with is a form of *workforce management* problem. It amounts to computing a suitable allocation of the available personnel of the seaport such that cargo ships mooring in the port are properly handled. To accomplish this task several constraints have to be satisfied. An appropriate number of employees, providing several different skills, is required depending on the size and the load of cargo ships. Moreover, the way an employee is selected and the specific role she will play in the team (each employee is able to cover several roles according to her skills) are subject to many conditions (e.g., fair distribution of the working load, turnover of heavy/dangerous roles, employees' contract rules, etc.). To cope with this crucial problem ASP has been exploited for developing a team builder. First of all, the input—the employees and their skills—was modeled by the predicate *hasSkill(employee, skillName)*, and the specification of a shift for which a team needs to be allocated, by predicate *shift(id, date, duration)*. The skills necessary for a certain shift, by *neededSkill(shift, skill)*. Weekly statistics specifying, for each employee, the last allocation date per skill by predicate *wstat(employee, skill, lastTime)*. Employees excluded due to a management decision by *excluded(shift, employee)*. Absent employees by predicate *absent(day, employee)*, and total amount of work-

```
assign(E,Sh,Sk) | nAssign(E,Sh,Sk) :-
   hasSkill(E,Sk), employee(E,_),
   shift(Sh,Day,Dur), not absent(Day,E),
   not excluded(Sh,E),
   neededSkill(Sh,Sk),
   workedHours(E,Wh), Wh+Dur<36.
:- shift(Sh,_,_),
   neededEmployee(Sh,Sk,EmpNum),
   #count{E: assign(E,Sh,Sk)}!=EmpNum.
:- assign(E,Sh,Sk1), assign(E,Sh,Sk2),
   Sk1!=Sk2.
:- wstats(E1,Sk,LastTime1),
   wstats(E2,Sk,LastTime2),
   LastTime1>LastTime2, assign(E1,Sh,Sk),
   not assign(E2,Sh,Sk).
:- workedHours(E1,Wh1), workedHours(E2,Wh2),
   threshold(Tr), Wh1+Tr<Wh2,
   assign(E1,Sh,Sk), not assign(E2,Sh,Sk).
```

Figure 6: A program for computing teams.

ing hours in the week per employee by predicate *workedHours(employee,weekHours)*. A simplified version of the program computing teams is shown in Figure 6. [3]

The first rule is disjunctive. It generates the search space by guessing the assignment of a number of available employees to the shift in the appropriate roles. Absent or excluded employees, together with employees exceeding the maximum number of weekly working hours are automatically discarded. Then, inadmissible solutions are discarded by means of four integrity constraints: the first constraint discards assignments with a wrong number of employees for some skill; the second one avoids that an employee covers two roles in the same shift; the third implements the turnover of roles; and the fourth constraint guarantees a fair distribution of the workload. Note that only the kernel part of the employed logic program is reported here (in a simplified form), and many other constraints were developed, tuned and tested.

The user interface allows for modifying manually computed teams, and the system is able to verify whether the manually-modified team still satisfies the constraints. In case of errors, causes are outlined and suggestions for fixing a problem are proposed. E.g., if no team satisfying all constraints can be generated, then the system suggests the user to relax some constraints. In this application, the pure declarative nature of the language allowed for refining and tuning both problem specifications and ASP programs while interacting with the stakeholders of the seaport. The system, developed by a spin-off company of the University of Calabria called Exeura s.r.l, has been used by the company ICO BLG operating automobile logistics in the seaport of Gioia Tauro. Further details can be found in (Ricca et al. 2012).

Intelligent call routing. Contact centers are used by many organizations to provide remote assistance to a variety of services. Their front-ends are flooded by a huge number of telephone calls every day. In this scenario the ability of routing customers automatically to the most appropriate service brings a two-fold advantage: improved quality of service and reduction of costs.

The company Exeura developed a platform for customer profiling for phone call routing based on ASP that is called zLog (`http://www.exeura.eu/en/solution/customer-profiling/`).

The key idea is to classify customer profiles and try to anticipate their actual needs for creating a personalized experience of customer care service. Call-center operators can define customer categories, but it is very likely that these employees may not have the competence for defining categories with a traditional programming language. Thus, the definition of customer categories is carried out via a user-friendly visual interface (see Figure 7) that allows one to specify and modify categories. Once a new category has been defined, zLog automatically generates an ASP program which provides its logical encoding, and can be executed by DLV over the database to populate the category. A category's definition criteria include customer behavioral aspects, such as recent history of contacts (e.g., telephone calls to the contact center, messages sent to customer assistance) or basic customer demographics (e.g., age, residence). The latter is useful, for instance, in case of natural disasters, or type of contract. When a customer calls the call-center, he/she is automatically assigned to a category (based on his/her profile) and then routed to an appropriate human operator or automatic responder.

The zLog platform has been deployed in a production system handling Telecom Italia call-centers, and it is in actual use. Every day, over one million telephone calls asking for diagnostic services reach the call-centers of Telecom Italia. The needs are optimizing the operators assignment process, in order to reduce the average call response times, and improve customer support quality. The zLog platform can detect customer category in less than 100 ms (starting from his/her telephone number) and manage over 400 calls/sec. As a result, zLog enables huge time savings for over one million daily calls.

e-Medicine. Medical knowledge bases, resulting from the integration of several different databases, often present errors and anomalies, which severely limit their usefulness. ASP has been successfully employed in this context. In particular, a multi-source data cleaning system, based on ASP and called DLVCleaner, has been realized, which detects and automatically corrects both syntactic and semantic anomalies in medical knowledge bases (Leone and Ricca 2015), based on ontological domain descriptions and measures for string similarities (Greco and Terracina 2013). DLVCleaner automatically generates the ASP programs which are able to identify and possibly correct the errors within the medical data. DLVCleaner has been employed to clean up the data stored in the tumor registries of the Calabria Region, integrating information from several local healthcare centers, including public hospitals, private healthcare centers, fam-

[3] The full version makes use of sophisticated constructs, like weak-constraints and more complex aggregates (Alviano and Leone 2015).

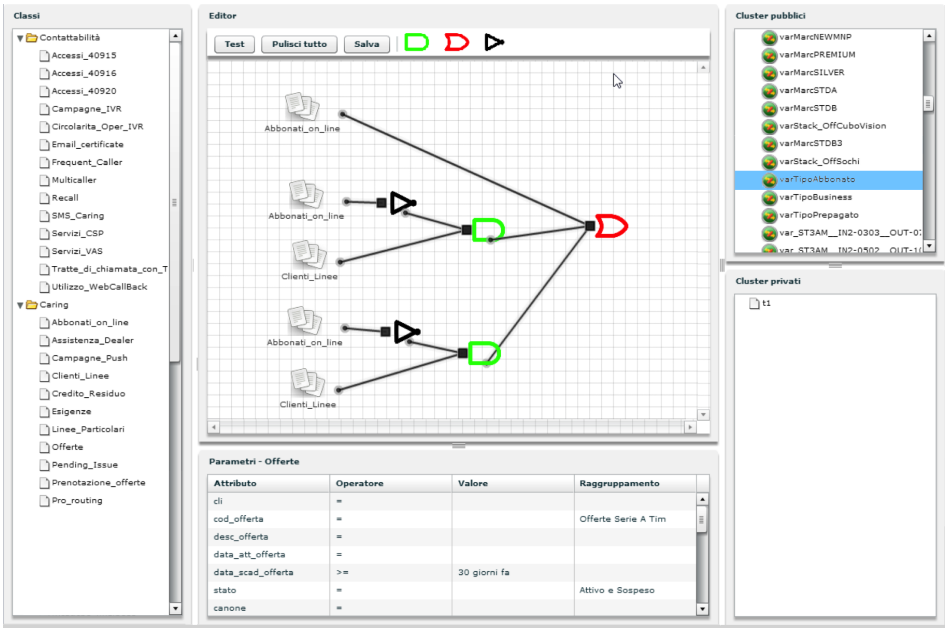

Figure 7: A visual definition of customer's categories in zLog.

ily doctors, etc. The main input table consisted of 1,000,000 tuples collecting records from 155 municipalities, whereas the dictionary stored about 15,000 tuples. DLVCleaner recognized that almost 50% of input tuples were wrong. Moreover, 72% of the wrong tuples were automatically corrected by DLVCleaner, which for an additional 26% of the tuples suggested multiple corrections to be evaluated by the user. Only 2% of input tuples have been detected as wrong and not repairable.

By using ASP in this application, a simplified and flexible specification of the logic of the data cleaning task is obtained.

(Re)configuration of products and services. One of the first industrial applications of ASP (using the ASP solver SMODELS) was for product configuration (Tiihonen, Soininen, and Sulonen 2003), used by Variantum Oy. The most recent (re) configuration applications have been carried out by the group of Gerhard Friedrich at Alpen-Adria Universität Klagenfurt, Austria, and deployed by Siemens.

In particular, ASP has been applied (with the ASP solver CLASP) for the configuration of railway safety systems in order to compute the connection structure between sensors, indicators and communication units. The task of this part of a railway safety system is to detect objects which entered but did not leave a section thus blocking a track. It turned out that this configuration problem is NP-hard and is challenging for the state-of-the-art problem solving frameworks, i.e., SAT, CSP, MIP and ASP (Aschinger et al. 2011). However, by applying ASP it was possible to solve configuration problems, which could not be solved by specialized configuration tools.

Besides configuration, the re-configuration of products and services plays in practice an important role. In many areas of configurable systems where the customer requirements change also the configured system is subject to adaptations. ASP is applied to model the possible changes of existing systems and to compute re-configuration solutions which optimize the adaptation actions. E.g., maximize the number of re-used modules and minimize the costs of additional equipment (Friedrich et al. 2011).

In addition to configuration tasks, ASP was applied to diagnose and repair systems. Friedrich et al. (2010) describes a system which computes repair-plans for faulty workflow instances employing ASP. Given the workflow structure, a set of possible repair actions and a workflow instance where an exception was triggered, a contingency plan is generated such that after the execution of this plan a correct completion of the workflow instances is achieved.

Decision support systems. ASP has been used by United Space Alliance to check correctness of plans and find plans for the operation of the Reaction Control System (RCS) of the Space Shuttle (Nogueira et al. 2001) (as briefly discussed in Example 2). The RCS is the shuttles system mainly for maneuvering the aircraft while it is in space. The RCS is computer controlled during takeoff and landing. While in orbit, however, astronauts have the primary control. During normal shuttle operations, there are pre-scripted plans that tell the astronauts what should be done to achieve certain goals. The situation changes when there are failures in the system. The number of possible sets of failures is too large to pre-plan for all of them. Meanwhile, RCS consists of fuel and oxidizer tanks, valves and other plumbing to provide propellant to the maneuvering jets of the shuttle, and it consists of electronic circuitry to control the valves in the fuel lines and to prepare the jets to receive firing commands. The actions of flipping switches have many ramifications on the

states of valves, and thus this application domain presents the further challenges of the ramification problem. Thanks to the expressivity of ASP in representing dynamic systems and handling the ramification problem (as explained in Example 2), an intelligent system has been implemented using ASP with the ASP solver SMODELS to verify and generate such pre-plans.

Some challenges addressed by ASP in industrial applications. To deal with industrial applications, ASP has to address various Software Engineering challenges. Thanks to its powerful and expressive framework, using ASP-based software development provides many advantages, such as flexibility, readability, extensibility, and ease of maintenance. Indeed, the possibility of modifying complex reasoning tasks by simply editing a text file with the ASP rules, and testing it "on-site" together with the customer, has been often a great advantage of the ASP-based development. This aspect of ASP-based software development was a success-reason especially for the *Workforce-Management* application, where the high complexity of the requirements was a main obstacle, and the availability of an executable specification language, like ASP, allowed to clarify and formalize the requirements much more quickly together with the customer.

Realizing complex features of an application in such a way also brings about advantages of lower (implementation) costs, compared to traditional imperative languages.

Another challenge in industrial applications is computational efficiency. Fortunately, there are are optimization techniques implemented in ASP solver to handle such challenges. For instance, in the *Intelligent call routing* application, an immediate response has to be given to queries over huge data sets. Thanks to the availability of the Magic-Set optimization technique (Alviano et al. 2012), DLV can localize the computation to the small fragment of the database which is relevant for the specific query at hand; using this optimization technique leads to a tremendous speed-up of the computation.

6 Conclusion

We have discussed some applications of ASP in knowledge representation and reasoning, robotics, bioinformatics and computational biology as well as some industrial applications. In these applications, ASP addresses various challenges. For instance, representation of defaults to handle exceptions, and the commonsense law of inertia to be able to reason about effects of actions are some of the important challenges in knowledge representation and reasoning. Hybrid reasoning, reasoning about commonsense knowledge and exceptions, optimizations over plans or diagnoses are some of the important challenges addressed by ASP in robotic applications. Provability of formulation of computational problems, expressing sophisticated concepts that require recursion and/or aggregates, integration of heterogeneous knowledge are some of the important challenges addressed by ASP in bioinformatics and computational biology. Similar challenges are also addressed in industrial applications, such as data cleaning, extraction of relevant knowledge from large databases, and software engineering challenges. Thanks to the expressive declarative languages of ASP that support default negation, aggregates, recursion, external atoms, consistency restoring rules and optimization statements, the presence of theoretical results that help for analysis of ASP formulations, and the sophisticated methods (like Magic Sets) implemented in the ASP solvers to improve computational efficiency, these challenges can be addressed by ASP.

7 Acknowledgments

Thanks to Gerhard Brewka, Francesco Calimeri, Wolfgang Faber, Martin Gebser, Tomi Janhunen, Volkan Patoglu, Simona Perri, Enrico Pontelli, Francesco Ricca, Torsten Schaub, Tran Son, and Mirek Truszczynski for their comments on an earlier draft of this article. The work of Esra Erdem is partially supported by TUBITAK Grants 111E116 and 114E491 (Chist-Era COACHES). The work of Nicola Leone is partially supported by the Italian Ministry of University and Research under PON project "Ba2Know (Business Analytics to Know) Service Innovation - LAB", No. PON03PE_00001_1.

References

Alviano, M., and Leone, N. 2015. Complexity and compilation of gz-aggregates in answer set programming. *TPLP* 15(4-5):574–587.

Alviano, M.; Faber, W.; Greco, G.; and Leone, N. 2012. Magic sets for disjunctive datalog programs. *Artif. Intell.* 187:156–192.

Aschinger, M.; Drescher, C.; Friedrich, G.; Gottlob, G.; Jeavons, P.; Ryabokon, A.; and Thorstensen, E. 2011. Optimization methods for the partner units problem. In *Proc. of CPAIOR*, 4–19.

Balduccini, M., and Gelfond, M. 2003. Logic programs with consistency-restoring rules. In *Proc. of Commonsense*, 9–18.

Baral, C.; Gelfond, M.; and Rushton, J. N. 2009. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming* 9(1):57–144.

Brooks, D. R.; Erdem, E.; Erdogan, S. T.; Minett, J. W.; and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *J. Autom. Reasoning* 39(4):471–511.

Chen, X.; Ji, J.; Jiang, J.; Jin, G.; Wang, F.; and Xie, J. 2010. Developing high-level cognitive functions for service robots. In *Proc. of AAMAS*, 989–996.

Dovier, A.; Formisano, A.; and Pontelli, E. 2009. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell.* 21(2):79–121.

Eiter, T.; G.Ianni; R.Schindlauer; and H.Tompits. 2006. Effective integration of declarative rules with external evaluations for Semantic-Web reasoning. In *Proc. of ESWC*.

Erdem, E.; Aker, E.; and Patoglu, V. 2012. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics* 5(4):275–291.

Erdem, E., and Oztok, U. 2015. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming* 15(1):35–78.

Erdem, E., and Türe, F. 2008. Efficient haplotype inference with answer set programming. In *Proc. of AAAI*, 436–441.

Erdem, E.; Erdem, Y.; Erdogan, H.; and Öztok, U. 2011. Finding answers and generating explanations for complex biomedical queries. In *Proc. of AAAI*.

Erdem, E.; Patoglu, V.; Saribatur, Z. G.; Schüller, P.; and Uras, T. 2013. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. *Theory and Practice of Logic Programming* 13(4-5):831–846.

Erdem, E.; Lifschitz, V.; and Ringe, D. 2006. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming* 6(5):539–558.

Erdem, E.; Patoglu, V.; and Saribatur, Z. G. 2015. Integrating hybrid diagnostic reasoning in plan execution monitoring for cognitive factories with multiple robots. In *Proc. of ICRA*, 2007–2013.

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.

Friedrich, G.; Ryabokon, A.; Falkner, A. A.; Haselböck, A.; Schenner, G.; and Schreiner, H. 2011. (Re)configuration based on model generation. *Proceedings of the International Workshop on Logics for Component Configuration* 26–35.

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

Gebser, M.; Schaub, T.; Thiele, S.; and Veber, P. 2011. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming* 11(2):1–38.

Gelfond, M., and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. New York, NY, USA: Cambridge University Press.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of ICLP*, 1070–1080. MIT Press.

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

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3.

Greco, G., and Terracina, G. 2013. Frequency-based similarity for parameterized sequences: Formal framework, algorithms, and applications. *Inf. Sci.* 237:176–195.

Havur, G.; Ozbilgin, G.; Erdem, E.; and Patoglu, V. 2014. Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach. In *Proc. of ICRA*, 445–452.

Hayes, P. J., and McCarthy, J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. Edinburgh University Press. 463–502.

Le, T.; Nguyen, H.; Pontelli, E.; and Son, T. C. 2012. ASP at work: An ASP implementation of phylows. In *Proc. of ICLP*, 359–369.

Leone, N., and Ricca, F. 2015. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, 308–326.

Liu, H., and Singh, P. 2004. ConceptNet: A practical commonsense reasoning toolkit. *BT Technology Journal* 22.

Manna, M.; Scarcello, F.; and Leone, N. 2011. On the complexity of regular-grammars with integer attributes. *J. Comput. Syst. Sci.* 77(2):393–421.

McCarthy, J. 1990. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proc. of PADL*, 169–183.

Ricca, F.; Dimasi, A.; Grasso, G.; Ielpa, S. M.; Iiritano, S.; Manna, M.; and Leone, N. 2010. A logic-based system for e-tourism. *Fundam. Inform.* 105(1-2):35–55.

Ricca, F.; Grasso, G.; Alviano, M.; Manna, M.; Lio, V.; Iiritano, S.; and Leone, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming* 12.

Shanahan, M. 1997. *Solving the Frame Problem: A Mathematical Investigation of the Commonsense Law of Inertia*. MIT Press.

Tiihonen, J.; Soininen, T.; and Sulonen, R. 2003. A practical tool for mass-customising configurable products. In *Proc. of ICED*, 1290–1299.

Tran, N., and Baral, C. 2009. Hypothesizing about signaling networks. *Journal of Applied Logic* 7(3):253 – 274.

Zhang, S.; Sridharan, M.; Gelfond, M.; and Wyatt, J. L. 2014. Towards an architecture for knowledge representation and reasoning in robotics. In *Proc. of ICSR*, 400–410.

Zhang, S.; Sridharan, M.; and Wyatt, J. L. 2015. Mixed logical inference and probabilistic planning for robots in unreliable worlds. *IEEE Transactions on Robotics* 31(3):699–713.