Answer Set Programming: A tour from the basics to advanced development tools and industrial applications

Nicola Leone and Francesco Ricca

Department of Mathematics and Computer Science, University of Calabria, Italy
{leone,ricca}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a powerful rule-based language for knowledge representation and reasoning that has been developed in the field of logic programming and nonmonotonic reasoning. After more than twenty years from the introduction of ASP, the theoretical properties of the language are well understood and the solving technology has become mature for practical applications. In this paper, we first present the basics of the ASP language, and we then concentrate on its usage for knowledge representation and reasoning in real-world contexts. In particular, we report on the development of some industry-level applications with the ASP system DLV, and we illustrate two advanced development tools for ASP, namely ASPIDE and JDLV, which speed-up and simplify the implementation of applications.

1 Introduction

Answer Set Programming (ASP) [11, 19, 30] is a powerful rule-based language for knowledge representation and reasoning that has been developed in the field of logic programming and nonmonotonic reasoning. ASP features disjunction in rule heads, non monotonic negation in rule bodies [30], aggregate atoms [16] for concise modeling of complex combinatorial problems, and weak constraints [12] for the declarative encoding of optimization problems.

Computational problems, even of high complexity [19], can be solved in ASP by specifying a logic program, i.e., a set of logic rules, such that its answer sets correspond to solutions, and then, using an answer set solver to find such solutions [38, 34].

After more than twenty years from the introduction of ASP, the theoretical properties of the language are well understood and the solving technology has become mature [13] for practical applications. The high knowledge-modeling power of ASP made it suitable for solving a variety of complex problems arising in scientific applications [13] from several areas ranging from Artificial Intelligence [2, 4, 5, 25, 39, 10, 27], to Knowledge Management [3, 6] and Databases [35, 9, 32, 7].

Recently, an ASP system, namely the DLV system [33], has undergone an industrial exploitation by a spin-off company called DLVSYSTEM l.t.d., favouring the interest of some industries in ASP and DLV, which has led to its successful usage in a number of industry-level applications [31]. A key advantage of DLV for applications development is its endowment with powerful development tools [24, 22], supporting the activities of researchers and implementors.

In this paper, after a brief introduction to the ASP standard language, we illustrate its usage for advanced Knowledge Representation and Reasoning by presenting a number of industry-level real-world applications of ASP, that we have implemented by using the DLV system and its accompanying tools. Namely:

- A platform employed by the call-centers of Italia Telecom, which automatically classifies the incoming calls for optimal routing. The platform works in real-time and deals with a very large number of parallel calls.
- A tool for the automatic generation of the teams of employees [42] that has been employed in the sea port of Gioia Tauro for intelligent resource allocation.
- A mediator system for e-tourism [41], where ASP is used to single out, in a short time, the travel solution that best matches the user profile.
- A tool for travel agents for the intelligent allotment of touristic packages. Basically, the system selects from service-suppliers blocks of touristic packages to be pre-booked for the next season in such a way that the expected earnings are maximized, and a number of preference criteria are satisfied.
- An ASP-based platform for data cleaning [44] that is part of a business intelligence suite developed for analyzing and cleaning-up the distributed archives of the Italian Healthcare System storing data on tumor diseases.

Moreover, we illustrate two advanced development tools for ASP, namely ASPIDE [24] and JDLV [22], that have played a crucial role for the successful usage of DLV in the above mentioned applications. ASPIDE is an extensible integrated development environment for ASP, which integrates powerful editing tools with a collection of development tools for program testing and rewriting, database access, solver execution configuration and output-handling. JDLV is a plug-in for Eclipse, supporting a hybrid language that transparently enables a bilateral interaction between ASP and Java. The development tools support researchers and software developers and simplify the integration of ASP in mature widely-adopted development platforms based on imperative and object-oriented programming languages.

2 Answer Set Programming

In this section we overview the language of ASP, and we recall a methodology for solving complex problems with ASP. More detailed descriptions and a more formal account of ASP, including the features of the language employed in this paper, can be found in [30, 28, 21, 12], whereas a nice introduction to ASP can be found in [3]. Hereafter, we assume the reader is familiar with logic programming conventions.

2.1 Syntax

Following a convention dating back to Prolog, strings starting with uppercase letters denote logical variables, while strings starting with lower case letters denote constants. Also *terms*, *atoms* and *literals* are defined as usual.

A *disjunctive rule* (*rule*, for short) r is a construct

$$a_1 \mid \cdots \mid a_n \text{ :- } b_1, \cdots, b_k, \texttt{ not } b_{k+1}, \cdots, \texttt{ not } b_m. \quad (1)$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0,\ m \geq k \geq 0$. The disjunction $a_1 \mid \cdots \mid a_n$ is called the *head* of r, while the conjunction $b_1, ..., b_k,\ \texttt{not}\ b_{k+1}, ...,\ \texttt{not}\ b_m$ is referred to as the *body* of r. Here `not` denotes default negation. A rule without head (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule having precisely one head atom (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a fact, and in this case the " :– " sign is usually omitted. An *ASP program* P is a finite set of rules.

In ASP, rules in programs are usually required to be safe. A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. An ASP program is safe, if each of its rules is safe, and in the following we will only consider safe programs. A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it.

Optimization problems are modeled in ASP using *weak constraints* [12]. A weak constraint ω is of the form:

$$:\sim b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m.[w@l]$$

where w and l are the weight and level of ω. (Intuitively, $[w@l]$ is read "as weight w at level l", where weight is the "cost" of violating the condition in the body of w, whereas levels can be specified for defining a priority among preference criteria). An ASP program with weak constraints is $\Pi = \langle P, W \rangle$, where P is a program and W is a set of weak constraints.

2.2 Semantics

Let P be an ASP program. The *Herbrand universe* U_P and the *Herbrand base* B_P of P are defined as usual (see e.g.,[3]). The ground instantiation G_P of P is the set of all the ground instances of rules of P that can be obtained by substituting variables with constants from U_P.

An *interpretation* I for P is a subset I of B_P. A ground literal ℓ (resp., $\texttt{not}\ \ell$) is true w.r.t. I if $\ell \in I$ (resp., $\ell \notin I$), and false (resp., true) otherwise. An aggregate atom is true w.r.t. I if the evaluation of its aggregate function (i.e., the result of the application of f on the multiset S) with respect to I satisfies the guard; otherwise, it is false.

A ground rule r is *satisfied* by I if at least one atom in the head is true w.r.t. I whenever all conjuncts of the body of r are true w.r.t. I.

A model is an interpretation that satisfies all the rules of a program. Given a ground program G_P and an interpretation I, the *reduct* [20] of G_P w.r.t. I is the subset G_P^I of G_P obtained by deleting from G_P the rules in which a body literal is false w.r.t. I. An interpretation I for P is an *answer set* (or stable model [30]) for P if I is a minimal model (under subset inclusion) of G_P^I (i.e., I is a minimal model for G_P^I) [20].

Given a program with weak constraints $\Pi = \langle P, W \rangle$, the semantics of Π extends from the basic case defined above. Thus, let $G_\Pi = \langle G_P, G_W \rangle$ be the instantiation of Π; a constraint $\omega \in G_W$ is violated by an interpretation I if all the literals in ω are true w.r.t. I. An *optimum answer set* O for Π is an answer set of G_P that minimizes the sum of the weights of the violated weak constraints in G_W as a prioritized way.

2.3 Programming Methodology

ASP has been exploited in several domains, ranging from classical deductive databases to artificial intelligence. ASP can be used to encode problems in a declarative fashion; indeed, the power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows one to obtain uniform solutions over varying instances. More in detail, many problems of comparatively high computational complexity can be solved in a natural manner by following a "Guess&Check" programming methodology, originally introduced in [18] and refined in [33]. The idea behind this method can be summarized as follows: a database of facts is used to specify an instance of the problem, while a set of (usually disjunctive) rules, called "guessing part", is used to define the search space; solutions are then identified in the search space by another (optional) set of rules, called "checking part", which impose some admissibility constraint. To grasp the intuition behind the role of both the guessing and checking parts, consider the well-known NP-complete problem 3-COLORING: given an undirected graph $G = (V, E)$, assign each vertex one of three colors -say, red, green, or blue- such that adjacent vertices always have distinct colors. 3-COLORING can be encoded in ASP as follows:

```
%Fact database specifying an instance
vertex(v).        ∀v ∈ V;        edge(i,j).        ∀(i, j) ∈ E

%Uniform non-ground program solving the problem
col(X,red) | col(X,green) | col(X,blue) :- vertex(X).        % guessing part
:- edge(X,Y), col(X,C), col(Y,C).                            % checking part
```

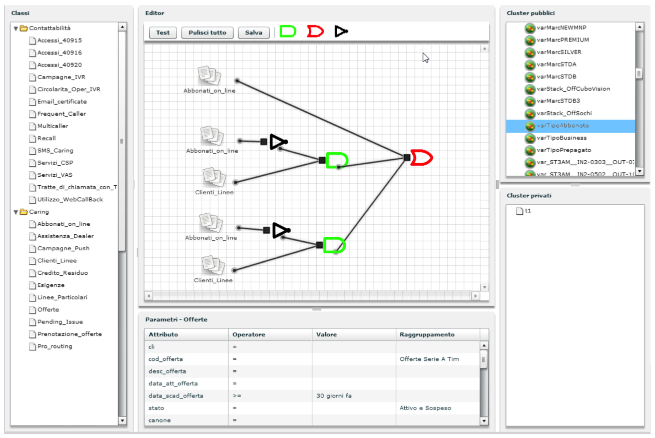
The first two lines introduce suitable facts, representing the input graph G, the third line contains a rule stating that each vertex needs to have exactly one color. The last line contains a rule that acts as an integrity constraint since it disallows situations in which two connected vertices are associated with the same color.

3 Applications

In this section we briefly describe a number of real-world applications based on ASP. These applications were implemented by using the DLV system. DLV is the first ASP system which is undergoing an industrial exploitation by a spin-off company called DLVSYSTEM l.t.d. The usage of ASP in real context outlined several advantages from a Software Engineering viewpoint of using such a powerful and expressive framework. In particular the main qualities of ASP are flexibility, readability, extensibility, ease of maintenance. A lesson learned by developing real world applications is that ASP allows one to develop complex features at a lower (implementation) price than in traditional imperative languages. Indeed, the possibility of modifying complex reasoning task by editing text files, and testing it "on-site" together with the customer has been often a great advantage of the ASP-based development.

3.1 Routing and classification of call-center customers

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.



Classi

- Contattabilità
  - Accessi_40915
  - Accessi_40916
  - Accessi_40920
  - Campagne_IVR
  - Circolarita_Oper_IVR
  - Email_certificate
  - Frequent_Caller
  - Multicaller
  - Recall
  - SMS_Caring
  - Servizi_CSP
  - Servizi_VAS
  - Tratte_di_chiamata_con_T
  - Utilizzo_WebCallBack
- Caring
  - Abbonati_on_line
  - Assistenza_Dealer
  - Campagne_Push
  - Clienti_Linee
  - Credito_Residuo
  - Esigenze
  - Linee_Particolari
  - Offerte
  - Pending_Issue
  - Prenotazione_offerte
  - Pro_routing

Parametri - Offerte

Attributo	Operatore	Valore	Raggruppamento
cli	=		
cod_offerta	=		Offerte Serie A Tim
desc_offerta	=		
data_att_offerta	=		
data_scad_offerta	>=	30 giorni fa	
stato	=		Attivo e Sospeso
canone	=		

Cluster pubblici

- varMarcNEWMNP
- varMarcPREMIUM
- varMarcSILVER
- varMarcSTDA
- varMarcSTDB
- varStack_OffCuboVision
- varMarcSTDB3
- varStack_OffSochi
- varTipoAbbonato
- varTipoBusiness
- varTipoPrepagato
- var_ST3AM__IN2-0303__OUT-0
- var_ST3AM__IN2-0502__OUT-1

Cluster privati

- t1

Fig. 1. Example of call center customer's class defined via the zLog user interface.

In this scenario the ability of routing automatically customers to the most appropriate service brings a two-fold advantage: improved quality of service and reduction of costs.

Exeura s.r.l, a spin-off company of the University of Calabria, developed a platform for customer profiling for phone calls routing based on ASP that is called zLog (`http://www.exeura.eu/en/archives/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. Contact 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 done by using an user-friendly user interface (see Figure 1) that allows to create and modify categories to be added to the call routing system in real time. Categories 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, etc.) or basic customer demographics (e.g., age, residence, etc. the latter useful, for instance, in case of natural disasters), or type of contract. When a customer calls the contact 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 customer categories specified trough the user interface are then automatically translated into ASP rules and fed as input to DLV together with the factual data extracted from the databases storing defined customer classes.

The zLog platform has been deployed in a production system handling Telecom Italia contact centers. Every day, over one million telephone calls asking for diagnostic services reach the contact 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.

We now report an example of ASP program defining a customer class extracted from a real-world scenario that is also depicted in Figure 1. The (simplified) set of rules generated by zLog corresponding to the specification of Figure 1 is the following:

```
varTipoAbbonato(CLI) :– OR1(CLI).

OR1(CLI) :– AND1(CLI).      OR1(CLI) :– AND2(CLI).
OR1(CLI) :– Abbonati_on_line1(CLI).

AND1(CLI) :– Clienti_Linee(CLI, ...), not Abbonati_on_line2(CLI).
AND2(CLI) :– Clienti_Linee1(CLI), not Abbonati_on_line2(CLI).

Abbonati_on_line1(CLI) :– Abbonati_on_line(CLI, ..., ESITO_OPSC, ESITO_TGDS, ...),
                          ESITO_OPSC="2", ESITO_TGDS="0".

Abbonati_on_line2(CLI) :– Abbonati_on_line(CLI, ..., ESITO_OPSC, ESITO_TGDS, ...),
                          DatiOPSC(ESITO_OPSC).

DatiOPSC(codifica: "11"). DatiOPSC(codifica: "12"). DatiOPSC(codifica: "13").

Clienti_Linee1(CLI) :– Clienti_Linee(CLI, ..., TIPO_CLIENTE, STATO, ...),
                       TIPO_CLIENTE="ABB", STATO="A".
```

Here it is easy to recognize that the above rules mimic the structure of the expression composed by using AND, OR, NOT operands in the graphical user interface. In particular it is defined the customer class labeled "varTipoAbbonato" (translated in English "kind of customer") outlined in blue in Figure 1. In this specification data is extracted from other customer classes, namely "Clienti_Linee", and "Abbonati_Online" representing customers that own a traditional telephone line and subscribed a contract via the Internet portal of the company, respectively. These are filtered according to some criteria on class attributes (only the relevant ones are reported shown in the program snippet) that are specified trough a specific panel of the user interface. In this case it corresponds to those that have a permanent contract (they are called "clienti in abbonamento" in Italian), but the device they are using is not known. The new class "varTipoAbbonato" is then computed applying the rules generated according to the graphical representation. zLog then exploits DLV in order to quickly compute the new class of customers.

3.2 Workforce-management in the international seaport of Gioia Tauro

The problem we dealt with in this application is a form of *workforce management* problem [37]. 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 the heavy/dangerous roles, employees' contract rules, etc.). To cope with this crucial problem DLV has been exploited for developing a team builder. First of all we modeled the input as follows:The employees and their skills by predicate *hasSkill(employee, skillName)*. The specification of a shift for which a team needs to be allocated, by predicate *shift(id, date, duration)*. The number of employees necessary for a certain skill on the shift, by *neededEmployee (shift, skill, num)*. Weekly statistics specifying, for each employee, both the number of worked hours per skill and the last allocation date by predicate *wstat(employee, skill, hours, lastTime)*. Employees excluded due to a management decision by *excluded(shift, employee)*. Absent employees by predicate *absent(day, employee)*, and total amount of working hours in the week per employees by predicate *workedHours(employee,weekHours)*. A simplified version of the program computing teams is the following:

```
(r)  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), neededEmployee(Sh,Sk,_),
                              workedHours(E,Wh), Wh + Dur ≤ 36.

(c1)  :– shift(Sh,_,_), neededEmployee(Sh,Sk,EmpNum),
                              #count{E : assign(E,Sh,Sk)} ≠ EmpNum.

(c2)  :– assign(E,Sh,Sk1), assign(E,Sh,Sk2), Sk1 ≠ Sk2.

(c3)  :– wstats(E1,Sk,_,LastTime1), wstats(E2,Sk,_,LastTime2),
                              LastTime1 > LastTime2, assign(E1,Sh,Sk),
                              not assign(E2,Sh,Sk).

(c4)  :– workedHours(E1,Wh1), workedHours(E2,Wh2), threshold(Tr),
                              Wh1 + Tr < Wh2, assign(E1,Sh,Sk),
                              not assign(E2,Sh,Sk).

(r')  workedHours(E,Wh) :– hasSkill(E,_),
                              #count{H,E : wstats(E,_,H,_)} = Wh.
```

The disjunctive rule r 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, admissible solutions are selected by means of constraints: c_1 discards assignments with a wrong number of employees for some skill; c_2 avoids that an employee covers two roles in the same shift; c_3 implements the turnover of roles; and c_4 guarantees a fair distribution of the workload. Finally, rule r' computes the total number of worked hours per employee. 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 final user interface allows to modify 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 plan 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 Exeura s.r.l, has been adopted by the company ICO BLG operating automobile logistics in the seaport of Gioia Tauro.

3.3 Advanced tools for the tourism industry

We now overview two applications of ASP to problems arising in the tourism industry. The first application is an intelligent advisor that select the most promising offers for customers of a travel agency. The second is a tool for the travel agent that helps in selecting blocks of touristic packages to pre-book during the allotment phase.

Intelligent Touristic Advisor. In [41] it is described a service based on ASP that has been integrated into an e-tourism portal. The idea is to devise a tool that helps both employees and customers 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. A knowledge base has been specified by analyzing the touristic domain in cooperation with the staff of a real touristic agency, which models the key entities that describe the process of organizing and selling a complete holiday package. In particular, all the required information, such as geographic information, kind of holiday, transportation means, etc is stored in the knowledge base. Moreover, the mere geographic information is, then, enriched by other information that is usually exploited by travel agency employees for selecting a travel destination. For instance, one might suggest avoiding sea holidays in winter; whereas, one should be recommended a visit to Sicily in summer. Also user preferences are stored, so to exploit the knowledge about users to personalize holiday package search. Then DLV has been used to develop 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) logic program that creates a selection of holiday packages:

```
%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 that the user is possibly interested in
possibleOffer(O) :– TouristicOffer(O, Place), possiblePlace(Place).
```

The first two rules select: possible places (i.e., the ones that offer the kind of holiday as input); and places to be suggested (because they offer the required kind of holiday in the specified period). Finally, the remaining rule searches in the available holiday packages the ones which offer an 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, for more details we refer the reader to [41].

Automatic Allotment. In the travel industry it is common for tour operators to pre-book from service suppliers blocks of touristic packages, which are called allotments in jargon. Basically, given a set of requirements on the properties of packages to be bought, budget limits, and an offer of packages from several suppliers, the problem from the perspective of the travel agent is to select a set of offers to be brought (or pre-booked) for the next season so that the expected earnings are maximized [15]. Despite allotment is one of the most commonly-used supplying practices in the tourism industry, the final selection of packages offered by travel suppliers is often done in small travel agencies more or less manually. Thus we developed an ASP-based tool for assisting tour operators in the allotment process. We now illustrates a simplified version of the ASP program which solves the allotment problem. In particular, the following disjunctive rule guesses a quantity to buy for each required package limiting the search space to available package tours which are requested and their selling price is in the requested range as follows:

$$\begin{aligned} \text{buy(P, Q)} \mid \text{nBuy(P, Q)} \ :- \ & \text{availablePackages(P, \_, D, T, SP, PP, \_, AvQ),} \\ & \text{requiredPackages(D, T, MinP, MaxP, ReqQ),} \\ & 0 \leq \text{Q} \leq \text{ReqQ, Q} \leq \text{AvQ, MinP} \leq \text{SP} \leq \text{MaxP.} \end{aligned}$$

The following constraint ensures only one quantity the same package is selected:

$$:- \ \text{\#count}\{\text{Q, P : buy(P, Q)}\} > 1, \text{availablePackages(P, \_, \_, \_, \_, \_, \_, \_).}$$

Here a special aggregate atom count is used see [16]. An other constraint enforces a critical requirement on the budget, i.e. the sum of prices of selected package tours must not exceed a limited budget:

$$\begin{aligned} :- \ \text{\#sum}\{ & \text{ PP*Q, P : buy(P, Q),} \\ & \text{availablePackages(P, S, \_, \_, SP, PP, \_, \_)} \} > \text{B, budget(B).} \end{aligned}$$

then earnings are maximized by using a weak constraint [12]:

$$:\sim \ \text{discountPrices(P, SP, PP), buy(P, Q), E=(SP-PP)*Q. [-E]}$$

Intuitively, when a stock of package tours is bought the violation of this constraint is associated with a cost depending on the earnings obtained by buying those packages. The weight of weak constraint is negative since weak constraints expresses the minimization of the cost associated to a solution.Travel agencies might specify a number of additional optional preference criteria that were encoded also by means of weak constraints. The ASP program is included as an advanced reasoning service of the e-tourism platform developed under the iTravelPlus project by the Tour Operator Top Class s.r.l. and the University of Calabria.

3.4 Business intelligence platform for cleaning medical archives

The approach described in the following addresses multi-source data cleaning for syntactic and semantic anomaly detection with ASP [45]. The idea is to define of an automatic procedure for generating logic programs able to identify and, whenever possible, correct errors within the data. Then, an automatically-generated logic program is embedded in a business intelligence work-flow (including data extraction, integration, manipulation and transformation) developed with Pentaho Kettle. The ASP-based solution has been implemented in a Penthao plugin called DLVCleaner. The proposed approach should be considered complementary to the existing ones, and capable to provide simplified and flexible specification of the logic of the data cleaning task. In the following we report a brief description of a real use case employing ASP for data cleaning. We refer the reader to [45] for more details on the DLV Cleaner. ASP was used to clean data from several tumor registries of the Calabria region. We first provide some background information about this scenario (from [45]). Currently no law obliges hospitals and clinics in Italy to collect and archive data on diagnosis and treatment of tumors. Then, various organizations autonomously collect such information in tumor registries. Currently, 34 tumor registries are active in Italy, covering overall almost 25% of the population. The registry used in our use case considers information related to several local healthcare centers from the Calabria Region. Data are collected from many different sources, including public hospitals, healthcare centers, family doctors, etc. Collected information include the kind of diagnosed cancer, personal data of the patient, current clinical conditions, past and current treatments, disease evolution, etc. All such information are extremely important to analyze causes and evolutions of cancer diseases, in order to study proper treatments, prevention policies, and to schedule sanitary budgets. Overall, we considered more than 200 tables as sources of data. Almost all of this information should be inter-linked by the identity of the patient. However, different registries used different schemas and standards to represent data; and such an information is often imprecise in local sources, since in many cases data are loaded manually. Thus these often contain errors or incomplete information. As a consequence, the proper identification of each mentioned patient through a subset of its attributes is a difficult and fundamental task. The entire dataset has been cleaned applying several cleaning workflows embedding several instances of DLVCleaner (in Figure 2) is reported a picture of a workflow configured for cleaning patient information). Each data flow is sent to a specifically configured DLVCleaner instance which, based on stream classifications rules specified in ASP outputs results onto one of four tables, namely *valid tuples*, *corrected tuples*, *suggested tuples*, and *anomalies*. As an example, the *DLVCleaner_3digit_birthplace* instance in Figure 2 embedds a transformation in which the birthplace is mapped onto the *nationality* attribute of the reference dictionary, whereas in *DLVCleaner_number_birthplace* the birthplace is mapped onto the *ISTAT code* dictionary attribute. Analogously, in *DLVCleaner_birthplace* the pair *(city - nation)* is handled by a matching function that first tokenizes the string, singling out the city name, and then matches it to the *city name* dictionary attribute. In order to detect potential corrections, the most proper comparison function is applied, depending on data format; as an example, for the three-digit birthplaces, we used the Hamming distance whereas for city names we used the Levenshtein distance. Setting up the workflow shown in Figure 2 takes only few minutes and it is possible to follow a try-and-error approach. Clearly, the cleaning step for birthplaces shown above is only one small step in a more complete workflow dealing with the overall database.

To give an idea of the size of the data involved in the described use case, the input table was composed of 1.000.000 tuples collecting records from 155 municipalities, whereas the dictionary stored about 15.000 tuples. From the application of the transformation shown in Figure 2 it was obtained that almost 50% of input tuples were wrong. 72% of wrong tuples have been automatically corrected, whereas 24% had multiple corrections. Only 2% of input tuples have been detected as wrong and not repairable.

3.5 Other Applications

The exploitation of DLV for developing applications is not limited to the examples reported in this section. Actually, DLV is at the basis of several other advanced applications of which it is worth mentioning data integration systems [32, 35], web data

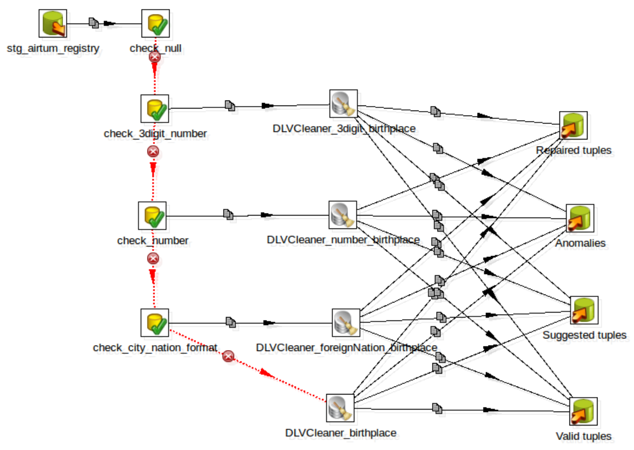


Fig. 2. Example of a Kettle workflow using the DLVCleaner plugin (from [45]).

extraction [26], and computation of minimum cardinality diagnoses [25]. Moreover, the Polish company Rodan Systems S.A. has exploited DLV in a tool for the detection of price manipulations and unauthorized use of confidential information, which is used by the Polish Securities and Exchange Commission. The company Exeura s.r.l. developed systems exploiting DLV for implementing specific modules in e-Government, e-Medicine and tele-assistance systems.

4 Development Tools

The real-world applications of DLV that we described in previous sections have demonstrated that ASP can be used to implement real-world applications. Nonetheless developers need specialized tools that make easier the development of applications, and that support the integration of different tools in the same environment. DLV is well-suited for applications development also thanks to the endowment of powerful development tools [24, 22], supporting the activities of researchers and implementors. Indeed, we endowed DLV with effective programming-tools, which are conceived to ease the usage and the integration of ASP-based technologies in the existing environments tailored for imperative/object-oriented programming languages. In the following we introduce two advanced development tools for developing ASP-based applications, namely *ASPIDE* and JDLV.

4.1 IDE for ASP

ASPIDE [24] is a complete IDE for ASP programs, which integrates an advanced editing tool with a collection of user-friendly graphical tools for program composition and execution. The user interface of *ASPIDE* is depicted in Figure 3. In the upper part of the interface a toolbar allows the user to quickly access some common operations. In the center of the interface there is the main editing area where it is possible to open several files organized in a tabbed panel. The left part of the interface is dedicated to the workspace explorer, which list projects, and to the error console, which organizes errors and warnings according to the project and files where they are localized. On the right, there are the outline panel and the template panel. The layout of the IDE is customizable, indeed the user can rearrange components the way he/she likes best.

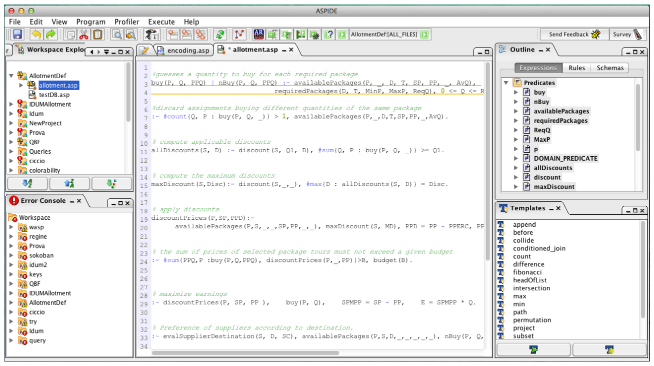
In the following we overview the main features that are available in *ASPIDE*.

Advanced Editor. The system allows for organizing logic programs in projects à la Eclipse, which are collected in a workspace. Projects collect either different parts of an encoding or several equivalent encodings solving the same problem. *ASPIDE* supports a number of file editors and can be extended to support virtually any kind of input files by user-defined plugins (which are described below). The main editor for ASP programs offers, besides the basic functionalities, such as code line numbering, find/replace, undo/redo, copy/paste, also:

– *Text coloring*. The editor performs keyword outlining (such as “:-’ and “`not` ”) and dynamic highlighting of predicate names, variables, strings, and comments.
– *Automatic completion*. The system is able to complete (on request) predicate names, as well as variable names. Predicate names are both learned while writing, and extracted from the files belonging to the same project; variables are suggested by taking into account the rule we are currently writing.
– *Refactoring*. The refactoring tool allows to modify programs in a guided way. For instance, variable renaming in a rule is done by considering bindings of variables; custom refactorings can applied by selecting rules and applying some functionality offered by a user-defined plugin.
– *Dynamic code checking and errors highlighting.* Programs are parsed while writing, and both errors or possible warnings are immediately outlined.
– *Quick fixes.* The editor suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of code.
– *Code templates. ASPIDE* provides support for assisted writing of rules (guessing patterns, etc.), as well as automated writing of entire subprograms (e.g., transitive closure rules) by means of code templates, which can be instantiated while writing.
– *Program Outline. ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files).
– *Visual editor.* The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules. The user can switch from the text editor to the visual one (and vice versa) thanks to a reverse-rengineering mechanism from text to graphical format.

Dependency Graph. *ASPIDE* creates automatically a graphical representation of several variants of the (non-ground) dependency graphs associated with a project, and can be used for analyzing rule dependencies and browsing the program.

Debugger and Profiler. *ASPIDE* embeds the debugging tool *spock* [8], and provides a graphical user interface that wraps the above mentioned tool. Regarding the profiler, *ASPIDE* fully embeds the graphical interface presented in [14].



```
%guesses a quantity to buy for each required package
buy(P, Q, PPQ) | nBuy(P, Q, PPQ) :- availablePackages(P, _, D, T, SP, PP, _, AvQ),
                                    requiredPackages(D, T, MinP, MaxP, ReqQ), 0 <= Q <= R

%discard assignments buying different quantities of the same package
:- #count{Q, P : buy(P, Q, _)} > 1, availablePackages(P,_,D,T,SP,PP,_,AvQ).


% compute applicable discounts
allDiscounts(S, D) :- discount(S, Q1, D), #sum{Q, P : buy(P, Q, _)} >= Q1.


% compute the maximum discounts
maxDiscount(S,Disc):- discount(S,_,_), #max{D : allDiscounts(S, D)} = Disc.


% apply discounts
discountPrices(P,SP,PPD):-
      availablePackages(P,S,_,_,SP,PP,_,_), maxDiscount(S, MD), PPD = PP - PPERC, PP


% the sum of prices of selected package tours must not exceed a given budget
:- #sum{PPQ,P :buy(P,Q,PPQ), discountPrices(P,_,PP)}>B, budget(B).



% maximize earnings
:~ discountPrices(P, SP, PP ),    buy(P, Q),    SPMPP = SP - PP,    E = SPMPP * Q.


% Preference of suppliers according to destination.
:~ evalSupplierDestination(S, D, SC), availablePackages(P,S,D,_,_,_,_,_), nBuy(P, Q,
```

Fig. 3. The user interface of *ASPIDE*.

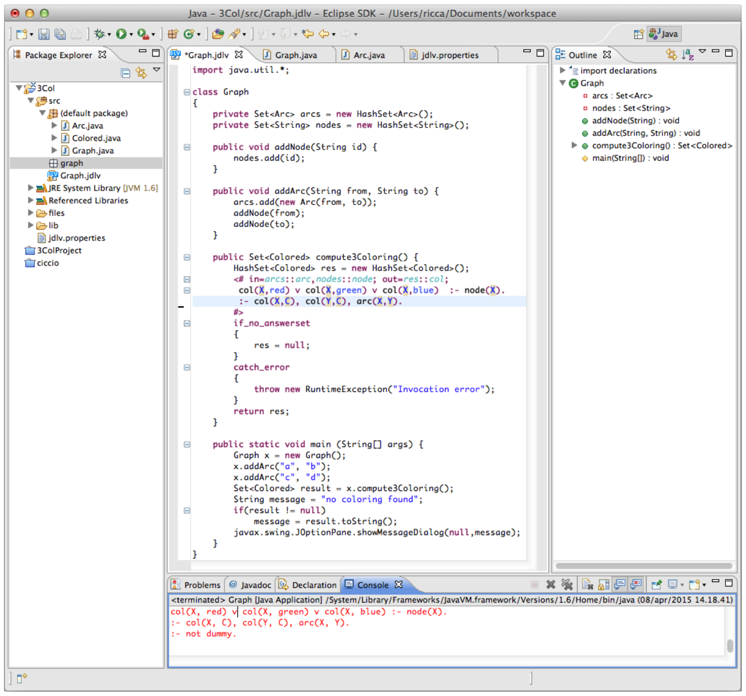
Unit Testing. In software engineering, the task of testing and validating programs is a crucial part of the life-cycle of software development process and a test conceived for verifying the behavior of a specific part of a program is called unit testing. The testing feature consists on a unit testing framework for logic programs in the style of JUnit. The developer can specify rules by composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the expected behavior of sub-programs. For an exhaustive description the testing language and the graphical tool we refer the reader to [23].

Interaction with Databases. *ASPIDE* simplifies access to external databases by a graphical tool connecting to DBMSs via JDBC. The database management feature of *ASPIDE* supports the creation of both *#import/#export* directives of DLV, and fully-graphical composition of TYP files [43]. Imported sources are emphasized also in the program editor by exploiting a specific color indicating the corresponding predicates. Database oriented applications can be run by setting DLV^{DB} as engine in a run configuration. A data integration scenario [32] can be implemented by exploiting these features.

Configuration of the execution. The execution of ASP programs is fully customizable by using the RunConfiguration Dialog that allows one to set the system executable, setup invocation options and input files. A number of shortcuts and drop down menus allows one for a quick execution of single files or selection of files within a project.

Results window. The results are presented to the user in a comfortable view combining tabular representation of predicates and a tree-like representation of answer sets. Further output extensions can be added by means of output plugins. Two examples are the ARVis comparator of answer sets [1] and the answer set visualizer IDPDraw [46].

User-defined Plugins. An important feature of *ASPIDE* is the possibility to extend it with user defined plugins. Developers can create libraries for extending *ASPIDE* with: (i) new input formats, (ii) program rewritings, and even (iii) customizing the visualization/format of results. An input plugin can take care of input files that appear in *ASPIDE* as a logic program, and an output plugin can handle the external conversion of the computed results. A rewriting plugin may encode a procedure that can be applied to



```
import java.util.*;

class Graph
{
    private Set<Arc> arcs = new HashSet<Arc>();
    private Set<String> nodes = new HashSet<String>();

    public void addNode(String id) {
        nodes.add(id);
    }

    public void addArc(String from, String to) {
        arcs.add(new Arc(from, to));
        addNode(from);
        addNode(to);
    }

    public Set<Colored> compute3Coloring() {
        HashSet<Colored> res = new HashSet<Colored>();
        <# in=arcs::arc,nodes::node; out=res::col;
         col(X,red) v col(X,green) v col(X,blue)  :- node(X).
         :- col(X,C), col(Y,C), arc(X,Y).
        #>
        if_no_answerset
        {
            res = null;
        }
        catch_error
        {
            throw new RuntimeException("Invocation error");
        }
        return res;
    }

    public static void main (String[] args) {
        Graph x = new Graph();
        x.addArc("a", "b");
        x.addArc("c", "d");
        Set<Colored> result = x.compute3Coloring();
        String message = "no coloring found";
        if(result != null)
            message = result.toString();
        javax.swing.JOptionPane.showMessageDialog(null,message);
    }
}
```

```
<terminated> Graph [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bin/java (08/apr/2015 14.18.41)
col(X, red) v col(X, green) v col(X, blue) :- node(X).
:- col(X, C), col(Y, C), arc(X, Y).
:- not dummy.
```

Fig. 4. The JDLV Eclipse plugin.

rules in the editor (e.g., disjunctive rule shifting can be applied on the fly by selecting rules in the editor and applying the mentioned rewriting). An SDK available from the *ASPIDE* web site allows one to develop new plugins.

System Availability. *ASPIDE* is written in Java and is available for all the major operating systems, including Linux, Mac OS and Windows. It can be downloaded from the system website `http://www.mat.unical.it/ricca/aspide`.

4.2 Combining Java and ASP

JDLV is a plug-in for the Eclipse platform [17], offering a seamless integration of ASP-based technologies within the most popular development environment for Java. JDLV is based on $\mathcal{JASP}$ [22], a hybrid language that transparently supports a bilateral interaction between (disjunctive) ASP and Java. A key ingredient of $\mathcal{JASP}$ is the mapping between (collections of) Java objects and ASP facts. In $\mathcal{JASP}$, Java Objects are mapped to logic facts (and vice versa) by adopting a structural mapping strategy. $\mathcal{JASP}$ exploits the same ideas of modern Object-Relational Mapping (ORM) frameworks, such as Hibernate and TopLink, where objects are saved/loaded from/to relational databases. $\mathcal{JASP}$ supports both a default mapping strategy, which fits the most common programmers' requirements, and custom ORM specifications that comply with the Java Persistence API (JPA) [40] to suit enterprise application development standards. The

$\mathcal{JASP}$ code is very natural and intuitive for a programmer skilled in both ASP and Java.

In Figure 4 is depicted the a simple $\mathcal{JASP}$ program open in the JDLV plugin that will serve as a running example. A monolithic block of plain ASP code (called *module*) is embedded in the Java method , which is executed "in-place", i.e., the solving process is triggered at the end of the module specification. In particular the program in Figure 4 defines the method *compute3Coloring()*, that contains a module to computes a 3-coloring of the given graph. Intuitively, the ASP program is enclosed within special tags ($< \# \ldots \# >$), and when *compute3Coloring()* is invoked, Java objects are transformed into logic facts, by applying an ORM strategy as specified in the module parameters. In the example Java variables *arcs* and *nodes* are mapped to corresponding predicates *arc* and *node*, respectively, whereas the local variable *res* is mapped as output variable to the predicate *col*. In this example, each string x in *nodes* is transformed in unary facts *node(x)*; similarly, each instance of *Arc* in the variable *arcs* produces a binary fact, e.g., *arc(from,to)*. These facts are input of the logic program, which is evaluated "in-place". If no 3-coloring exists, the variable *res* is set to *null*; otherwise, when the first answer set is computed, for each fact *col* contained in the solution a new object of the class *Colored* is created and added to *res*, which, in turn, is returned by the method. Here the $\mathcal{JASP}$'s default ORM strategy is applied to map one object per logic fact, which compound keys, i.e., keys made of all basic attributes, and *embedded values* for one to one associations, which naturally fits the usual way of representing information in ASP, e.g., in the example, one fact models one node. Such a mapping is inverted to obtain Java objects from logic facts, and ensures the safe creation of new Java objects without requiring value invention in logic programs. Although this strategy poses (very few) restrictions such as non-recursive type definition (e.g., tree-like structures are not admitted in $\mathcal{JASP}$*-core*), based on our experience, it is sufficient to handle common use cases. On the other hand, as we show in the following, full $\mathcal{JASP}$ language allows for custom ORM strategies specified by JPA [40] annotations. It is now clear that, $\mathcal{JASP}$ directly extends the syntax of Java such that $\mathcal{JASP}$ module statements are allowed in Java *block statements*. Concerning the syntax allowed within modules, $\mathcal{JASP}$ is compliant with the language of DLV, and also supports a number of advanced features that are mentioned in the following.

The language also features a number of additional features that further ease the development of programs, such as incremental modules, non positional notation, and database access. We refer the reader to [22] for a full account of the $\mathcal{JASP}$ language.

System Availability. JDLV is available in form of an Eclipse platform [17] plugin from `http://www.dlvsystem.com/dlvsystem/index.php/JDLV`. JDLVincludes *Jdlvc*, a compiler to generate plain Java classes from $\mathcal{JASP}$ files. The *Jdlvc* compiler produces plain Java classes which manage the generation of logic programs and control statements for the underlying solver DLV.

5 Conclusion

In this paper we have introduced ASP, and we have described some industry-level applications of the ASP system DLV. These applications confirmed the applicability of

ASP-based technologies for solving complex real-world applications. Moreover, it is worth observing that the DLV system is well-suited for applications development also thanks to the endowment of powerful development tools. In particular, we described two of them conceived for developing ASP-based applications, namely *ASPIDE* and JDLV . *ASPIDE* is an integrated development environment, supporting the entire life-cycle of logic programs development; JDLV is an implementation of $\mathcal{JASP}$, a new programming framework integrating ASP with Java.

References

1. Ambroz, T., Charwat, G., Jusits, A., Wallner, J.P., Woltran, S.: Arvis: Visualizing relations between answer sets. In: LPNMR 2013, LNCS, vol. 8148, pp. 73–78. (2013),
2. Balduccini, M., Gelfond, M., Watson, R., Nogeira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR-01. LNCS, vol. 2173, pp. 439–442. (2001)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
4. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 257–279. Kluwer (2000)
5. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: LPNMR-01. LNAI, vol. 2173, pp. 186–199. (2001)
6. Bardadym, V.A.: Computer-Aided School and University Timetabling: The New Wave. In: Burke, E., Ross, P. (eds.) PTAT'95. LNCS, vol. 1153, pp. 22–45. (1996)
7. Bertossi, L.E., Hunter, A., Schaub, T. (eds.): Inconsistency Tolerance, vol. 3300. (2005)
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is illogical captain. the debugging support tool spock for answer-set programs: System description. In: Vos, M.D., Schaub, T. (eds.) SEA 07
9. Bravo, L., Bertossi, L.: Logic programming for consistently querying data integration systems. In: IJCAI-03 . pp. 10–15 (2003)
10. Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.): ECAI 2006, 29 - September 1, 2006, Riva del Garda, Italy, Including PAIS 2006, FAIS, vol. 141. IOS Press (2006)
11. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
12. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE 12(5), 845–860 (2000)
13. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. TPLP 14(1), 117–135 (2014)
14. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A Visual Tracer for DLV. In: Proc. of SEA'09. Potsdam, Germany (Sep 2009)
15. Castellani, M., Mussoni, M.: An economic analysis of tourism contracts: Allotment and free sale*. In: Advances in Modern Tourism Research, pp. 51–85. (2007)
16. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003. pp. 847–852. Acapulco, Mexico (Aug 2003)
17. Eclipse: Eclipse (2001), `http://www.eclipse.org/`
18. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 79–103. Kluwer (2000)
19. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (Sep 1997)

20. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. (LNAI), vol. 3229, pp. 200–212. (Sep 2004)
21. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. AI 175(1), 278–298 (2011), special Issue: John McCarthy's Legacy
22. Febbraro, O., iGiovanni Grasso, Leone, N., Ricca, F.: JASP: a framework for integrating Answer Set Programming with Java. In: Proc. of KR2012. AAAI Press (2012)
23. Febbraro, O., Leone, N., Reale, K., Ricca, F.: Unit testing in aspide. CoRR abs/1108.5434 (2011)
24. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated Development Environment for Answer Set Programming. In: LPNMR 2011. (LNAI), vol. 6645, pp. 317–330. (2011)
25. Friedrich, G., Ivanchenko, V.: Diagnosis from first principles for workflow executions. Tech. rep., Alpen Adria University, Applied Informatics, Klagenfurt, Austria (2008), http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf
26. Furche, T., Gottlob, G., Grasso, G., Guo, X., Orsi, G., Schallhart, C.: Opal: Automated form understanding for the deep web. In: WWW (2012)
27. Garro, A., Palopoli, L., Ricca, F.: Exploiting agents in e-learning and skills management context. AI Communications 19(2), 137–154 (2006)
28. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . AI 138(1–2), 3–38 (2002)
29. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: ICLP/SLP 1988. pp. 1070–1080. MIT Press, Cambridge, Mass. (1988)
30. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
31. Grasso, G., Leone, N., Manna, M., Ricca, F.: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of M. Gelfond, (LNAI), vol. 6565. (2011)
32. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005. pp. 915–917. ACM Press, (Jun 2005)
33. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (Jul 2006)
34. Lifschitz, V.: Answer Set Planning. In: Schreye, D.D. (ed.) ICLP'99. pp. 23–37. The MIT Press)
35. Manna, M., Ricca, F., Terracina, G.: Consistent query answering via ASP from different perspectives: Theory and practice. TPLP 13(2), 277–252 (2013)
36. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. CoRR cs.LO/9809032 (1998)
37. Naveh, Y., Richter, Y., Altshuler, Y., Gresh, D.L., Connors, D.P.: Workforce optimization: Identification and assignment of professional workers using constraint programming. IBM Journal of Research and Development 51(3.4), 263–279 (2007)
38. Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In: Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning. pp. 72–79. Trento, Italy (1998)
39. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog Decision Support System for the Space Shuttle. In: PADL 2001. vol. 1990, pp. 169–183. (2001)
40. Oracle: JSR 317: JavaTM Persistence 2.0 (2009), `http://jcp.org/en/jsr/detail?id=317`
41. Ricca, F., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic-Based System for e-Tourism. FI 105((1–2)), 35–55 (2010)

42. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the gioia-tauro seaport. TPLP. *CUP* 12(3), 361–381 (2012)
43. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8, 129–165 (2008)
44. Terracina, G., Martello, A., Leone, N.: Logic-based techniques for data cleaning: An application to the italian national healthcare system. In: LPNMR 2013, Corunna, Spain, 15-19, 2013. Proceedings. LNCS, vol. 8148, pp. 524–529. (2013)
45. Terracina, G., Martello, A., Leone, N.: Logic-based techniques for data cleaning: An application to the italian national healthcare system. In: LPNMR 2013, Corunna, Spain, 15-19, 2013. Proceedings. LNCS, vol. 8148, pp. 524–529. (2013)
46. Wittocx, J.: IDPDraw, a tool used for visualizing answer sets (since 2009), `http://dtai.cs.kuleuven.be/krr/software/visualisation`