Visual Querying and Application Programming Interface for an ASP-based Ontology Language*

Lorenzo Gallucci[2,3] and Francesco Ricca[1]

[1] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
ricca@mat.unical.it
[2] DEIS, University of Calabria, 87036 Rende (CS), Italy gallucci@deis.unical.it
[3] Exeura S.r.l., c/o University of Calabria, 87036 Rende (CS), Italy gallucci@exeura.it

Abstract. Answer Set Programming (ASP) is a logic-based programming paradigm which has been recently exploited for solving complex real-world applications in an effective way. However, ASP systems currently miss important tools for the development of industry-level applications, such as easy-to-use graphic environments and application programming interfaces.
In this paper, we present two new tools, tailored for OntoDLP (an ASP-based ontology representation and reasoning language), which represent a step towards overcoming the above-mentioned limitations: a novel visual querying interface, which allows non-expert users to compose and run queries; and a Java API, enabling the development of software systems embedding ASP programs.

1 Introduction

Motivation. Answer Set Programming (ASP) is a novel programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such a solution [1]. The language of ASP is able to express all problems belonging to the complexity classes Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. Thus, ASP is strictly more powerful than SAT-based programming (unless some widely believed complexity assumptions do not hold), and, at the beginning, it has been profitably exploited to solve problems of high complexity from the AI field (e.g. diagnosis and planning under incomplete knowledge[4]).

Furthermore, the availability of some efficient ASP systems, like DLV [3], GnT [4], Clasp [5], NoMore++[6] and Cmodels [7], made ASP a powerful tool for developing advanced knowledge-based applications; and the viability of the approach has been confirmed by the recent applications of ASP systems for solving problems in the areas of Knowledge Management (KM), Security, and Information Integration [8].

* Supported by M.I.U.R. within the PRIN project "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and within Internationalization project "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

[4] Note that, both the above-mentioned problems are complete for the complexity class Σ_2^P or Π_2^P

However, ASP systems are far away from comfortably enabling the development of industry-level applications, mainly because they miss important tools for supporting users and programmers. In particular, friendly user interfaces are missing, and there is a lack of advanced Application Programming Interfaces (API) for implementing applications on top of ASP systems.

In this paper, we try to overcome the above-mentioned limitations by developing and implementing advanced interfaces for both users and programmers of an ASP-based system called OntoDLV [9]. OntoDLV is conceived for ontology representation and reasoning, and it is already employed in a couple of industrial applications [10, 11].

OntoDLP. An ontology is the specification of a common vocabulary by defining the meaning of terms and their relations, usually modeled by using primitives such as concepts organized in taxonomy, relations, and axioms. Ontology representation languages have become a central tool in many research areas and in particular in the field of the Semantic Web. However, in general, the most common ontology languages miss[5] "rule-based" inference mechanisms, an important feature considered indispensable for enabling agents to reason about the knowledge represented in an ontology [14].

OntoDLP [9] is a novel ontology representation language which naturally combines the reasoning power of ASP with the benefits of a set of ontology-representation constructs. In particular, the language includes, besides the concept of **relation**, the object-oriented notions of **class**, **object** (class instance), **object-identity**, **complex object**, **(multiple) inheritance**, and the concept of modular programming by means of **reasoning modules**.

A *class* can be thought of as a collection of individuals that belong together because they share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects).

Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*).

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type). As in DLP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). In this way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, OntoDLP logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

The OntoDLP language has been implemented in the OntoDLV system[9], which is a cross-platform visual development environment for knowledge modeling and ad-

[5] Even if there are some proposal combining Description Logic-based languages with rules (e.g. see [12, 13])

vanced knowledge-based reasoning. The OntoDLV system seamlessly integrates the DLV system [3] exploiting the power of a stable and efficient DLP solver.

Importantly, the strongly-typed nature of OntoDLP allowed for the implementation of a number of type-checking routines that verify the correctness of a specification on the fly, resulting in an help for the programmer.

Contribution. In this paper, we present two novel and important features of the OntoDLV system which represent a first step towards overcoming the above-mentioned limitations of ASP systems:

- an *advanced visual-quering interface*, which allows the user to formulate and run queries on OntoDLV by using an intuitive graphic interface à la QBE;
- and, an *Application Programming Interface* which enables the implementation of Java applications embedding OntoDLP ontologies and reasoning modules.

The remainder of this paper is structured as follows. In the next section, we present an informal overview of the OntoDLP language; followed, in Section 3 by a description of the OntoDLV system. After that, in Section 4 and 5, we present the visual-query interface and the OntoDLV API , respectively. Finally, Section 6 we draw our conclusions.

2 The OntoDLP Language

In this section we informally describe the OntoDLP language, a knowledge representation and reasoning language which allows one to define and to reason on ontologies.

An ontology in OntoDLP can be specified by means of *classes* and *relations*. Classes are organized in an *inheritance* (ISA) hierarchy, while the properties to be respected are expressed through suitable *axioms*, whose satisfaction guarantees the consistency of the ontology. *Reasoning modules* allow us to express rich forms of reasoning on the ontologies.

For a better understanding, we will describe each construct in a separate section and we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

OntoDLP is actually an extension of the ASP language, which has been enriched by ontology representation concepts, and hereafter we assume the reader to be familiar with ASP syntax and semantics (for further details refer to [3]).

2.1 Classes

A *class* can be thought of as a collection of individuals that belong together because they share some properties.

Classes can be defined in OntoDLP by using the the keyword **class** followed by its name, and class attributes can be specified by means of pairs *(attribute-name : attribute-type)*, where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*.

For instance, we can define the class *person* having the attributes name, age, father, mother, and birthplace, as follows:

class *person(name:string, age:integer, father:person, mother:person, birthplace:place).*

Note that, this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects[6]. Moreover, many properties can be represented by using alphanumeric strings and numbers by exploiting the built-in classes *string* and *integer* (respectively representing the class of all alphanumeric strings and the class of non-negative numbers).

In the same way, we could specify the other above mentioned classes in our domain as follows:

class *place(*name:string*).*

class *food(*name:string, origin:place*).*

class *animal(*name:string, age:integer, speed:integer*).*

Each class definition contains a set of attributes, which is called *class scheme*. The class scheme represents, somehow, the "structure" of (the data we have about) the individuals belonging to a class.

Next section illustrates how we represent individuals in OntoDLP.

2.2 Objects

Domains contain individuals which are called *objects* or *instances*.

Each individual in OntoDLP belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, with the following two facts

rome : place(name:"Rome").
john:person(name:"John", age:34, father:jack, mother:ann, birthplace:rome).

we declare that "Rome" and "John" are instances of the class *place* and *person*, respectively. Note that, when we declare an instance, we immediately give an oid to the instance (e.g. *rome* identifies a place named "Rome"), which may be used to fill an attribute of another object. In the example above, the attribute birthplace is filled with the oid *rome* modeling the fact that "John" was born in Rome; in the same way, "*jack*" and "*ann*" are suitable oids respectively filling the attributes *father*, *mother* (both of type person).

The language semantics (and our implementation) guarantees the referential integrity, both *jack*, *ann* and *rome* have to exist when *john* is declared.

[6] Attributes model the properties that *must* be present in all class instances; properties that *might* be present or not should be modeled by using relations. In other words, an attribute $(n : k)$ of a class c is a total function from c to k; while partial functions from c to k can be represented by a binary relation on (c, k).

2.3 Inheritance

OntoDLP allows one to model taxonomies of objects by using the well-known mechanism of inheritance.

Inheritance is supported by OntoDLP by using the special binary relation isa. For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in OntoDLP as follows:

class *student* ***isa*** {*person*}(
code:string,
school:string,
tutor:person).

class *employee* ***isa*** {*person*}(
salary:integer,
skill:string,
company:string,
tutor:employee).

In this case, we have that *person* is a more generic concept or *superclass* and both *student* and *employee* are a specialization (or *subclass*) of *person*. Moreover, an instance of *student* will have both the attributes: code, school, and tutor, which are defined locally, and the attributes: name, age, father, mother, and birthplace, which are defined in *person*. We say that the latter are "inherited" from the superclass *person*. An analogous consideration can be made for the attributes of *employee* which will be name, age, father, mother, birthplace, salary, skill, company, and tutor.

An important (and useful) consequence of this declaration is that each proper instance of both *employee* and *student* will also be automatically considered an instance of *person* (the opposite does not hold!).

For example, consider the following instance of *student*:

al:student(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome, code:"100", school:"Cambridge", tutor:hanna).

It is automatically considered also instance of person as follows:

al:person(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome).

Note that it is not necessary to assert the above instance.

In OntoDLP there is no limitation on the number of superclasses (i.e. multiple inheritance is allowed). We complete the description of inheritance recalling that there is also another built-in class in OntoDLP, which is the superclass of all the other classes and is called $object$ (or $\top$). For a formal description of inheritance we refer the reader to [9].

2.4 Relations

Relationships can be modeled in OntoDLP by means of *Relations*.

Relations are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes.

As an example, the relation *friend*, which models the friendship between two persons, can be declared as follows:

relation *friend(pers1:person, pers2:person).*

Like classes, the set of attributes of a relation is called *scheme*, while the cardinality of the scheme is called arity. The scheme of a relation defines the structure of its tuples (this term is borrowed from database terminology).

In particular, to assert that two persons, say "john" and "bill" are friends (of each other), we write the following logic facts (that we call tuples):

friend(pers1:john, pers2:bill). *friend(pers1:bill, pers2:john).*

Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an oid).

2.5 Axioms and Consistency

An *axiom* is a consistency-control construct modeling sentences that are always true (at least, if everything we specified is correct). They can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness.

As an example suppose we declared the relation colleague, which associates persons working together in a company, as follows:

relation *colleague (emp1:employee, emp2:employee).*

It is clear that the information about the company of an employee (recall that there is an attribute company in the scheme of the class employee) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we assert the following axioms:

(1) :– $colleague(emp1 : X1, emp2 : X2)$, `not` $colleague(emp1 : X2, emp2 : X1)$
(2) :– $colleague(emp1 : X1, emp2 : X2)$,
$X1 : employee(company : C)$, `not` $X2 : employee(company : C)$.

The above axioms states that, (1) the relation colleague is symmetric, and (2) if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

If an axiom is violated, then we say that the ontology is inconsistent (that is, it contains information which is, somehow, contradictory or not compliant with the intended perception of the domain).

2.6 Reasoning modules

Given an ontology, it can be very useful to reason about the data it describes.

Reasoning modules are the language components endowing OntoDLP with powerful reasoning capabilities. Basically, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules in OntoDLP are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem in a unique definition identified by a name).

As an example consider the following module, which allows to single out in the derived predicate *youngAndShy* the names of the persons who are less than 18 years old, and who have less than ten friends:

$$\textbf{\textit{module}}(shyFriends)\{$$
$$youngAndShy(N) :\!- P : person(name : N, age : A), A < 18,$$
$$\#count\{F : friend(pers1 : P, pers2 : F)\} < 10.\}$$

Note that, this information is implicitly present in the ontology, and the reasoning module just allows to make it explicit.

2.7 Querying

An important feature of the language is the possibility of asking queries in order to extract knowledge contained in the ontology, but not directly expressed. As in DLP a query can be expressed by a conjunction of atoms, which, in OntoDLP, can also contain complex terms.

As an example, we can ask for the list of persons having a father who is born in Rome as follows:

X:person(father:person(birthplace:place(name: “Rome”)))?

Note that we are not obliged to specify all attributes; rather we can indicate only the relevant ones for querying. In general, we can use in a query both the predicates defined in the ontology and the derived predicates in the reasoning modules.

For instance, consider the reasoning module *shyFriends* defined in the previous section, the following query asks whether there is a person whose name is “Jack” and is “young and shy”:

youngAndShy(X), X:person(name:”Jack”))?

3 The OntoDLV System

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies. We refrain from describing the implementation details of OntoDLV in this paper. Rather, we illustrate the overall OntoDLV architecture, and present the main features of the system; subsequently, in the following sections, we will describe the main components of the graphical user interface of OntoDLV.

The system architecture of OntoDLV, depicted in Figure 1, can be divided in three abstraction levels. The lowest level, named *OntoDLV core* contains the components implementing the main functionalities of the system, namely: *Persistency Manager*, *Type Checker*, and *Rewriter*. The Persistency Manager provides all the methods needed to store and manipulate the ontology components. In particular, it exploits the *Parser*

submodule to analyze and load the content of several OntoDLP text files, and a *DB Manager* submodule to implement data persistency on relational databases through Hibernate/JDBC.

The admissibility of an ontology is ensured by the Type Checker module which implements a number of type checking routines. The *Rewriter* module translates OntoDLP ontologies, axioms, reasoning modules and queries to an equivalent ASP program which runs on the DLV system, and redirects results and possible error messages to the Persistency Manager. The *Rewriter* features a number of optimization and caching techniques in order to reduce the time used by interacting with DLV. All

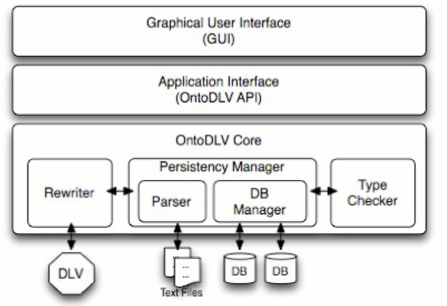

Fig. 1. The OntoDLV architecture

the features implemented by the *OntoDLV core* (data persistency, browsing invocations results etc.) can be employed by both system developers and programmers through a sophisticated application interface (which will be described in detail in Section 5): the *OntoDLV API*. Eventually, the end user exploits the system through an easy-to-use visual environment called *GUI* (Graphical User Interface), which is built on top of the *OntoDLV API*. The *GUI* combines a number of specialized visual tools for authoring, browsing and querying a OntoDLP ontology. In particular, the *GUI* features a graph-based ontology viewer and a graphical query environment (which will be described in detail in the next Section).

The OntoDLV system has been implemented in Java and exploits the DLV system, a state-of-the-art ASP solver that has been shown to perform efficiently on both hard and "easy" (having polynomial complexity) problems

The DLV system is a highly portable software written in ISO C++, available for various operating systems. Thus, the OntoDLV system runs under a variety of operating systems.

4 Visual Querying

In this section we describe the visual query interface of the OntoDLV system. This tool has been designed in a way that a non-expert user can ask queries without worrying about the syntax of the language, and a programmer can compose and test in an easy way complex queries. The query interface is integrated in the OntoDLV Graphical User

Interface. We fist report a description of the GUI, in order to give an idea of the environment in which the query tool is embedded, and then describe it by running an example.

4.1 The OntoDLV GUI

The OntoDLV GUI was designed to be simple for a novice to understand and use, and powerful enough to support experienced users. A snapshot of the system running the ontology described in Section 2 is depicted in Figure 2. The GUI presents several

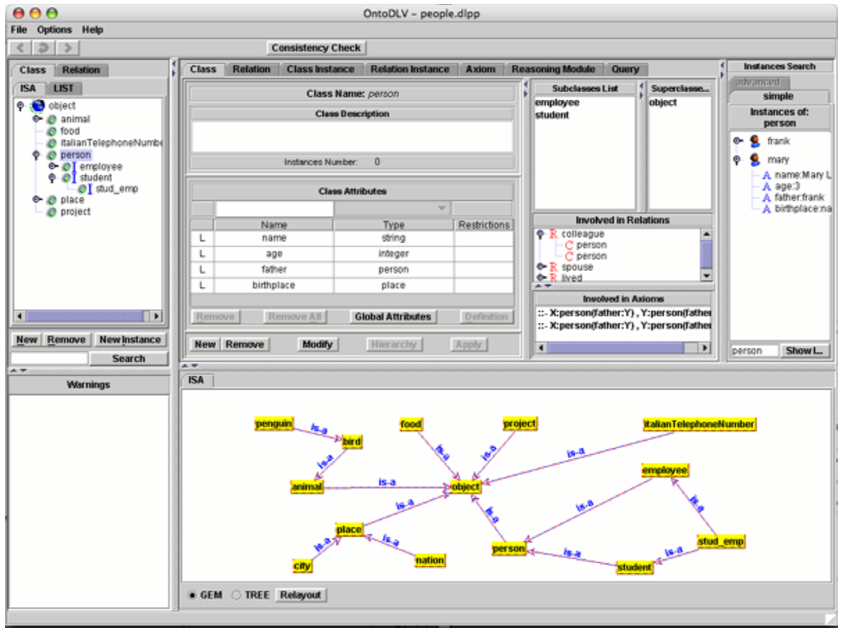

Fig. 2. OntoDLV GUI: Browsing and editing the ontology.

panels offering access to several facilities combining the browsing environment with the editing environment.

The class/subclass hierarchy is displayed both in an indented text (on the left in Figure 2) and a graph-based form (on the bottom in Figure 2).

The user can browse the ontology by double-clicking the items in the panels. The structure of each ontology entity (classes, relations, and instances) can be displayed in the middle of the screen by switching between several tabbed panels. For example, in Figure 2 the class person is selected in the class list and the class panel shows the scheme of that class. In particular, the name and the type of the class attributes are shown in a table, while, on the left, both the relations and the axioms involving the class, together with the list of the instances, are reported in an indented text form.

In the editing phase, the user enters the domain information by filling in the blanks of intuitive forms and selecting items from lists (exploiting an simple mechanism based on drag-and-drop). An up-to-date list of messages informs the user about the occurrence of errors (e.g. type checking messages, etc.) in the ontology under development. When the user clicks on an error message item the system promptly shows the entity involved in it. Reasoning and querying can be performed by selecting the appropriate panel, where the user can create/edit reasoning modules and queries, respectively. The reasoning module panel contains a text editor featuring syntax coloring and a simple auto-complete feature. The interface also allows the reasoning modality (both brave reasoning and cautious reasoning are supported) to be selected, and the reasoning modules needed to solve the specified reasoning task to be enabled/disabled.

4.2 Querying Interface

After creating or loading an ontology, the most common operation performed by users is to query the system to obtain information stored in the ontology. This task can be performed in OntoDLV by running queries through an appropriate interface[7]. Even if the OntoDLP language simplifies (w.r.t. standard ASP languages) the task of writing a query by exploiting both complex terms and strong typing, this operation may be performed by expert users only. In order to make more intuitive and easy this task, and to allow a non-expert user to query an ontology, the system features a full graphical query interface similar to the QBE (Query By Example) editors, which are nowadays largely adopted for formulating queries on relational databases. Compared to relational QBE interfaces (like, e.g., the QBE of MS Access), ours interface is more powerful thanks to the exploitation of the strong typing information of the underlying language. Thus, by using the graphical interface an user can create queries without worrying about the syntax, simply selecting classes and relations from the panels (elements can be added exploiting drag-and-drop) and creating links between class attributes and relation parameters.

In order to practically understand how the interface works, we describe it by the following example. Suppose the system already loaded the living being ontology described in Section 2, and an we want to compose the following query:

$X : person(father : person(birthPlace : place(name : "Rome")))?$

(i.e. who are the people whose father was born in a place named Rome?).

This query can be easily composed by selecting from the left panel, displaying the list of classes of the ontology (Fig. 3a), the person class, and by dragging it inside the query panel. Automatically, a box representing the person class together with its attributes (name, age, father, and birthplace, namely) appears in the panel (Fig. 3b). To complete the query we now have to indicate that the father of this person was born in a place named “Rome”. To do that, we just drag the attribute father out of the box representing the class person (Fig. 3c). The system automatically builds a list (by exploiting the strongly typed nature of the language) suggesting classes and relations that can correctly “join” with the attribute father, which is of the type person (Fig. 3d). In this case, we

[7] Due to space constraints, and since we are mainly interested in describing the graphical query editor, we refrain from describing the text-based query interface.

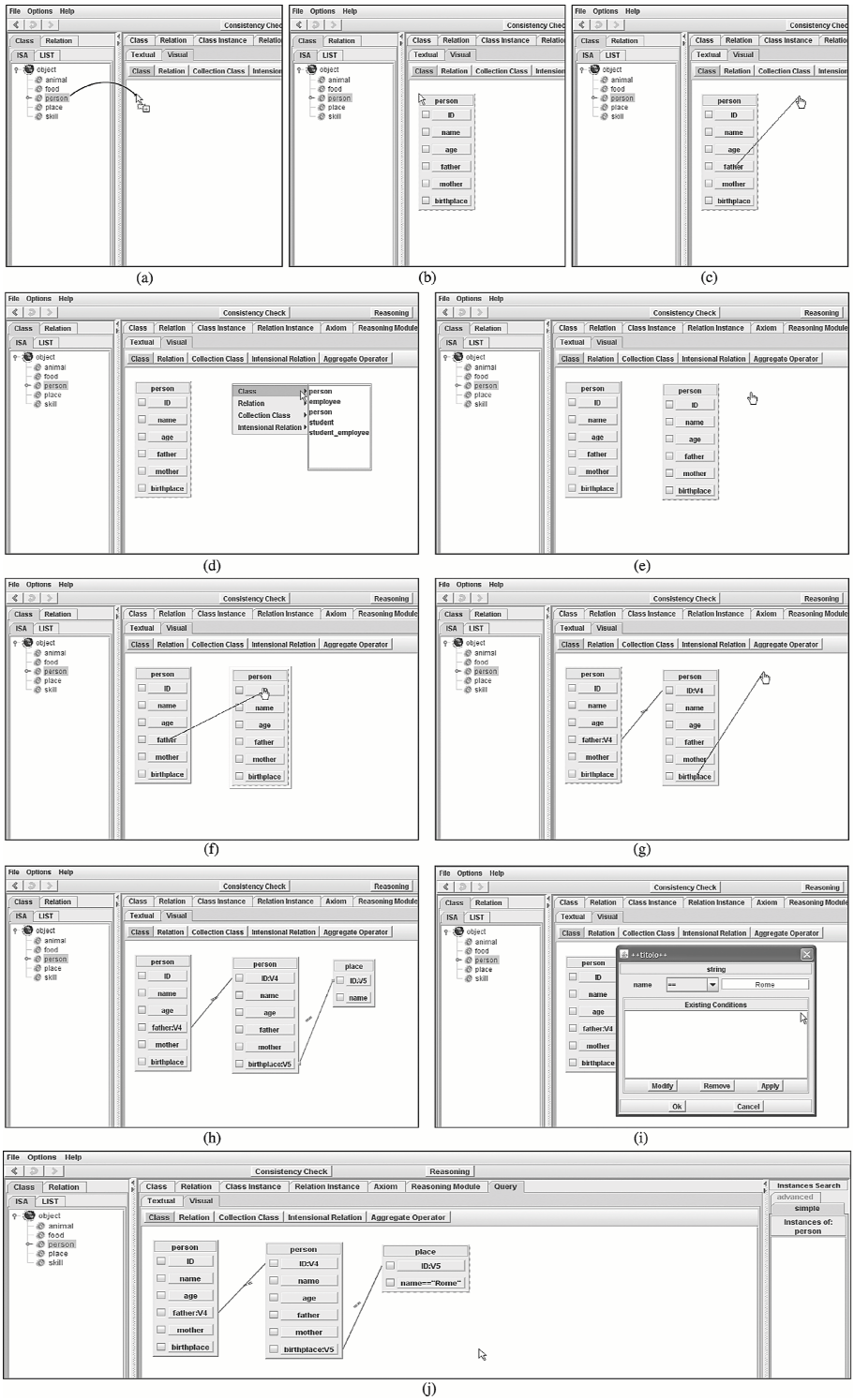

Fig. 3. OntoDLV GUI: How to build a query.

select the person class in order to indicate that the father is a person having birthplace attribute valued to rome. Consequently, another box of type person appears (Fig. 3e), and we link the oid field with the father attribute of the original person box (Fig. 3f). We continue by applying the same criterion; in particular, we drag-out (Fig. 3g) the birthplace attribute (which is of type place) of the second person box (representing the father) and we select the place class (creating a place box linked with the birthplace attribute, see Fig. 3h). Finally, we double click on the name attribute (which is of type string) of the place box to set the value of this attribute to "Rome" (Fig. 3i). The obtained query is shown in Figure 3j. It is easy to see that the graphical interface makes the meaning of that query more intuitive, and it allows an unexperienced user to work with the system without knowledge about the underlying syntax details. Importantly, the system helps the user suggesting the classes or the relation that are allowed to "join" a given attribute, exploiting the strongly-typed nature of the language. Moreover, to help expert users, a sort of "reverse-engineering" procedure allows to smoothly switch between the text editing and the visual editing environment.

5 OntoDLV API

In order to enable third parties develop their own knowledge-based applications on top of OntoDLV, we developed an application programming interface named OntoDLV API. Since OntoDLV is a Java application, the OntoDLV API has been written in this language. In particular, all the operations the user can require (e.g. creation and browsing of ontology elements, reasoner invocations etc.) are made available through a suitable set of Java interfaces. It is worth noting that, the OntoDLV API is characterized by a rather high level of abstraction; and it is composed of a relatively rich set of Java interfaces, together with a single factory class (like, e.g., the JAXP API from Sun). However, the extensive usage of standard Java components (e.g. both the interfaces $Collection$ and $Iterator$ play a central role) makes expert programmers rapidly familiar with our interface.

It is impossible, due to space constraints, to give here an in-depth description of all the methods and classes which constitute the OntoDLV API; however, in the following subsections we describe its core components and we sketch its working principles by running an example.

5.1 Core API Components and Ontology browsing

In the core part of the OntoDLV API each language construct (class schema, relation schema, instance etc.) has an associated Java interface describing it. In particular, the available interfaces are: $Class$, $Relation$, $ClassInstance$, $Tuple$, $Query$, $Axiom$, $ReasoningModule$. All the concrete objects implementing the above-mentioned interfaces are made available to the user through another interface containing a set of browsing methods called $ComponentBrowser$. In particular, $ComponentBrowser$ has seven methods which return lists of component, namely: $classes()$, $relations()$, $classInstances()$, $tuples()$, $queries()$, $axioms()$, $modules()$. The first method returns the list of all class objects, the second one the list of all relation objects and so

forth. For example, if cb is a $ComponentBrowser$, one can print out the definition of all known classes with this code:

```
for (Class cl: cb.classes()) System.out.println(cl);
```

It is worth noting that these lists are not "materializations" of the corresponding entities[8]; they rather represent virtual "views" aggregating a set of objects, possibly coming from many sources (e.g. different physical storage[9]), and they are a extensions of Java standard $Collection$s, which henceforth can manipulated using well-known Java methods such as $add()$, $contains()$, $remove()$, etc.

The same principle, based on lists of $Components$, is applied to browse the content of schemas and instances. For example, the $Class$ component has a method which returns the list of all superclasses of the given class object. Moreover, the lists returned by the browsing methods also provide the user the ability to perform *selections* over the set of objects through specialized methods. Those methods, called "selectors", return a list of the same kind as the one they were called on (cascading calls are allowed), but filtered on the basis of a given criterion.

A number of selection criteria has been designed by exploiting the properties of each collection; and, for instance, a list of classes has a set of specialized selectors that deal with the schema properties (such as $havingSubclass()$ and $havingSuperclass()$). As an example, the following code snippet allows one to print out the names of all classes (if any) which are common ancestors of both $aClass$ and $bClass$:

```
System.out.printf("Class names are: %s",
   cb.classes().havingSubclass(aClass).
   havingSubclass(bClass).names());
```

Similarly, a list of instances (namely, either $ClassInstanceList$s or $TupleList$s) may be queried for the occurrence of a particular value for an attribute by using the method $havingValue()$. For example, one can obtain the list of instances (of **any** class) having, among their attribute values, both the number 1974 and the string "Rome" (clearly, for different attributes of a given instance) in this way:

```
ClassInstanceList specialInstances =
cb.classInstances().havingValue(1974).havingValue("Rome");
```

5.2 OntoDLP API Usage

In this section, we show how to use OntoDLV API by running an example. In particular, we describe a snippet of Java code which uses the API to deal with the living being ontology introduced in Section 2. We refrain from reporting all the technical details (package inclusions, main function declaration etc.), while we focus on the part of the code where the API methods are used. In particular, we report a program which executes the following four operations:

[8] Importantly, whereas core data is always kept in memory, any information derived by the framework for internal purposes (such as collections, dependency graphs, computed attributes, etc.) is "memoized" (basically, it is stored to make the computation efficient); but, if needed, the garbage collector of the Java virtual machine can reclaim it. This allows the API to dynamically adapt the memory usage to the available system resources.

[9] As described in Secion 3 OntoDLV Core supports both filesystem and database persistency, which are handled transparently by the API

1. load a text file containing the living being ontology;
2. add some new data to the relation $friends$;
3. build the reasoning module $shyFriends$ described in Section 2.6;
4. perform the query *youngAndShy(X), X:person(name:"Jack"))?*, and print the obtained results in standard output.

To perform step 1, we first create an instance of the *Project* class, which, in general, allows one to handle many different sources of data (e.g. text files, and/or, relational databases).

```
Project project = ProjectFactory.buildEmptyProject();
```

Then, we load the "living-beings.dlpp" text by writing:

```
project.buildStreamRepository("LB",
                              new File("living-beings.dlpp"));
```

This statement, actually, creates a new *Repository* class object that handles the data stored in the "living-beings.dlpp" text file. Basically, the text file is parsed, and an in-memory representation of its content can be handled exploiting that object.

Then, we add some tuple to the relation $friends$ (step 2) by writing as follows:

```
repository.buildTuple("friend(pers1:ted, pers2:frank).");
repository.buildTuple("friend(pers1:frank, pers2:josh).");
```

In order to perform step 3, we build an object of the class *ReasoningModule*, and we add a rule within it:

```
ReasoningModule module = ontology.buildReasoningModule(
                                                    "shyFriends");
module.buildRule("youngAndShy(N) :- P:person(name:N, age:A),
    A<18, #count{ F : friend(pers1:P, pers2:F)} < 10.");
```

Eventually, we perform step 5 by building a *QueryInvocation* object as follows:

```
String queryText = "youngAndShy(X), X:person(name:"Jack"))?";
QueryInvocation queryInvocation =
project.getEngine().performQuery(queryText, DerivationMode.BRAVE);
queryInvocation.invokeSynchronously();
```

The last statement, basically, performs a synchronous invocation of the internal reasoner (i.e. the current thread it is constrained to wait until the output is computed); then we get and print the results on standard output by writing:

```
QueryResult result = queryInvocation.getResults();
System.out.printf("Results: \%s", result.toString());
```

6 Conclusions

In this paper we have presented two novel tools tailored for an integrated ontology development and reasoning platform called OntoDLV:

– a *visual query interface* à la QBE, which simplifies the usage of the system for both developers and unexperienced users;
– an *application programming interface*, which enables the programmers to embed ASP programs in systems that are based on Java.

These tools represent a step towards the development of frameworks supporting the implementation of industry-level applications based on ASP.

References

1. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
3. 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) (2006) 499–562
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007)
6. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005. LNCS 3835
7. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
8. 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: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
9. Ricca, F., Leone, N.: Disjunctive logic programming with types and objects: The dlv$^+$ system. Journal of Applied Logics (2005) To appear. `http://www.kr.tuwien.ac.at/research/reports/rr0510.ps.gz`.
10. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
11. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387
12. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary. (2003) 48–57
13. Horrocks, I., Patel-Schneider, P.F.: A proposal for an owl rules language. In: Proceedings of the 13th international conference on World Wide Web, (WWW 2004), New York, USA (2004) 723–731
14. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml (2004) W3C Member Submission. `http://www.w3.org/Submission/SWRL/`.