OntoDLV: an ASP-based System for Enterprise Ontologies*

Tina Dell'Armi[1,3], Lorenzo Gallucci[1,3], Nicola Leone[1], Francesco Ricca[1], and Roman Schindlauer[1,2]

[1] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{leone,ricca}@mat.unical.it
[2] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
roman@kr.tuwien.ac.at
[3] Exeura S.r.l., c/o University of Calabria, 87036 Rende (CS), Italy
{dellarmi,gallucci}@exeura.it

Abstract. Enterprise/Corporate ontologies are specifications of information of business enterprises. Semantic peculiarities of ASP, like the Closed World Assumption (CWA) and the Unique Name Assumption (UNA), are more appropriate than OWL assumptions for enterprise ontologies, also because these ontologies often are the evolution of relational databases, where both CWA and UNA are adopted.
In this paper we present OntoDLV, a system based on Answer Set Programming (ASP) for the specification and reasoning on enterprise ontologies.
OntoDLV implements a powerful ontology representation language, called OntoDLP, extending (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations and axioms. OntoDLP is strongly typed, and includes also complex type constructors, like lists and sets. Importantly, OntoDLV supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information also from OWL Ontologies and to exploit this information in OntoDLP ontologies and queries. The system is already used in a number of real-world applications including agent-based systems, information extraction, and text classification applications.

1 Introduction

In the last few years, the need for knowledge-based technologies is emerging in several application areas. Industries are now looking for *semantic* instruments for knowledge-representation and reasoning. In this context, *ontologies* (i.e. abstract models of a complex domain) have been recognized to be a fundamental tool; and the World Wide Web Consortium (W3C) has already provided recommendations and standards related to ontologies, like, e.g., RDF(s) [1] and OWL [2]. In particular, OWL has been conceived for the Semantic-Web, with the goal to enrich web pages with machine-understandable

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

descriptions of the presented contents (the so-called Web ontologies). OWL is based on expressive Description Logics (DL)[3]; distinguishing features of its semantics are the adoption of the Open World Assumption and the non-uniqueness of the names (the same individual can be denoted by different names).

While the semantic assumptions of OWL make sense for the Web, there are domains where they are unsuited; in particular, they are very inappropriate for Enterprise ontologies. Enterprise/Corporate ontologies are specifications of terms and definitions relevant to business enterprises; they are used to share/manipulate the information already present in a company. Since an enterprise ontology describes the knowledge of specific aspects of the "closed world" of the enterprise, a Closed World Assumption (CWA) seems more appropriate than the OWA (appropriate for the Web, which is an open environment). Moreover, the presence of naming conventions, often adopted in enterprises, guarantee names uniqueness making also the Unique Name Assumption (UNA) plausible. It is worthwhile noting that enterprise ontologies often are the evolution of relational databases, where both CWA and UNA are mandatory. To understand the better suitability for CWA and UNA for enterprise ontologies, consider the following example.

The enterprise ontology of a food-distribution company stores its pasta suppliers and their respective production branches in the relation depicted in Table 1 (of the company database).

Supplier	**Branch City**	**Branch Street**
Barilla	Rome	Veneto
Barilla	Naples	Plebiscito
Voiello	Naples	Cavour

Table 1. The Supplier-Branch table.

Consider the following query: "which are the pasta suppliers of the company having a branch *only* in Naples?". The expected answer to this query is clearly "Voiello". This answer is obtained whenever the CWA is adopted (if the world is "closed", then Voiello cannot have branches other than those specified), and computed also in the query language SQL. OWL, instead, provides an empty answer; it cannot entail that Voiello has only a branch in Naples (since, according with the OWA, Voiello could have also a branch in Rome).

To understand the role of the UNA, consider an axiom stating that each supplier has a branch only in one city. Then, a language adopting UNA derives that the ontology is inconsistent; while, OWL, missing the UNA, derives that Rome=Naples (i.e., the names Rome and Naples denote the same city!).

Similar scenarios are frequent when we deal with enterprise ontologies. In these cases logic programming languages like ASP, strongly relying on CWA and UNA, are definitely more appropriate than OWL. Answer Set Programming (ASP) [4], is a powerful logic programming language, which is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, ASP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [5]. However, ASP is, somehow, a "low-level" formalism for ontologies since in its classical formulation it does not directly support the most common ontology concepts like classes, inheritance, individuals,

etc. Moreover, 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.

This paper describes OntoDLV, an ASP-based system for knowledge modeling and advanced knowledge-based reasoning, which addresses all the above-mentioned issues.

Indeed, OntoDLV implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations, and axioms. OntoDLP is strongly typed, and includes also complex type constructors, like lists and sets.

Importantly, OntoDLV supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information also from OWL Ontologies and to exploit this information in OntoDLP ontologies and queries[4]

Moreover, OntoDLV allows one for the development of complex applications in a user-friendly visual environment; and it seamlessly integrates the DLV system [8] exploiting the power of a stable and efficient ASP solver.

Using OntoDLV, domain experts can create, modify, navigate, and query ontologies thanks to a user-friendly visual environment; and, at the same time, application developers can easily implement knowledge-intensive applications embedding OntoDLP specifications by exploiting a complete Application Programming Interface (API). Indeed, OntoDLP is already used for the development of real-world applications including agent-based systems, information extraction and text classification frameworks.

Remark 1. OntoDLV has its roots in the previous DLP+ system [9]; but, compared to its predecessor, OntoDLV brings many major new features. Among them, we recall Complex types (like Sets and Lists), Objects reclassification support (Collection Classes), Intensional Relations, OWL interoperability mechanisms, Application Programming Interface, more advanced Graphical User Interface, together with many optimization techniques, making OntoDLV well-suited for the development of industrial applications (see Section 5).

2 The OntoDLP Language

In this section, we describe OntoDLP, a language for ontology specification and reasoning. OntoDLP extends the ASP language by adding the most important ontological constructs, namely classes, attributes, relations, inheritance and axioms.

For a better understanding, we will describe each construct in a separate paragraph 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.

Hereafter, we assume the reader to be familiar with ASP syntax and semantics, for further details refer to [4, 8].

[4] It is well known that rule-based inference systems are needed by OWL applications [6, 7].

Classes. A (base) *class*[5] 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*. 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 positive integers).

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

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

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 that *jack*, *ann* and *rome* have to exist when *john* is declared.

Sets and Lists. An important feature of OntoDLP is the possibility to directly handle group of objects. More specifically, a set of *distinct* objects of a class C is considered as an individual belonging to a special class named $\{C\}$ *(set of C)*.

For instance, suppose we defined the class *car* as follows:

[5] For simplicity, we refer to *base classes* by omitting the *base* adjective, since it only distinguishes this construct from another one called *collection class* that will be described later in this Section.

[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.

class *car (brand:string, model:string, year:integer, owner:driver).*

Then, the class driver, which is a subclass of person having a special attribute *cars* of type *set of cars*, can be defined as follows:

class *driver (cars:* {*car*}*) isa* {*person*}.

Note that, the name of the class *car* has been surrounded with curly brackets to specify the class *set of cars*. An instance of the class driver can be declared as follows:

jim: driver(name:"Jim", age:35, father:jack, cars:{*herbie, kitt*}*)*

Here, *jim* is a driver that owns two cars, namely *herbie* and *kitt*, which are the elements of the set {herbie, kitt}.

In an similar way, OntoDLP allows one to specify lists of objects (of a given class) as an individual belonging to of a special class *list*. In this case, following the well-known Prolog convention, lists of elements are syntactically given in brackets.

As an example, suppose we want to model the concept of organized trip which has a set of persons (the travellers) and an ordered sequence of visited places. This can be done as follows:

class *trip (travelers:* {*person*}*, tour:[place]).*

trip1: trip(travelers:{*john, jim*}*, tour:[rome,naples,rome,pisa,milan,rome])*

Note that in a list the order of its elements is meaningful and the same individual might occur more than once. In the example, to model the trip *trip1*, we exploited a list of places to model the ordered sequence of cities visited by *john* and *jim*, and, in this case, *rome* has been visited for three times.

Sets and lists can be handled in logic rules by following the usual Prolog conventions (e.g., a list can be split in head and tail), while the language provides a set of built-in predicates that allows one to access the elements (also random access is supported), to know the size/length or to manipulate the content (by adding or removing elements).

Remark 2. Special terms, like lists and sets are not supported by conventional ASP systems, and even richer ASP variants like DLV$^+$ [9] does not provide similar language features.

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"[7] 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).

This instance is automatically considered also as an 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 latter instance.

In OntoDLP there is no limitation on the number of superclasses (i.e., multiple inheritance is allowed).

Moreover, all the set and list classes are part of the OntoDLV inheritance hierarchy. Basically, the class [*A*] (resp. $\{A\}$) is a subclass of class [*B*] (resp. $\{B\}$) if class *A* is a subclass of class *B*. For example, the class [student] (resp. {student}) is subclass of [person] (resp. {person}) since *student* is subclass of *person*.

Finally, in OntoDLP there is a common built-in superclass called $object$ (or $\top$), which has five different built-in subclasses, namely: *individual, integer, string, [object]*, and $\{object\}$. The first one is the superclass of all the user-defined classes; *integer* is the class of positive integers; *string* is the class of all alphanumeric strings; *[object]* is the generic "list of objects" (and superclass of all list classes); and $\{object\}$ is the generic "set of objects" (and superclass of all set classes).

Relations. Relationships can be modeled in OntoDLP by means of (base) *Relations*. Relations are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes, and models relationships among objects.

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

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

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).*

[7] Attribute inheritance in OntoDLP follows the same criteria exploited in in both DLP$^+$ and COMPLEX, see [9, 10] for a formal description.

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).

We complete the description of relations observing that OntoDLP allows one to organize also relations in taxonomies. In this case, relation attributes and tuples are inherited following the same criterions defined above for classes. Clearly, the taxonomies of classes and relations are distinct (classes and relations are different constructs).

Collection Classes and Intensional Relations. The notions of base class and base relation introduced above correspond, from a database point of view, to the the *extensional* part of the OntoDLP language. However, there are many cases in which some property or some class of individuals can be "derived" (or inferred) from the information already stated in an ontology. In the database world, the *views* allow to specify this kind of knowledge, which is usually called "intensional". In OntoDLP there are two different "intensional" constructs: *collection classes* and *intensional relations*.

As an example, suppose we want to define the class of persons which are less than 21 years old and have less than two friends (we name this class *youngAndShy*). This class should have a single attribute Note that this information is implicitly present in the ontology, and the "intensional" class *youngAndShy* can be defined as follows:

collection class *youngAndShy(friendsNumber: integer)* {
$X : youngAndShy(friendsNumber : N) :- \ X : person(age : Age),$
$Age < 21, \#count\{F : friend(pers1 : X, pers2 : F)\} = N, N < 2.$ }

Evidently, the attribute of the collection class ($friendsNumber$) has to be defined by the rule head. Note that in this case the instances of the class *youngAndShy* are "borrowed" from the (base) class person, and are inferred by using a logic rule. Basically, this class *collects* instances defined in another class (i.e., *person*) and performs a re-classification based on some information which is already present in the ontology. Thus, in general, the collection classes neither have proper instances nor proper oids while they "collect" already defined objects.

In an analogous way we specify "derived relations" by using the key words *intensional relation*. For example, the binary relation *relative* (modeling the common ancestry among persons) can be easily derived from the information already present in the class *person* as follows:

intensional relation *relative(sub:person, obj:person)* {
$relative(sub : X, obj : Y) :- \ X : person(father : Y).$
$relative(sub : X, obj : Y) :- \ X : person(mother : Y).$
$relative(sub : X, obj : Y) :- \ relative(sub : X, obj : Z),$
$relative(sub : Z, obj : Y).$ }

Here the first two rules populate the new intensional relation *relative* with the information about parents (X is relative of Y if X is parent of Y); while the third rule infers all the other connections (X is a relative of Y if exists another relative of X, namely Z, and Z is a relative of Y).

Importantly, the programs defining collection classes and intensional relations must be normal and stratified (see e.g., [8]). Thus, in general, *collection classes* and *intensional relations* are both more natural and more expressive than relational database views, in

fact they allow the use of the navigational style of object-oriented programming combined with a more powerful language allowing recursion and negation as failure.

Moreover, both collection classes and intensional relations can be organized in taxonomies by using the *isa* relation. It is worth noting that the inheritance hierarchy of collection classes (resp. intensional relations) and the one of base classes (resp. relations) are distinct (i.e., a collection class cannot be superclass or subclass of a base class and vice versa).

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.[8]

As an example, in our living being ontology, we may enforce that the the father cannot be younger than the son as follows:

$$:- X : person(age : A1, father : person(age : A2)), A1 > A2.$$

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).

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).

We now show an example demonstrating that OntoDLP can be easily exploited for solving complex real-world problems. Given our living being ontology, we want to compute a project team satisfying the following restrictions (i.e., we want to solve an instance of *team building problem*):

- the project team has to be constituted of a fixed number of employees;
- a given number of different skills has to be ensured inside the team;
- the sum of the salaries of the team members cannot exceed a given budget;
- the salary of each employee in the team cannot exceed a certain value.

Suppose that the ontology contains the class *project* whose instances specify the information about the project requirements, i.e., the number of team employees, the number of different skills required in the project, the available budget, and the maximum salary of each team employee:

[8] Note that the notion of axiom in OntoDLP is very different from the one employed in other ontology languages, like i.e., OWL [2]. In fact, an axiom in OntoDLP is a consistency control construct and cannot be used to specify or infer knowledge.

class $project(numEmp : integer, numSk : integer, budget : integer, maxSal : integer)$.

We can solve the above team building problem with the following module:

module$(teamBuilding)\{$
(r) $\quad inTeam(E,P) \vee outTeam(E,P) :- E : employee(), P : project().$
$(c_1) :- P : project(numEmp : N),$ `not` $\#\texttt{count}\{E : inTeam(E,P)\} = N.$
$(c_2) :- P : project(numSk : S),$ `not` $\#\texttt{count}\{Sk : E : employee(skill : Sk), inTeam(E,P)\} \geq S.$
$(c_3) :- P : project(budget : B),$ `not` $\#\texttt{sum}\{Sa, E : E : employee(salary : Sa), inTeam(E,P)\} \leq B.$
$(c_4) :- P : project(maxSal : M),$ `not` $\#\texttt{max}\{Sa : E : employee(salary : Sa), inTeam(E,P)\} \leq M. \}$

Intuitively, the disjunctive rule r guesses whether an employee is included in the team or not, generating the search space, while the constraints c_1, c_2, c_3, and c_4 model the project requirements, filtering out those solutions that do not satisfy the constraints.

In practice, reasoning modules isolate a set of logic rules and constraints conceptually related, and they exploit the expressive power of answer set programming to perform complex reasoning tasks on the information encoded in an ontology (i.e. one can solve problems which are complete for the second level of the polynomial hierarchy).

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 ASP 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 auxiliary predicates in the reasoning modules.

It is worth noting that, in presence of disjunction or unstratified negation in modules, we may obtain multiple answer sets; in this case both brave and cautious reasoning [8] are supported.

3 OWL Interoperability

As discussed above, OntoDLP is more suitable than OWL for Enterprise Ontologies, while OWL has been conceived for describing and sharing information on the Web (i.e., to deal with Web ontologies). However, it may happen that enterprise systems have to share or to obtain information from the Web; thus, from inside an enterprise ontology, one may need to access and query an external OWL ontology for specific purposes. At the same time, it is well known that Semantic-Web applications may need to integrate rule-based inference systems, to enhance their deductive capabilities. Based on these observations, our system supports a mechanisms for OWL interoperability. Actually, we

lifted the approach of [11] to the OntoDLP framework. In the following, we describe how to import OWL knowledge into OntoDLP ontologies and how this information can be exploited to write reasoning modules (and, thus, logic programs) that allow one to (somehow) add rules and reason on top of OWL.

To enable the interfacing and import of existing OWL ontologies into the framework of OntoDLP, the so-called *OWL Atoms* have been introduced. OWL Atoms can be used in rule bodies of OntoDLP's reasoning components and facilitate the evaluation of specific queries to an OWL knowledge base. This allows to import ABox data, like concept and role extensions, but also TBox information, like concept subsumption, ancestors and descendants. To comfortably handle the translation of names in this interfacing process, a *mapping* component can be specified.

OWL atoms can be used in OntoDLP constructs wherever ordinary atoms are allowed. They can contain variables and are as such also subject to the grounding of the logic program. A ground OWL atom has a truth value, depending on the evaluation of the respective query. The flow of information between an OntoDLP program is strictly uni-directional, i.e., data from ontologies is imported to the OntoDLP program. Moreover, the parameters and hence the evaluation of OWL atoms does not depend on other rules, thus they can be fully evaluated prior to any model computation procedure.

3.1 OWL Atoms

The types of queries that can be stated by an OWL atom is specified by the the DIG Description Logic Interface. The DL Implementation Group (DIG) is a self-selecting assembly of researchers and developers associated with implementations of Description Logic systems. The DIG interface allows for a number of TBox and ABox queries, returning either a truth value for boolean queries or a set of result tuples of values.

The set of constants that are imported by OWL atoms extends the set of object identifiers of the OntoDLP program. In other words, OWL Atoms can intuitively be regarded as functional queries that import new values into the OntoDLP program. Since these atoms can not occur in any recursion, the entire set of objects stays strictly finite.

An OWL query atom is characterized by the identifier $\#OWL$. It has three obligatory parameters, the query type, the query itself, and the data source:

$$\#OWL[querytype, query, source]$$

The query and source strings have to be double-quoted. The possible values for $querytype$ are those allowed in the DIG ASK directive, comprising queries such as instances of a concept, pairs of a role, subsumption of concepts, all children concepts of a concept, all types of an individual, etc. A specific query type determines the syntax of the actual query string.

The third parameter, $source$, specifies the source address of the ontology to be queried. This can be either a URI, such as "$http://www.example.org/data.owl$" or a local file, like "$/home/user/data.owl$".

For example, the OWL atom

$$\#OWL[disjoint, \text{“}Truck\ SUV\text{”}, \text{“}http://ex.org/vehicle.owl\text{”}]$$

is a purely boolean query, evaluating to true if the concepts $Truck$ and SUV are disjoint in the specified OWL KB.

The following rule imports all children classes of the concept $Mammal$ of the spcified OWL-KB:

$$mammals(X) :- \#OWL[children, \text{“}?X\ Mammal\text{”}, \text{“}http://ex.org/animals.owl\text{”}].$$

These children concept names instantiate the variable X in the respective rule. In order to be distinguishable from upperase concept or role identifiers, variable symbols within the query string are prefixed with '?'. Variables in such queries act just like variables in ordinary body atoms, being bound to a specific extension, with the difference that the extension is not determined within the program itself, but by an external evaluation.

The next rule imports the extension of a class into the OntoDLP program:

$$projects(P) :- \#OWL[instances, \text{“}projects(?X)\text{”}, \text{“}http://ex.org/dep.owl\text{”}].$$

The following collection class gathers all red parts together with their prices. Note that the part object and its price stems from OntoDLP itself, while the color information is derived from an external ontology.

collection class *redParts(price: integer)* {
$X : redParts(price : P) :- \ X : part(price : P),$
$\#OWL[relatedIndividuals, \text{“}hasColor(?X, red)\text{”}, \text{“}inventory.owl\text{”}].$ }

3.2 Name Mappings

Mappings ease the syntactic translation of constant names when they are imported into the OntoDLP program. A mapping is defined via the `mapping` keyword. It is used like a module:

$mapping\ family$ {
'dad' '$father$'
'mom' '$mother$'
}

If this mapping is specified in a query atom, each occurrence of '$father$' resp. '$mother$' in the query answer (i.e., data that comes from an OWL ontology) is translated to the name 'dad' resp. 'mom' in the OntoDLP program. The mapping specification itself is a list of pairs of strings. The first string in each pair is the local (i.e., in OntoDLP) name to be translated, the second string is the ontology name. The mapping-name is used to refer to a name mapping within a query atom, where one or more mappings can be optionally specified:

$$\#OWL[relatedIndividuals, \text{“}fatherOf(?X, ?Y)\text{”}, \text{“}family.owl\text{”}][family]$$

Thus, mappings are always local to a specific query-atom.

Mappings are not functional, hence they can be seen as $n : n$ relations. Consequently, one name can be mapped to multiple replacement names, which will all be inserted, and multiple names can be mapped to the same single replacement name. It is in the

responsibility of the author of a mapping to consider the effect of such mappings on the Unique Name Assumption.

More than one mapping can be specified in a single query atom, such as:

$$\#OWL[instances, \text{“}Person(?X)\text{”}, \text{“}people.owl\text{”}][persons, family]$$

Since mappings are not functional, simply their union comes into effect. However, the user will have the possibility to specify a command line switch which enforces unique mappings and applies a priority relation in case of conflicting mappings: the one further left has priority, as shown in the following example. Consider the following mappings:

$mapping\ persons\ \{$
‘$ciccio$’ ‘$http$://$mat.unical.it/\#Ricca$’
‘gb’ ‘$http$://$mat.unical.it/\#Ianni$’
$\}$
$mapping\ family\ \{$
‘$gibbi$’ ‘$http$://$mat.unical.it/\#Ianni$’
$\}$

and the query atom above. Assume that the user specifically requested functional mappings. Considering the namespaces, two conflicting mappings for the external name ‘$http$://$mat.unical.it/\#Ianni$’ exist. Since the mapping $persons$ is specified before $family$, the name will be translated into ‘gb’.

4 The OntoDLV System

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies. We now illustrate the OntoDLV architecture, and present the main features of the system. We refrain from giving an in-depth description of all the technical details underlying the implementation of OntoDLV in this paper, rather we will describe the main components of the system.

System Architecture and Implementation. The system architecture of OntoDLV, depicted in Figure 1a, 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 [8][9]; either the results or possible error messages are redirected to the Persistency Manager. Importantly, *ontologies* are translated into an equivalent (stratified) ASP program which is solved by DLV in polynomial time (under data

[9] DLV is a state-of-the-art ASP solver that has been shown to perform efficiently on both hard and “easy” (having polynomial complexity) problems

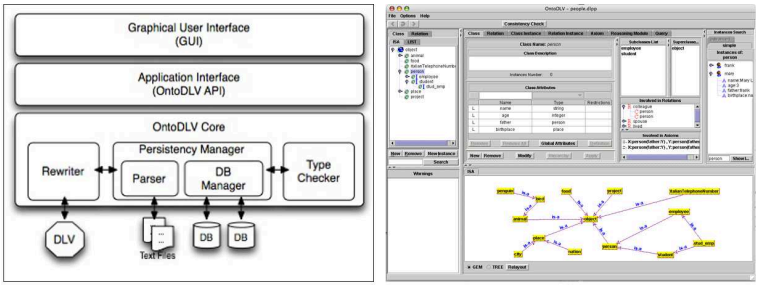

Fig. 1. The OntoDLV architecture (a) and the OntoDLV GUI (b).

complexity). Moreover, the *Rewriter* features a number of optimization and caching techniques in order to reduce the time spent interacting with DLV.

The OntoDLV system has been implemented in Java and all the features in the *OntoDLV core* can be employed by both system developers and programmers through a sophisticated application interface: the *OntoDLV API*.

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,[10] it is composed of a relatively rich set of interfaces extensively exploiting standard Java components (e.g., both the interfaces $Collection$ and $Iterator$ play a central role). This feature makes expert Java programmers rapidly familiar with the OntoDLP API.

The Graphical User Interface. The end user exploits the system through an easy-to-use and intuitive 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.

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 1b. The GUI presents several 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 1b) and a graph-based form (on the bottom in Figure 1b).

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 1b 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

[10] The design principles makes the OntoDLP API similar to the JAXP API from Sun.

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. If the user clicks on an error message the system promptly shows the entity involved in it.

Reasoning and querying can be done by selecting the appropriate controls. In particular, the reasoning module panel contains a text editor featuring syntax coloring and a simple auto-complete feature; while, queries can be created by exploiting a powerful and intuitive query environment. Indeed, the query panel combines a textual editor with a graphical "QBE-like" interfacewhich allows one to create queries without worrying about the syntax. Moreover, the interface is conceived to smoothly switch between the text editor and the visual editor (the system exploits a reverse-engineering procedure which "translates" queries from textual to graphic representation and vice versa).

5 Current Applications and Conclusion

In this paper, we have presented OntoDLP, an extension of disjunctive logic programming with relevant object-oriented constructs, including classes, objects, (multiple) inheritance, lists, sets, and types. We have described the syntax of OntoDLP and shown its usage for ontology representation and reasoning by example.

The features of the language, like the closed world assumption and its rich set of tools for ontology specification and reasoning, combined with an high computational power (allowing for the direct implementation of complex problem-solving tasks like planning, team building, etc.) make OntoDLP very suitable for dealing with Enterprise/Corporate ontologies. Moreover, OntoDLV supports a powerful interoperability mechanism with OWL, allowing one to simultaneously deal with both OWL and OntoDLP ontologies.

Importantly, we have provided a concrete implementation of the language: the OntoDLV system. OntoDLV features both an advanced Graphical User Interface (GUI) and an Application Programming Interface (API). This way, both the novice and expert user can exploit the system for solving problems and developing real-world applications based on OntoDLP. The system is built on top of DLV (a state-of-the art ASP system), and it implements all features of OntoDLP. Moreover, it combines an advanced visual-interface and a powerful type-checking mechanism for fast ontologies specification and errors detection.

OntoDLV is based on the DLP+ system [9], but its language supports more advanced features, like Complex types (like Sets and Lists), Objects reclassification support (Collection Classes) Intensional Relations, and OWL interoperability mechanisms. Moreover, the OntoDLV system provides a more advanced user interface (w.r.t. he one of DLP$^+$) and, importantly, OntoDLV includes an API that makes it ready for the development of knowledge-based applications. Indeed, even though OntoDLP has been released only very recently, it is already employed, playing a central role in advanced applications like:

– *HiLEx* [12], an advanced tool for semantic information-extraction from unstructured or semi-structured documents. Here, an OntoDLP ontology is used to represent concepts of the documents domain, while a set of "semantic" regular expressions (HiLEx expressions) represent ways of writing a concept in a document. The extraction is achieved by rewriting such expressions in OntoDLP (by exploiting modules

and collection classes) and computing the answer sets of the obtained OntoDLP specification.

– *OLEX* (OntoLog Enterprise Categorizer System) [13], is a system developed by Exeura s.r.l. (`http://www.exeura.it`) for text classification (the task of assigning to each concept of a given ontology all documents that are recognized to be relevant for it). Roughly, sets of documents are automatically classified by the system (w.r.t. a given ontology) by using suitable reasoning modules.
– The *RAP platform*, developed by Orangee (`http://www.orangee.com`) an agent-based system, implemented by using the JADE Framework, for the governance of the distribution process of antiblastic medicines in hospitals. Basically, in this application, the "agent's brain" is an OntoDLP program.

Ongoing work concerns the enhancement of OntoDLV by extending its language with new features such as optional attributes, concrete data-types, and a more powerful kind of reasoning module.

References

1. W3C: The resource description framework. (2006) `http://www.w3.org/RDF/./`.
2. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) `http://www.w3.org/TR/owl-guide/`.
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
4. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
5. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
6. 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
7. 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/`.
8. 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
9. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics **5**(3) (2007) 545–573
10. Greco, S., Leone, N., Rullo, P.: COMPLEX: An Object-Oriented Logic Programming System. IEEE TKDE **4**(4) (1992)
11. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada. (2004) 141–151 Extended Report RR-1843-03-13, Institut für Informationssysteme, TU Wien, 2003.
12. 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)
13. Curia, R., Ettorre, M., Iiritano, S., Rullo, P.: Textual Document Per-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)