Interoperability Mechanisms for Ontology Management Systems*

Lorenzo Gallucci, Giovanni Grasso, Nicola Leone, Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{gallucci|grasso|leone|ricca}@mat.unical.it

Abstract. The Ontology Web Language (OWL) is a W3C recommendation which has been conceived to enrich Web pages with machine-understandable descriptions of the semantics of the presented contents (i.e. Web Ontologies). OWL is based on the Open World Assumption (OWA), and indeed, it is suitable for describing and sharing Web information. However, OWL is unpractical in some cases (for example in applications dealing with Enterprise Ontologies) where different assumptions, like the Closed World Assumption (CWA), are better suited. Conversely, the OntoDLP language, an extension of Disjunctive Logic Programming with the most important constructs of ontology specification languages, is based on the CWA and can be fruitfully exploited in such cases. Nevertheless, it may happen that enterprise systems have to share or obtain information from the Web. This means that suitable interoperability tools are needed.
In this paper we present a pragmatic approach to the interoperability between OWL and OntoDLP. Basically, we provide a couple of syntactic transformations that allow one to import an OWL ontology in a corrispondent one specified in OntoDLP and vice versa. The proposed technique is based on an intuitive translation and preserves the semantics of the original specification when applied to interesting fragments of the two languages.

1 Introduction

The OWL (Ontology Web Language) [1] is a recommendation of the World Wide Web Consortium (W3C) which is playing an important role in the field of Semantic Web. Indeed, OWL has been conceived to enrich Web pages, now presenting information for humans, with machine-readable content. Thus, it aims at providing a common way for specifying the semantic content of Web information (i.e. *Web Ontologies*). OWL is built on-top of other preexisting Web languages, such as XML [2] and RDF(S) [3], and, its semantics is based on expressive Description Logics (DL)[4]. Essentially, the idea behind all the DLs (and, consequently, behind OWL) is the one of representing relationships among concepts

* 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."

of a domain. Thus, the basic constructs of DLs allows one to specify restrictions on relationships, like e.g. $\forall R.C$ denotes the class of all individuals that are in the relationship R with only individuals belonging to the concept C. The semantics of DLs are usually given by exploiting a set-theoretic interpretation of concepts, i.e. a concept is interpreted as a set of individuals and roles (which are binary relationships among individuals) are interpreted as sets of pairs of individuals. The non-finiteness of the domain of interpretation and the Open World Assumption (OWA) are distinguishing features of Description Logics with respect to other modeling languages which have been developed in the study of non-monotonic reasoning, logic programming, and databases. These characteristics makes DLs, and thus OWL, particularly well suited for defining and sharing data in the Web, i.e. for defining *Web Ontologies*. However, there are many other application domains in which the data belongs to more conventional and structured sources (such as relational databases), and in which the specification of the knowledge must be considered complete. For example, this happens very often when enterprises share data for business purpose; a typical scenario is the one of banks or insurance companies sharing information about behavior and/or reliability of customers. In this scenario, *Enterprise Ontologies* (i.e ontologies containing specifications of terms and definitions relevant to business enterprises), are often used for sharing information stored in relational databases. It is straightforward to see that, in this settings, (i) the nature of the domain makes the CWA an essential tool; and, (ii) rule-based formalisms (permitting sophisticated forms of reasoning and querying) can be fruitfully applied. Thus, OWL may be unpractical to use and, conversely, OntoDLP [5] a novel ontology representation language based on Answer Set Programming (ASP) [6, 7], which presents both the above-mentioneed features, can be fruitfully applied. Indeed, OntoDLP adds to an expressive logic programming language the most common ontology specification constructs (e.g. classes, relations, inheritance, axioms etc.); and more generally, OntoDLV can be suitably used when one has to deal with domains requiring the CWA, and/or the the reasoning capabilities of an expressive rule-based language.

It is worth noting that, even if the world of enterprise ontologies and the Web have very different requirements (e.g. open vs closed world assumption) it may happen that the enterprise applications need to exploit information made available in a Web site and vice versa. In this scenario, it is important to provide *interoperability* tools which make the systems able to simultaneously deal with both OWL and OntoDLP ontologies, or, at least, they must be able to *share* and/or *exchange* the information expressed in one of the two formalism with the other and vice versa.

In this paper, we propose a *pragmatic* approach to the problem of interoperability between a OntoDLP and OWL. In particular, we designed a general strategy (i) for "importing" an OWL ontology in OntoDLP; and (ii) for "exporting" an OntoDLP specification in an OWL one. We obtained an *import* and an *export* transformations which tries to "translate" each construct of the original language in an *intuitively* equivalent one on the destination language (for some

constructs the obtained behavior is only intuitively approximated). Moreover, we studied the theoretical properties of the above-mentioned transformations and we identified some fragments of the languages for which the *semantic equivalence* between the original OWL (resp. OntoDLP) ontology and the obtained OntoDLP (resp. OWL) ontology is guaranteed (i.e. the consequences entailed by the original specification under the source language semantics are the same of the ones entailed by the obtained specification under the destination language semantics).

The rest of the paper is organized as follows: Section 2 briefly describes the OntoDLP language; Section 3 presents the import and export transformations; Finally, in Section 5 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. 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.

OntoDLP is actually an extension of (disjunctive) Answer Set Programming under the stable model semantics, and hereafter we assume the reader to be familiar with ASP syntax and semantics (for further details refer to [7]).

Classes. A (base) *class*[1] 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 objectsMoreover, many properties can be represented by using alphanumeric strings

[1] For simplicity, we often refer to *base classes* by omitting the *base* adjective, which has the sole purpose of distinguishing this construct of the language from another one called *collection class* that will be described later in this section.

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).*

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.

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

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).

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). Moreover, there are two more built-in classes in OntoDLP. The first one, called *individual*, is the superclass of all the user-defined classes; and the second one, called *object* (or $\top$), is the only (direct) superclass of *individual*, *string*, and *integer* (thus, *object*is the most general OntoDLP type).[2]

We complete the description of inheritance recalling that OntoDLP allows one to organize also relations in taxonomies. In this case, relation attributes and tuples are inherited following the same criteria defined above for classes. Clearly, the taxonomies of classes and relations are distinct (class 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* allows 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 peoples which are less than 21 years old and have less than two friends (we name this class *youngAndShy*). Note that, this information is implicitly present in the ontology, and the "intensional" class *youngAndShy* can be defined as follows:

[2] For a formal description of inheritance we refer the reader to [5].

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

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 oid's while they "collect" already defined objects.

In an analogous way we specify *"intensional relations"* by using the key words **intensional relation** followed by a logic program defining its tuples. It is worth noting that the programs which define collection classes and intensional relations must be normal and stratified ([7]).

Thus, in general, *collection classes* and *intensional relations* are both more natural and more expressive than relational database views, because they exploit a powerful language that allows recursion and negation as failure.

It is important to say that both collection classes and intensional relations can be organized in taxonomies by using the *isa* relation. Importantly, 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). Axioms 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) $:- \ colleague(emp1 : X1, emp2 : X2), \texttt{not}\ colleague(emp1 : X2, emp2 : X1)$
(2) $:- \ colleague(emp1 : X1, emp2 : X2),$
$X1 : employee(company : C), \texttt{not}\ X2 : employee(company : C).$

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)[3].

[3] Note that, OntoDLP axioms are consistency control constructs (intended for *constraining* the ontology to the intended specification) and, thus they are radically different from OWL axioms.

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 an ASP 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).

As an example, suppose that the living being ontology contains a base relation that models the roads connecting neighbor places. The following module can be used to know wethere exists a route connecting two places.

$$\begin{array}{l}\textbf{\textit{module}}(connections)\{\\ \quad route(X,Y) \text{ :- } road(start:X, end:Y).\\ \quad route(X,Y) \text{ :- } road(start:X, end:Z), route(Z,Y).\ \}\end{array}$$

The above rules basically computes the transitive closure of the relation *road*; moreover, the "ouptut" relation *route* has been used in this reasoning module, without the need of defining its scheme. We did not declare the (auxiliary) predicate *route* in the ontology, because it is related only to this computation (we simply used it) Note that, the information about routes is implicitly present in the ontology and the reasoning module just allows to make it explicit.

It is worth noting that, since reasoning modules have the full power of (disjunctive) Answer Set Programming, they can be also used to perform complex reasoning tasks on the information contained in an ontology. In practice, they allow one to 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 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 auxiliary predicates in the reasoning modules. For instance, consider the reasoning module *connections* defined in the previous section, the query *route(rome,milan)?* asks whether there is a route connecting Rome and Milan.

It is worth noting that in presence of disjunction or unstratified negation in modules, we might obtain multiple answer sets; in this case the system supports both brave and cautious reasoning (see [5]).

3 Interoperability between OWL and OntoDLP

In this section we describe a pragmatic strategy that allows one to import an OWL-DL [1][4] ontology in OntoDLP, and to export an OntoDLP ontology in OWL. Hereafter, we assume the reader to be familiar with with both syntax and semantics of OWL[1] and Description Logics[4].

3.1 Importing OWL in OntoDLP

In the following we provide a description of the *import* strategy by exploiting some examples. Each group of OWL constructs is described in a separate paragraph.

OWL Thing ($\top$) and OWL Nothing ($\bot$). The OWL universal class *Thing* corresponds to the OntoDLP class *individual* (because both are the set of all individuals). Conversely, in OntoDLP we cannot directly express the empty class $\bot$, but we approximate it as follows:

class Nothing. ::– X:*Nothing()*.

Note the axiom imposes that the extension of *Nothing* is empty.

Atomic classes and class axioms (C, $C \sqsubseteq D$). Atomic classes are straightforwardly imported in OntoDLP. For example, we write: **class** $Person()$to import the specification of the atomic class *Person*.

Inclusion axioms directly correspond to the *isa* operator in OntoDLP. Thus, the statement $Student \sqsubseteq Person$ (asseting that student is a subclass of person) is imported by writing: **class** Student **isa**Person.

In OWL one can assert that two or more atomic classes are equivalent (i.e. they have the same extension) by using an equivalent class axiom ($\equiv$). OntoDLP does not have a similar construct, but we can obtain the same behavior by using collection classes and writing suitable rules to enforce the equivalence. For example, $USPresident \equiv PrincipalResidentOfWhiteHouse$ is imported as follows:

collection class *USPresident* {
X:*USPresident()* :– X: *PrincipalResidentOfWhiteHouse()*. }
collection class *PrincipalResidentOfWhiteHouse* {
X:*PrincipalResidentOfWhiteHouse()* :– X:*USPresident()*. }

Another class axiom provided by OWL, called *disjointWith*, asserts that two classes are disjoint. We approximate this behavior by using an axiom in OntoDLP. For example:

$Man \sqcap Woman \sqsubseteq \bot$

[4] OWL-DL is the largest *decidable* fragment of OWL which directly corresponds to a powerful Description Logics.

in represented in OntoDLP using the axiom:

::– X:*Man()*, X:*Woman()*.

which asserts that an individual cannot belong to both class *Man* and class *Woman*.

Enumeration classes $\{a_1, ..., a_n\}$. A class can be defined in OWL by exhaustively enumerating its instances (no individuals exist outside the enumeration).

For example, if we model the RGB color model as follows:

$RGB \equiv red, green, blue$

we will import it in OntoDLP by using a collection class in this way:

collection class *RGB* { *green* : RGB(). *red* : *RGB()*.
green : *RGB()*. *blue* : *RGB()*. }

and we also add to the resulting ontology, the axiom ::– #`count`{ *X: X:RGB()* } > 3. in order to correctly fix the number of admissible instances of the class.

Properties and Restrictions ($\forall$, $\exists$, $\lesseqgtr nR$). One of the main features of OWL (and, originally of Description Logics) is the possibility to express restriction on relationships. Mainly, relationships are represented in OWL by means of properties (which are binary relations among individuals) and, three kinds of restrictions are supported: $\exists R.C$ (called some values from), $\forall R.C$ (called all values from) and restrictions on cardinality $\lesseqgtr nR$. While properties are naturally "imported" in OntoDLP by exploiting relations, the restrictions on properties are simulated by exploiting logic rules.

We start considering $\exists R.C$, and for example, we define the class *Parent* as follows: $Parent \supseteq \exists hasChild.Person$, which means that parent contains the class of all individuals which are child of some instance of person. Importing this fragment of OWL in OntoDLP we obtain:

collection class *Parent* {
X:*Parent()* :– *hasChild(X,Y)*, Y:*Person()*. }

The rule allows one to infer all individuals having at least one child.

Also for the $\forall R.C$ property restriction we use a simple example, in which we define the concept *HappyFather* as follows:

$HappyFather \sqsubseteq \forall hasChild.RichPerson$

In practice, an individual is an happy father if all its children are rich. The above statement can be imported in OntoDLP in the following way:

collection class *RichPerson* {
Y:*RichPerson()* :– *hasChild(X,Y)*, X:*HappyFather()*. }

Similarly, we import the property restriction $\exists R.\{o\}$. For example we can describe the class of persons which are born in Africa as follows:

$African \equiv \exists\ bornIn.africa$

where *africa* is a specific individual representing the mentioned continent. To import it in OntoDLP, we write:

collection class *African* {
X:*African()* :– *bornIn(X,africa).* }
intensional relation *bornIn (domain:object, range:object).*{
$bornIn(X, africa)$:– X : *African().* }

Note that, in this case the import strategy is more precise than the one used of $\exists R.C$; in fact, we could also "fill" the *bornIn* (intensional) relation with exactly all the individuals belonging to class $African$.

We now consider the cardinality constraints that allow one to specify for a certain property either an exact number of fillers ($= nR.C$), or at least n / at most n different fillers (respectively $\geq nR.C$ and $\leq nR.C$). In order to describe the way how $\leq nR.C$ is imported, we define the class *ShyPerson* as a *Person* having at most five friends:

$ShyPerson \equiv \leq 5 hasFriend$

To import it in OntoDLP we write:

collection class *ShyPerson* {
X:*ShyPerson()* :– *hasFriend(X,_)*,
#`count`{ *Y:hasFriend(X,Y)*}<= 5. }

Note that, the aggregate function #`count`(see [5]) allows one to infer all the individuals having less than (or exactly) five friends.

The remaining cardinality constraints can be imported by only modifying the operator working on the result of the aggregate function (with >= and = for $\geq nR.C$ and $= nR.C$, respectively).

OWL also allows to specify domain and range of a property. As an example, consider the property *hasChild* which has domain *Parent* and range *Person*.

$\top \sqsubseteq \forall hasChild^{-}.Parent \qquad \top \sqsubseteq \forall hasChild.Person$

when we import this in OntoDLP we obtain:

relation *hasChild (domain:Parent, range:Person).*

It is worth noting that consistently with *rdfs:domain* and *rdfs:range* semantic, we can state that an individual that occurs as subject (resp. object) of the relation *hasChild*, also belongs to the *Parent* (resp. *Person*) class. To simulate this behavior, the definition of the collection classes *Parent* and *Person* is modified by introducing the following rules (the first for Parent, the second for Person):

X:Parent() :– *hasChild (X,_).*
Y:Person() :– *hasChild (_,Y).*

Moreover, in OWL properties can be organized in hierarchies, can be defined equivalent (by using the *owl:equivalentProperty* construct), functional, transitive and symmetric. Property inheritance is easily imported by exploiting the corresponding OntoDLP relation inheritance, while the remaining characteristics of a property (like being inverse of another) are expressed in OntoDLP by using *intensional relations* with suitable rules. For example if the relation *hasChild* is declared inverse of *hasParent*, when we import it in OntoDLP we have:

intensional relation *hasChild (domain: Parent, range: Person)* {
$hasChild(X, Y)$:– $hasParent(Y, X)$. }
intensional relation *hasParent (domain: Person, range: Parent)* {
$hasParent(X, Y)$:– $hasChild(Y, X)$. }

Similarly, the transitive property *ancestor*, is imported in OntoDLP as:

intensional relation *ancestor (domain: Person,range: Person)* {
$ancestor(X, Z)$:– $ancestor(X, Y), ancestor(Y, Z)$. }

A classic example of symmetric property is the property *marriedWith*. We can import such a property into OntoDLP as:

intensional relation *marriedWith (domain:Person, range:Person)* {
$marriedWith(X, Y)$:– $marriedWith(Y, X)$. }

Moreover, OWL functional and inverse functional properties are encoded by using suitable OntoDLP axioms. For example, consider the functional property *hasFather* and its inverse functional property *childOf*; they are importd in OntoDLP as:

::– *hasFather*(X,_), #count{ *Y:hasFather(X,Y)*}> 1.
::– *childOf*(_,Y), #count{*X:childOf(X,Y)*}> 1.

Intersection, Union and Complement (⊓, ⊔, ¬). In OWL we can define a class having exactly the instances which are common to two other classes. Consider, for example the class *Woman* which is equivalent to the intersection of the classes *Person* and *Female*; in OWL we write:

$Woman \equiv Person \sqcap Female$

This expression is imported in OntoDLP as:

collection class *Woman* ***isa*** { *Person, Female*}*()* {
X:*Woman()* :– X:*Female()*, X:*Person()*. }

Note that we use inheritance in OntoDLP in order to state that each instance of class *Woman* is both instance of *Person* and *Female*; and, conversely, the logic rule allows one to assert that each individual that is common to *Person* and *Female* is an instance of class *Woman*.

In a similar way we deal with the class union construct. For instance, if we want to model the *Parent* class as the union of *Mother* and *Father*, then in OWL we write:

$Parent \equiv Mother \sqcup Father$

and the following is the result of the import of this axiom in OntoDLP:

collection class *Parent* {
X:*Parent()* :– X:*Mother()*.
X:*Parent()* :– X:*Father()*. }

Another interesting construct of OWL is called complement-of, and is analogous to logical negation. An example is the class *InedibleFood* defined as complement of the class *EdibleFood*, as follows:

$InedibleFood \equiv \neg EdibleFood$

and, we import it in OntoDLP by using *negation as failure* as follows:

collection class *InedibleFood* {
X:*InedibleFood()* :– X:*individual()*, `not` X:*EdibleFood()*. }

Individuals and datatypes. The import of the ABox of a OWL ontology is straightforward; and actually, the A-Box assertions are directly imported in OntoDLP facts. For example, consider the following

$Person(mike)$ $hasFather(mark, mike)$

which are, thus imported in OntoDLP as:

mike:Person(). *hasFather(mark,mike).*

OWL makes use of the RDF(S) datatypes which exploit the XMLSchema data-type specifications[8].

OntoDLP does not natively support datatypes other than integer and string. To import the others OWL datatypes we encode each datatype property filler in an OntoDLP string that univocally represents its value.

3.2 Exporting OntoDLP in OWL

In this section, we informally describe how an OntoDLP ontology is exported in OWL by using some example.

Classes. Exporting (base) classes (with no attribute), and inheritance it is quite easy since they can be directly encoded in OWL . For instance:

class *Student* ***isa*** *Person.* becomes simply: $Student \sqsubseteq Person$

However, OntoDLP class attributes do not have a direct counterpart in OWL, and we represent them introducing suitable properties and restrictions. Suppose that the class *Student* has an attribute *advisor* of type *Professor*. To export it in OWL, we first create a the functional property *advisor*, with *Student* as domain and *Professor* as range; and, then we export the class Student as $Student \sqsubseteq \forall advisor.Professor$.[5]

[5] If a class C has more than one attribute, we create a suitable property restrictions for each attribute of C and we impose that C is the the intersection of all the defined property restrictions.

Relations. We can easily export binary (base) relations and inheritance hierarchies in OWL, since the destination language natively supports them. In particular, *isa* statements are translated in inclusion axioms, and domain and range description allowed us to simulate the attributes. For relations having arity greater than two, we adopt the accepted techniques described in the W3C Working Group Note on n-ary Relations [9].[6]

Instances. As we have seen for the import phase, also instances exporting is straightforward. For instance, if we have:

john:Person(father : mike). *friends(mark,john).*

then we can export it in OWL as:

$Person(john)$ $person\_father(john, mike)$ $friends(mark, john)$

Note the *person_father* property, created as explained above for class attributes.

Collection classes and intensional relations. These constructs, representing the "intensional" part of the OntoDLP language, do not have corresponding language feature in OWL. Moreover, collection classes and intensional relations are exploited in the import strategy to "simulate" the semantics of several OWL constructs. Since we want to preserve their meaning as much as possible in our translation, we implemented a sort of "rule pattern matching" technique that recognizes wether a set of rules in a collection class or in an intensional relation corresponds to (the "import" of) an OWL construct. For example, when we detect the following rule (within a intensional relation):

$ancestor(X, Z) :- ancestor(X, Y), ancestor(Y, Z).$

we can assert that the relation *ancestor* is a transitive property. This can be done for all the supported OWL feature, because the correspondence induced by the import strategy between OWL constructs and corresponding collection classes is direct and not ambiguous.

In case of rules that do not "correspond" to OWL features, we export them as strings (using an auxiliary property). In this way, we are able to totally rebuild a collection class (intensional relation) when (re)importing a previously exported OWL ontology.

Axioms and Reasoning Modules. OWL does not support rules, thus we decided to export axioms and reasoning modules only for storage and completeness reasons. To this end, we defined two OWL classes, namely: *OntoDLPAxiom* and *OntoDLPReasoningModule*. Then, for each reasoning module (resp. axiom) we create an instance of the *OntoDLPReasoningModule* (resp. *OntoDLPAxiom*) class representing it; and we link the textual encoding of the rules (resp. axioms) to the corresponding instances of the *OntoDLPReasoningModule* (resp. *OntoDLPAxiom*) class.

[6] Basically, to represent an n-ary relation we create a new auxiliary class having n new functional properties.

4 Theoretical Properties

In this section we show some important properties of out import/output strategies. In particular, we single out fragments of OWL DL and OntoDLP where *equivalence* between the input and the output of our interoperability strategies is guaranteed.

Syntactic Equivalence. Let $import(O_{owl})$ and $export(O_{dlp})$ denote, respectively, the result of the application of our import and export strategies to OWL ontology O_{owl} and OntoDLP ontology O_{dlp}.

Theorem 1. *Given a OWL DL ontology O_{owl}, and an OntoDLP ontology O_{dlp} without class attributes and n-ary relations, we have that:*
(i) $export(import(O_{owl})) = O_{owl}$, and
(ii) $import(export(O_{dlp})) = O_{dlp}$.

This means that if we import (resp. export) an ontology, we are able to syntactically reconstruct it by successively applying the export (resp. import) strategy. Intuitively, the property holds because we defined a bidirectional mapping between the primitives of the two languages (actually, there is no ambiguity since we use a syntactically different kind of rule for each construct).

Semantic Equivalence. We now single out a restricted fragment of OWL DL in which the import strategy preserves the semantics of the original ontology (i.e. the two specifications have equivalent semantics).

Theorem 2. *Let Γ, Γ^R and Γ^L be the following sets of class descriptors: $\Gamma = \{A, B \sqcap C, \exists R.o\}$, $\Gamma^R = \Gamma \cup \{\forall R.C\ \}$, $\Gamma^L = \Gamma \cup \{\exists R.C, \sqcup\}$, and, let O_{owl} be an ontology containing only:*

- *class axioms $A \equiv B$ where $A,B \in \Gamma$;*
- *class axioms $C \sqsubseteq D$ where $C \in \Gamma^L$ and $D \in \Gamma^R$;*
- *property axioms: domain, range, inverse-of, symmetry and transitivity;*
- *ABox assertions*

then $import(O_{owl})$ under the OntoDLP semantic entails precisely the same consequences as O_{owl}.

Intuitively, the equivalence property holds because[7] the the First Order Theories equivalent to the admitted fragment of OWL only contains Horn equality-free formulae whose semantics corresponds to the one of the produced logic program.

[7] According to the approach of Borgida [10], also used in [11].

5 Conclusion and Related Work

In this paper, we proposed a *pragmatic* approach to the problem of interoperability between OntoDLP and OWL. In particular, we designed and implemented in the OntoDLV system two transformations which are able to *import* an OWL ontology in OntoDLP, and *export* an OntoDLP specification in an OWL one.

Moreover, we studied the theoretical properties of the above-mentioned transformations and we obtained some interesting results. The first one regards a syntactic property of the two transformations. Basically, it is guaranteed that applying the *import* (resp. *export*) transformation to an OWL (resp. OntoDLP) ontology O, we can rebuild O by applying the "inverse" *export* (resp. *import*) transformation. Furthermore, we identified some fragments of OWL for which the *semantic equivalence* between the original OWL ontology and the obtained OntoDLP ontology is guaranteed.

Our approach is, somehow, connected with the effort of combining OWL with rules for the Semantic Web (see [12] for an excellent survey). Indeed, one might think to import an OWL ontology in OntoDLP in order to exploit the latter to perform reasoning on top of the original specification. This simple idea has the drawback (from the Semantic Web viewpoint) of relaxing important assumptions like both open-world and unique name assumption. Conversely, one of the major problems existing in the interaction of rules and description logics with strict semantic integration is retaining decidability (which is, instead, ensured in our framework) without loosing easy of use and expressivity. For instance, the SWRL[13] approach is undecidable; while, in the so-called DL-safe rules [14] a very strict safety condition is imposed to retain decidability. Notably, this safety condition has been recently weakened in some works [15, 16] thus obtaining a more flexible environment. However, the goal of the above-mentioned approaches is different from the one achieved in this paper; where we introduced some interoperability mechanisms to combine OWL and OntoDLP ontologies. Conversely, the techniques exploited to obtain our import transformation are similar to (even if more pragmatic and less general than) the ones used for reducing description logics to logic programming (see [17, 18, 11, 19, 20]).

References

1. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) http://www.w3.org/TR/owl-guide/.
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language (XML) 1.0. W3C Recommendation (2000) http://www.w3.org/TR/REC-xml/.
3. Brickley, D., Guha, R.V.: Rdf vocabulary description language 1.0: Rdf schema. w3c recommendation (2004) http://www.w3.org/TR/rdf-schema/.
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
5. Ricca, F., Leone, N.: Disjunctive Logic Programming with Types and Objects: The DLV+ System. Journal of Applied Logics **5** (2007)

6. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
7. 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** (2006) 499–562
8. T.Paul, V. Biron, K.P., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition. (2004) http://www.w3.org/TR/xmlschema-2/.
9. Noy, N., Rector, A.: Defining N-ary Relations on the Semantic Web. W3C Working Group Note (2006) http://www.w3.org/TR/swbp-n-aryRelations/.
10. Borgida, A.: On the relative expressiveness of description logics and predicate logics. Artificial Intelligence **82** (1996) 353–367
11. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proc. of the International World Wide Web Conference, WWW2003, Budapest, Hungary. (2003) 48–57
12. Antoniou, G., Damsio, C.V., Grosof, B., Horrocks, I., Kifer, M., Maluszynski, J., Patel-Schneider, P.F.: Combining Rules and Ontologies. A survey. (2005)
13. 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/.
14. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. J. Web Sem. **3** (2005) 41–60
15. Rosati, R.: Integrating ontologies and rules: Semantic and computational issues. In: Reasoning Web. (2006) 128–151
16. Rosati, R.: Dl+log: Tight integration of description logics and disjunctive datalog. In: KR. (2006) 68–78
17. Belleghem, K.V., Denecker, M., Schreye, D.D.: A strong correspondence between description logics and open logic programming. In: ICLP. (1997) 346–360
18. Swift, T.: Deduction in ontologies via asp. In: LPNMR. (2004) 275–288
19. Heymans, S., Vermeir, D.: Integrating semantic web reasoning and answer set programming. In: Answer Set Programming. (2003)
20. Hustadt, U., Motik, B., Sattler, U.: Reducing shiq-description logic to disjunctive datalog programs. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada. (2004) 152–162