OntoDLV: An ASP-based System for Enterprise Ontologies

FRANCESCO RICCA, *Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy.*
E-mail: ricca@mat.unical.it

LORENZO GALLUCCI, ROMAN SCHINDLAUER and TINA DELL'ARMI, *Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy and Exeura S.r.l., c/o University of Calabria, 87036 Rende (CS), Italy.*
E-mail: gallucci@exeura.it, roman@mat.unical.it, dellarmi@exeura.it

GIOVANNI GRASSO and NICOLA LEONE, *Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy.*
E-mail: {grasso,leone}@mat.unical.it

Abstract

Enterprise/Corporate ontologies are widely adopted to conceptualize business enterprise information. In this area, the semantic peculiarities of Answer Set Programming (ASP), like the Closed World Assumption (CWA) and the Unique Name Assumption (UNA), are more appropriate than the Ontology Web Language (OWL) assumptions, also because such ontologies frequently stem from relational databases, where both CWA and UNA are adopted. This article presents OntoDLV, a system based on 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 features 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 from OWL ontologies, and build rule-based reasoning on top of OWL ontologies. The system is already used in a number of real-world applications including agent-based systems, information extraction, and text classification.

Keywords: Logic programming, disjunctive logic programming, answer set programming, ontology languages, enterprise ontologies, OWL.

1 Introduction

In the last few years, the need for knowledge-based technologies has emerged 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) [48] has already provided recommendations and standards related to ontologies, like RDF(S) [49] and the Ontology Web Language (OWL) [43]. OWL was conceived for the Semantic Web, with the goal of enriching Web pages with machine-understandable descriptions of the presented contents. OWL is based on expressive Description Logics (DL) [5]; distinguishing features of its semantics w.r.t. Logic Programming languages are the adoption of the Open World Assumption (OWA) and the non-uniqueness of names (different names can denote the same individual).

While the semantic assumptions of OWL make sense for the Web, they are unsuited for Enterprise ontologies. Enterprise/Corporate ontologies are specifications of terms and definitions relevant to

Vol. 19 No. 4,
Published online 14 August 2008 doi:10.1093/logcom/exn042

TABLE 1. The supplier-branch table

Supplier	Branch city	Branch street
Barilla	Rome	Veneto
Barilla	Naples	Plebiscito
Voiello	Naples	Cavour

business enterprises; they are used to share/manipulate the information already present in a company. Since an enterprise ontology describes the knowledge regarding specific aspects in 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 domain). Moreover, the presence of naming conventions, often adopted in enterprises, can guarantee name uniqueness, making also the Unique Name Assumption (UNA) plausible. Importantly, enterprise ontologies often are the evolution of relational databases, where both CWA and UNA are mandatory. To understand the suitability of 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).

Consider the following query: 'who are the pasta suppliers of the company having a branch *only* in Naples?' The expected answer 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 to 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 (actually, the names Rome and Naples denote the same city). This, indeed, is the only way to satisfy the constraint, even if it is not the intended/expected behaviour in this scenario.

Similar scenarios are frequent when enterprise ontologies are being dealt with. In these cases logic programming languages like ASP, strongly relying on CWA and UNA, are definitely more appropriate than OWL.

Answer Set Programming (ASP) [17], 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) [12]. However, traditional ASP is not well-suited for ontology specifications, since it does not directly support features like classes, taxonomies, individuals, etc. Moreover, ASP systems are a long way from comfortably enabling the development of industry-level applications, mainly because they lack important tools for supporting users and programmers. In particular, convenient user interfaces are missing, and there is a lack of advanced Application Programming Interfaces (API) for implementing applications on top of ASP systems. This article describes OntoDLV, an ASP-based system for enterprise ontologies, 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 from external OWL Ontologies and to exploit this data in OntoDLP ontologies and queries.[1] Moreover, OntoDLV facilitates the development of complex applications in a user-friendly visual environment; it is endowed with a robust persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates the DLV system [32] exploiting the power of a stable and efficient ASP solver.

Using OntoDLV, domain experts can create, modify, store, navigate and query ontologies thanks to a user-friendly visual environment; at the same time, application developers can easily implement knowledge-intensive applications embedding OntoDLP specifications using a complete API. Indeed, OntoDLP is already used for the development of real-world applications including agent-based systems, information extraction and text classification frameworks (Section 7).

REMARK
OntoDLV has its roots in the previous DLP+ system [39]; but, compared to its predecessor, OntoDLV includes many new major features. Among them, Complex types (like Sets and Lists), Object reclassification support (Collection Classes), Intensional Relations, OWL interoperability mechanisms, an API and a more advanced Graphical User Interface (GUI). All these together with many optimization techniques make OntoDLV well-suited for the development of industrial applications (Section 7). Importantly, the OntoDLP language has a direct model-theoretic semantics, while the meaning of DLP+ programs was specified merely by a rewriting technique (See Section 6 for a comparison with DLP+). ■

The remainder of the article is organized as follows: in Section 2, an informal overview of the OntoDLP language by examples is provided; Section 3 gives a formal definition of OntoDLP; Section 4 describes our mechanism for OWL interoperability; Section 5 overviews the architecture and the implementation of the OntoDLV system; related work is discussed in Section 6 and Section 7 concludes the article and points out a couple of relevant applications of our system.

2 OntoDLP by example

This section describes OntoDLP, an ontology representation and reasoning language which provides the most important ontological constructs, namely classes, attributes, relations, inheritance and axioms, and combines them with the reasoning capabilities of ASP. To this end, an example (the *banking ontology*) is esploited which will be specified throughout the whole section, thus illustrating the features of the language. For a better understanding, each construct will be described in a separate paragraph.

Hereafter, the reader is assumed to be familiar with ASP syntax and semantics, for further details refer to [17, 32].

Classes. A (base) *class*[2] can be thought of as a collection of individuals who belong together because they share some properties. Classes can be defined in OntoDLP by using the keyword **class** followed by its name. 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 the aim is to model the domain of a *banking* enterprise, and some classes of individuals have been identified, namely: *banks*, *branches*, *accounts*, *persons*, *enterprises* and *places*.

[1] It is acknowledged that rule-based inference systems are needed by OWL applications [19, 27]. OntoDLP can also be exploited for rule-based reasoning on top of OWL ontologies.

[2] For simplicity, we often 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.

Suppose also that a number of relevant properties (or attributes) shared by all the individuals belonging to these classes are recognized. For instance, it is known that: banks have a name and own an asset; the branches of a given bank are located into a given place and also have an asset; accounts have a balance; enterprises have a name and a country (which is a place); persons have name, age, residence (which is also a place), father and mother (which are other persons); and finally, each place has a name.

The above-listed classes and (related) attributes can be represented in OntoDLP as follows:

class *bank*(*name* : *string*, *asset* : *integer*).
class *account*(*balance* : *integer*).
class *branch*(*bank* : *bank*, *location* : *place*, *asset* : *integer*).
class *place*(*name* : *string*).
class *enterprise*(*name* : *string*, *country* : *place*).
class *person*(*name* : *string*, *age* : *integer*,
father : *person*, *mother* : *person*, *residence* : *place*).

Note that OntoDLP permits user-defined classes as attribute types, thus allowing for objects made of other objects (complex objects). Class attributes in OntoDLP model the properties that *must* be present in all class instances; *optional* properties should be modelled by using other constructs such as lists, sets and relations, that will be described later in this section. Moreover, class definitions can be recursive (e.g. in class person both father and mother are of type *person*), and attribute types can exploit the built-in classes *string* and *integer* (respectively, representing the class of all alphanumeric strings and the class of non-negative integers).

Objects. Domains contain individuals which are called *objects* or *instances*. Each individual in OntoDLP belongs to a class and is uniquely identified by 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 facts

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

it is declared that *rome* and *john* are instances of the class *place* and *person*, respectively. Note that, when an instance is declared, an oid is immediately given 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 residence is filled with the oid *rome* modelling the fact that *john* lives in Rome; in the same way, *jack* and *ann* are suitable oids, respectively, filling the attributes *father*, *mother* (both of type person).

Referential integrity (Section 3) is guaranteed by the language (and our implementation), thus *jack*, *ann* and *rome* have to exist in order to declare *john*. Moreover, in OntoDLP oids are proper of a given base class, i.e. base classes cannot share individuals. However, an individual may belong to different classes when other two modelling tools are employed (that will be described later), namely: inheritance and collection classes.

Sets and lists. A feature of OntoDLP is the possibility of exploiting two types of *container classes* (i.e. classes whose instances are groups of objects): sets and lists.

A *set* is a collection of instances (the order of which is not significant), while a *list* is an ordered collection of instances that accepts *multiple copies* of the same instance. Given a class C, one can define the class 'set of C' (resp. 'list of C'), denoted by $\{C\}$ (resp. $[C]$), having as instances all sets (resp. lists) of individuals belonging to class C.

For instance, the class $\{string\}$ (resp. $[string]$) represents the class having as instances all sets (resp. lists) of strings. Analogously, {"*This*", "*That*"} is the set containing two strings, namely "*This*" and "*That*"; while ["*This*", "*That*"] is the list containing the string "*This*" followed by "*That*".

Container classes are very useful for representing multi-valued attributes. Suppose that in our *banking* domain an account has an associated set of services that can be bought from the bank, e.g. an internet-banking access, an overdraft protection, a payroll payment, etc. Suppose also that account services are represented as follows:

class *accountService*(*cost* : *integer*).
internet : *accountService*(*cost* : 10).
payroll : *accountService*(*cost* : 1). …

The definition of class *account* is upgraded by adding a new attribute *services* of type 'set of *accountService*':

class *account*(*services* : {*accountService*}, *balance* : *integer*).
a0001 : *account*(*services* : {*internet*, *payroll*}, *balance* : 2000).

where the second one is an instance of *account* having associated the two services internet-banking and payroll payment.

Clearly, it is not relevant for accounts to consider an ordering between account services, but there are scenarios where a list is more appropriate than a set. Suppose that, of interest for each payment is its amount and date of execution. Hence, the class account is modified by adding two new attributes, namely *withdrawals* and *deposit*, both of which are of type 'list of payments':

class *payment*(*amount* : *integer*, *date* : *date*).
class *account*(*services* : {*accountService*}, *balance* : *integer*,
withdrawals : [*payment*], *deposits* : [*payment*]).
a0001 : *account*(*services* : {*internet*, *payroll*}, *balance* : 2000,
withdrawals : [ ], *deposits* : [*dp1*]).

This means that, money has never been withdrawn from account *a0001* ([ ] denotes the empty list), whereas only once some money has been deposited in it.

Taxonomies. Concepts in an ontology are usually organized in taxonomies by using the *specialization/generalization* mechanism (which is called *inheritance* in object-oriented languages). This is done when a subset of individuals has some attributes that are not shared by all other individuals in the same class. For instance, employees are a special category of persons having extra attributes, like *salary* and *company*. OntoDLV supports inheritance by means of the special binary relation *isa*. In particular, the above-mentioned employee class can be declared as follows:

class *employee* ***isa*** {*person*}(*salary* : *integer*, *company* : *enterprise*).

In this case, *person* is a more generic concept or *superclass* and *employee* is a specialization (or *subclass*) of *person*. Moreover, an instance of *employee* will have the local attributes *salary* and *company* and *name*, *age*, *father*, *mother* and *residence*, which are defined in *person*. We say that the latter are 'inherited' from the superclass *person*. Hence, each proper instance of *employee* will also be automatically considered an instance of *person* (the opposite does not hold!). For example, the instance:

bob : *employee*(*name* : "*Robert*", *age* : 25, *father* : *jack*, *mother* : *betty*,
residence : *rome*, *salary* : 2000, *company* : *microsoft*).

is automatically considered an instance of person as follows:

bob : *person*(*name* : "*Robert*", *age* : 25, *father* : *jack*, *mother* : *betty*, *residence* : *rome*).

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

Inheritance can be further applied to refine the design of our *banking* ontology. For instance, banks usually offer two different kinds of accounts, *checking* and *savings*. Moreover, inheritance can be applied repeatedly, without limitation on the number of superclasses (i.e. multiple inheritance

is allowed). For instance, a bank may offer two special types of checking account: *gold account* having a fixed minimum balance; and *young account*, which is reserved to customers aged up to 21 years, and is, at the same time, both a saving account and a checking account. This may be specified in OntoDLP as follows:

class *checkingAccount* ***isa*** {*account*}(*overdraftAmount* : *integer*).
class *savingsAccount* ***isa*** {*account*}(*interestRate* : *integer*).
class *goldAccount* ***isa*** {*checkingAccount*}(*minimumBalance* : *integer*).
class *youngAccount* ***isa*** {*savingsAccount*, *checkingAccount*}().

Note that all 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 *A* is a subclass of *B*. For example, the class [*savingsAccount*] (resp. {*savingsAccount*}) is subclass of [*account*] (resp. {*account*}) since *savingsAccount* is subclass of *account*. Finally, OntoDLP has a common built-in superclass called *object*, which, apart from *integer* and *string* also includes *individual*, the superclass of all the user-defined classes.

Relations. Another important feature of an ontology language is the ability to model relationships among individuals. Relations are declared like classes: the keyword ***relation*** (instead of ***class***) precedes a list of attributes. The set of attributes of a relation is called *schema* as for classes, and the cardinality of the schema is called *arity*. As an example, we model a relationship between persons and their bank account as follows:

relation *customerHoldsAccount*(*customer* : *person*, *account* : *account*).

The instances of a relation are called *tuples*, and they can be declared by using logic facts. For instance, it can be asserted that *john* holds account *acc001* by writing a logic fact. Moreover, (since an account may be held by one more customer) two facts can be specified that assign the ownership of account *acc012* to both *ann* and *john*:

customerHoldsAccount(*customer* : *john*, *account* : *acc001*).
customerHoldsAccount(*customer* : *ann*, *account* : *acc012*).
customerHoldsAccount(*customer* : *john*, *account* : *acc012*).

Contrary to class instances, tuples are not equipped with an oid.

The description of relations is completed observing that OntoDLP allows one also to organize them in taxonomies. Basically, attributes and tuples are inherited by following the same criteria defined above for classes. For instance, the relation *holdsSavingsAccount* may be exploited to represent the relationship among persons and their saving account as follows:

relation *holdsSavingsAccount* ***isa*** {*customerHoldsAccount*}(*account* : *savingsAccount*).

Collection classes and intensional relations. The notions of base class and base relation introduced above correspond, from a database point of view, to 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, *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*.

Collection classes are mainly intended for object reclassification (i.e. for classifying individuals of an ontology repeatedly). Suppose the class *richPerson* should be modelled; a person is rich if he owns more than one million in a savings account. The 'intensional' class *rich-person* can be defined as follows:

collection class *richPerson*(*name* : *string*).
$P : richPerson(name : N) :- P : person(name : N), A : account(balance : B),$
$holdsSavingsAccount(customer : P, account : A), B > 1000000.$

Note that in this case the instances of the class *richPerson* are 'borrowed' from the (base) class *person*, and are inferred by using a logic rule. Basically, this class *collects* instances defined by another class (i.e. *person*) and performs a re-classification based on some information which is already present in the ontology. Importantly, the programs (set of rules) defining collection classes must be normal and stratified (see e.g. [4, 38]). Like base classes, collection classes also support inheritance.

Intensional relations allow the definition of 'derived' relations via rules. Instances are declared by a set of logic rules. For example, the binary relation *relative* (modelling the common ancestry among persons) can be easily derived from the information already present in the class *person* as follows:

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

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); the third rule infers all the other connections (X is a relative of Y if exists a third relative Z of X and Y). Note that this definition is recursive, implying that *intensional relations* (as well as *collection classes*) are strictly more powerful than relational database views.

Intensional relations and collection classes can be organized in taxonomies by using the ***isa*** relation. The inheritance hierarchy of intensional relations and collection classes are distinct from the inheritance of base relations and base classes.

Axioms and consistency. *Axioms* are a consistency-control construct modelling sentences that are always true. They can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness. In our example ontology, we may enforce that the father cannot be younger than his son as follows:

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

If an axiom is violated, it can be said that the ontology is inconsistent (i.e. it contains information which is contradictory or not compliant with the domain's intended perception).[3]

Reasoning modules. In addition to the ontology specification, *reasoning modules* are the language components within OntoDLP that allow for reasoning about the ontological data by means of a disjunctive logic program. Reasoning modules are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints. Syntactically, the keyword *module* precedes the name which is followed by a logic program enclosed in curly brackets. By collecting rules in such a block, OntoDLP features a form of modular programming and allows the organization of logic programs in libraries.

The rules of a module can access the information present in the ontology. The programmer may also introduce a number of *auxiliary* predicates which do not require an explicit schema definition.

As an example consider the following module, which computes the number of payments (withdrawals + deposits) performed on a bank account:

$$\textbf{\textit{module}}\ computePaymentsNumber\ \{$$
$$paymentsNumber(A, PayN) :-$$
$$\quad A:account(withdrawals:Withdrawals, deposits:Deposits),$$
$$\quad \#\mathsf{length}(Withdrawals, Wnum), \#\mathsf{length}(Deposits, Dnum),$$
$$\quad PayN = Wnum + Dnum.\}$$

[3] Note that the notion of axiom in OntoDLP is very different from the one employed in other ontology languages like OWL [43]. In fact, an axiom in OntoDLP is a consistency control construct and cannot be used to specify or infer knowledge.

The auxiliary predicate *paymentsNumber* associates an account *A* with the overall number of payments *PayN* performed on it. *PayN* is computed by summing the lengths of both the list of deposits and withdrawals. Note that the length of a list is obtained by exploiting the built-in predicate #length.

Reasoning modules isolate a set of rules and constraints conceptually related and exploit the expressive power of ASP to perform complex reasoning tasks on the information encoded in an ontology.

Querying. An important feature of the language is the possibility of asking queries in order to extract knowledge implicitly contained in the ontology. As in ASP a query can be expressed by a conjunction of atoms, which, in OntoDLP, can also contain complex terms. As an example, the list of persons having a father who lives in Rome is requested as follows:

$X : person(father : person(residence : place(name : \text{“}Rome\text{”})))?$

It is not obligatory to specify all attributes; rather we can indicate only the relevant ones for querying.

3 Formal definition of the OntoDLP language

This section provides a formal definition of OntoDLP. First, it is shown how the structure of the ontology (i.e. the entities schemes and their *isa* relationships) can be defined. Subsequently, it is demonstrated how instances are specified in our logic-based language, formally defining the notion of an OntoDLP program and its answer set semantics.

Hereafter, it is assumed that the reader is familiar with standard ASP [17].

3.1 Schema specification

An *entity-schema specification* is an expression of the form:

Entity-type e ***isa*** $\{e_1,\ldots,e_m\}$ $(n_1 : c_1,\ldots,n_n : c_n)$.

where *Entity-type* is one of {***collection class***,***class***,***relation***,***intensional relation***}; the entity e is called (user-defined) base class, collection class, base relation or intensional relation accordingly. $e_1,\ldots,e_m$ are called *superentities* of e; they are also called *superclasses* or *superrelations* of e, according to whether e is a class or a relation. The pair $\alpha = n_i : c_i$ is called *attribute*; in particular, n_i and c_i are called *name* and *type* of α.[4]

An *ontology-schema specification* Σ is a set of entity-schema specifications. An ontology-schema specification fixes the structure of the instances of each ontology entity, and the way how the entities are *isa*-related. Next the precise meaning of an ontology-schema specification is defined. To this end, we assume that an ontology-schema specification Σ has been fixed.

The set of relation names (specified) in Σ is denoted by R. The set C of all class names of the ontology (specified by Σ) is the union of the following disjoint sets:

- The set of *user-defined class names*.
- The set of *built-in class names object, individual, string, integer*.
- The set of *container-class names* containing both $[c]$ and $\{c\}$, for each built-in or user-defined class name c.

[4] The ***isa*** keyword can be omitted if the set of superentities is empty.

The inheritance hierarchy ***isa*** extends to the entire set of classes and relations in $C \cup R$ as follows:

- e ***isa*** s, for each pair of user-defined entities e and s such that s is a superentity of e.
- c ***isa*** *individual*, for each user-defined base class c.
- c ***isa*** *object*, for each built-in class c in $\{integer, string, individual\}$.
- $[c]$ ***isa*** $[s]$ and $\{c\}$ ***isa*** $\{s\}$, for each pair of user-defined classes c and s so that s is a superentity of c.

The ontology-schema specification provides a minimal set of information on inheritance and schemes, as the user specifies only the 'direct' superentities and indicates only the specific attributes of an entity. In order to completely determine the entity schemes, also the attributes that are implicitly inherited by indirect superentities (e.g. coming from the superentity of a superentity) have to be considered. To this end, the reflexive-transitive closure of ***isa*** is considered, denoted by $\preceq$, and, for each entity $e \in C \cup R$, the schema $\sigma(e)$ of e is defined as follows. If e is either a built-in class or a container class, then $\sigma(e) = \{(self : e)\}$.[5] If e is a relation or a user-defined class with a specification of the form (I), then let $\mathcal{A}$ be union of the set of all attributes appearing in the specification of any entity s such that $e \preceq s$, and $\mathcal{N}$ be the set of the names of the attributes in $\mathcal{A}$.[6] Then, $\sigma(e)$ contains precisely one attribute $n : t$ for each element $n \in \mathcal{N}$; the type t of $n : t$ is the greatest lower bound in C, according with the $\preceq$ relation, of the types of all attributes with name n in $\mathcal{A}$.[7] Moreover, if e is a (user-defined) class, then $\sigma(e)$ additionally contains also $(self : e)$.

Classes always have an attribute named *self* in the schema which is conceived to be filled by the object identifier of class instances. Attributes inheritance may cause some conflicts, when an entity e inherits two attributes $a : t_1$ and $a : t_2$ with the same name but with different types t_1 and t_2. This problem is dealt with by setting in the scheme of e only one attribute with name a, whose type is the 'meet' of the types t_1 and t_2 (formally, this is accomplished by taking the greatest lower bound of t_1 and t_2 in C).

EXAMPLE 1
Consider the following entity-schema specifications:

(1) ***class*** *person*(*name* : *string*, *children* : {*person*}).
(2) ***class*** *employee* ***isa*** {*person*}(*salary* : *integer*).
(3) ***collection class*** *richPerson*(*name* : *string*).

Statement (1) declares (the schema of) a base class named *person* which has two attributes, namely *name* of type *string* and *children* of type {*person*}. Statement (2) declares a subclass *employee* of *person* having attribute *salary* of type *integer*. Statement (3) declares a collection class named *richPerson* with an attribute called *name* of type *string*. The complete $\preceq$ relation is as follows:

$$employee \preceq person \preceq individual \preceq object; \quad [employee] \preceq [person] \preceq [individual] \preceq [object];$$
$$string \preceq object; \quad integer \preceq object; \quad \{employee\} \preceq \{person\} \preceq \{individual\} \preceq \{object\};$$

and the resulting entity schemas are:

$$\sigma(person) = \{self : person, name : string, children : \{person\}\}$$
$$\sigma(employee) = \{self : employee, name : string, salary : integer, children : \{person\}\};$$
$$\sigma(richPerson) = \{self : person, name : string\}$$

■

[5] *self* is a special attribute name, denoting the object-identifier of a class.
[6] Note that $\mathcal{N}$ might have a smaller cardinality than $\mathcal{A}$, as the same name could have multiple occurrences in $\mathcal{A}$.
[7] If such a greatest lower bound does not exist, then the ontology-schema is inadmissible, and our system issues an error.

An ontology-schema specification Σ is required to satisfy three admissibility conditions:

- (s_1) for each entity e in $C \cup R$, there is only one schema specification for e in Σ;
- (s_2) if e' and e'' are two entities defined in Σ such that $e' \preceq e''$, then e' and e'' are of the same type (i.e. they are both base classes, or both collection classes, or both base relations, or both intensional relations);
- (s_3) $\preceq$ is a partial order.[8]

Basically, condition (s_1) ensures that there is only one declaration for each entity; condition (s_2) imposes inheritance hierarchies of base classes, base relations, collection classes and intensional relations to be distinct (e.g. only a base class can be a superclass of a base class, and so on).

In the following, it is assumed that the above-mentioned conditions are satisfied by Σ.

3.2 Instances specification: OntoDLV programs

Once the schema Σ of the ontology has been given, the instances are specified in OntoDLP by means of a logic program on Σ. A *term* is either a variable or constant; constants are of five kinds: (positive) *integers* —sequences of digits, *strings*— sequences of characters enclosed in double quotes, *symbolic constants* —unquoted strings starting with a lower case letter, *set constants*— sets of either *symbolic constants, integers* or *strings* surrounded by curly brackets, *list constants* —sequences of either *symbolic constants, integers* or *strings* surrounded by brackets.[9] A *class atom* is $oid : c(a_1 : t_1, \ldots, a_n : t_n)$, where $oid, t_1, \ldots, t_n$ are terms, $c \in C$ is a class name whose schema is $\sigma(c) = \{self : c, a_1 : k_1, \ldots, a_n : k_n\}$. A *relation atom* is $r(a_1 : t_1, \ldots, a_n : t_n)$, where $t_1, \ldots, t_n$ are terms, $r \in R$ is a relation name whose schema is $\sigma(r) = \{a_1 : k_1, \ldots, a_n : k_n\}$. An *atom* is either a class atom or a relation atom.[10] An atom is called *base class, base relation, collection class* or *intensional relation atom* according with the type of the corresponding entity. As a syntactic sugar, we allow to omit some attributes in atoms; the missing attributes are implicitly filled with fresh variables. That is, if $\sigma(e) = \{a_1 : k_1, \ldots, a_n : k_n\}$, then an atom $e(a_1 : t_1, \ldots, a_m : t_m)$ with $m < n$ is a shorthand for $e(a_1 : t_1, \ldots, a_m : t_m, a_{m+1} : Z_{m+1}, \ldots, a_n : Z_n)$, where $Z_{m+1}, \ldots, Z_n$ are fresh variables (not occurring elsewhere).

EXAMPLE 2
Consider the schema of Example 1. Then, $john : person(name : \text{“}john\text{”}, children : C)$ is a base class atom; $X : richPerson(name : X)$ is a collection class atom. ■

A *literal* is either an atom A (*positive literal*) or its negation *not* A (*negative literal*), where *not* is the default negation symbol. A *rule* r is a formula:

$$a :- b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m.$$

where a is either a class atom or a relation atom, $b_1, \ldots, b_m$ are atoms, and $m \geq k \geq 0$. The consequent a of r is called the *head* of r; the antecedent $b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m$ of r is called the *body* of r. If the body is empty, r is called *fact*, and the $:-$ symbol is omitted. If the head is missing, then r

[8] That is, $\preceq$ is *reflexive, antisymmetric and transitive.*

[9] Symbolic constants denote the oids of user-defined classes, integers and strings denote the oids of the respective built-in classes, and sets and list constants denote the respective instances (sets and lists).

[10] Actually, OntoDLV offers built-in support also for the standard comparison predicates $=, <, \leq, >, \geq, \neq\}$ and for aggregates as in DLV [11, 32]. Since their treatment is precisely the same as in standard ASP, we omit their formal specification here, for simplicity.

is called *integrity constraint* or *axiom*. A rule r is *safe* if each variable of r appears in at least one positive body literal of r.[11] Terms, atoms, literals and rules are *ground* if no variables appear in them.

EXAMPLE 3
Consider the following rule r:

$P : richPerson(name : N) :- P : employee(name : N,\ salary : 100000).$

The head of r is $P : richPerson(name : N)$, while the body of r contains the base class atom $P : employee(name : N, salary : 100000)$. r is not ground but it is safe. ■

As a syntactic sugar, nesting class atoms is also allowed for 'navigating' through objects, by using the so called *complex terms*. A *complex term* is a class atom without the initial oid-term and colon, which can occur in place of a regular term in the rule body, and represents a syntactic shorthand for a more involved expression. Consider a relation atom containing a complex term $A = e(a : q(a_1 : t_1, \ldots, a_n : t_n))$. Then A simply stands for the conjunction $e(a : K), K : q(a_1 : t_1, \ldots, a_n : t_n)$. This syntactic transformation trivially extends to any atom where complex terms fill more than one attribute.

EXAMPLE 4
Consider the following rules:

(1) $hasFatherJohn(person : P) :- P : person(father : person(name : \text{“}john\text{”}))$.
(2) $hasFatherJohn(person : P) :- P : person(father : F), F : person(name : \text{“}john\text{”})$.

Then rule (1) is a shorthand for rule (2) where the complex term has been eliminated. ■

In order to manipulate the content of list and set terms, a number of special built-in predicates is allowed in rule bodies. Built-in predicates follow the approach described in [7]; a complete listing of these predicates, together with a description of their behaviour and applicability conditions, is available on the web at http://www.mat.unical.it/ricca/downloads/rt-ontodlp.zip.

An *OntoDLP program* $\mathcal{P}$ on Σ is a finite set of rules such that: (p_1) every rule having either a base class atom or a base relation atom in the head has an empty body; (p_2) every rule is *safe*; (p_3) $\mathcal{P}$ is *stratified* w.r.t. negation [4, 38].[12]

Condition (p_1) ensures that base classes and base relations are extensionally defined by using ground facts (i.e. following the well-known database terminology, base class and base relation are extensional constructs); condition (p_2) guarantees the finiteness of the domains, by imposing that variables are range-restricted; condition (p_3) ensures that the OntoDLP program has (only) one answer set and specifies an ontology unambiguously. Thus, base class and base relation instances are extensionally specified by explicitly stating a number of facts. Collection classes and intensional relations have an associated set of rules defining their instances. A program can contain also a number of axioms checking the consistency of a specification (w.r.t. the intended model of the world).

EXAMPLE 5
Given the schema specification of Example 1, the following is an OntoDLP program specifying its instances:

$\mathcal{P}_{ex}$ = {
(1) $john : person(name : \text{“}John\text{”}, children : \{ann\})$.
(2) $ann : employee(name : \text{“}Ann\text{”}, children : \{\}, salary : 5000)$.
(3) $P : richPerson(name : N) :- P : employee(name : N, salary : S), S > 4000.$}

[11] The motivation of safety comes from the field of databases, where it guarantees that queries (programs in our setting) do not depend on the universe considered, see [1] for a detailed discussion.

[12] Stratification trivially extends from ASP to OntoDLP. The basic idea of stratification is that programs which are recursive through negation are never stratified. Stratified programs have only one answer set.

In particular, (1) and (2) are two facts declaring instances of *person* and *employee*, while (3) is a rule defining the instances of *richPerson*. ■

3.3 Program semantics

Given an OntoDLP program $\mathcal{P}$ on a schema Σ, it is shown how $\mathcal{P}$ determines the instances of the classes and the relations in Σ. To this end we first define the notions of *universe*, *base* and *instantiation* of an OntoDLP program; in turn, we then define the notion of *interpretation, model* and *answer set* of an OntoDLP program. The answer set of $\mathcal{P}$ (which is unique for an admissible OntoDLP program) allows one to completely determine the instances populating the ontology classes and relations.

Universe and base. Given an OntoDLP program $\mathcal{P}$ on a schema Σ, let $U_{\mathcal{P}}$– the Universe of $\mathcal{P}$– denote the set of constants (oid's) appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$– the Base of $\mathcal{P}$– be the set of ground (class and relation) atoms constructible with the relation and class names (including built-in classes) in Σ as the predicates, and the constants in $U_{\mathcal{P}}$ as the terms. A class atom of the form $o{:}c(a_1{:}f_1, \cdots, a_n{:}f_n) \in B_{\mathcal{P}}$ is called an *instance for c identified by o*. Analogously, relation atom of the form $r(a_1{:}f_1, \cdots, a_n{:}f_n) \in B_{\mathcal{P}}$ is called an *instance (or tuple)* for *r*.

EXAMPLE 6
Consider the program $\mathcal{P}_{ex}$ of Example 5, the universe $U_{\mathcal{P}_{ex}}$ is the set $U_{\mathcal{P}_{ex}} = \{john, ann, \text{“}John\text{”}, \text{“}Ann\text{”}, \{\}, \{ann\}, 5000, 4000\}$; and the base is the following set:[13]

$$
\begin{aligned}
B_{\mathcal{P}_{ex}} = & \bigcup_{x,y,z \in U_{\mathcal{P}_{ex}}} \{x : person(name : y, children : z)\} \\
& \bigcup_{x,y,z,t \in U_{\mathcal{P}_{ex}}} \{x : employee(name : y, children : z, salary : t)\} \\
& \bigcup_{x,y \in U_{\mathcal{P}_{ex}}} \{x : richPerson(name : y)\} \\
& \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : \{person\}()\} \quad \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : \{employee\}()\} \\
& \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : \{richPerson\}()\} \quad \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : [person]()\}, \\
& \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : [employee]()\} \quad \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : [richPerson]()\} \\
& \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : string()\} \quad \bigcup_{x \in U_{\mathcal{P}_{ex}}} \{x : integer()\}
\end{aligned}
$$

■

Instantiation. Given a rule *r*, $Ground(r)$ denotes the set of ground rules obtained from *r* by applying all possible substitutions from the variables in *r* to elements of $U_{\mathcal{P}}$. The *ground instantiation* $Ground(\mathcal{P})$ of an OntoDLP program $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

EXAMPLE 7
Consider the program $\mathcal{P}_{ex}$ of Example 5, its instantiation $Ground(\mathcal{P}_{ex})$ is:

$john : person(name : \text{“}John\text{”}, children : \{ann\}).$
$ann : employee(name : \text{“}Ann\text{”}, children : \{\}, salary : 5000).$
$john : richPerson(name : \text{“}John\text{”}) :-$
$\qquad john : employee(name : \text{“}John\text{”}, salary : 4000), 4000 > 4000.$
$john : richPerson(name : \text{“}John\text{”}) :-$
$\qquad john : employee(name : \text{“}John\text{”}, salary : 5000), 5000 > 4000.$
…

Note that the first two atoms were already ground, while the rules are obtained by replacing the variables of rule (3) with constants in $U_{\mathcal{P}_{ex}}$. ■

[13] We omit the atoms of the built-in classes for brevity.

Well-typed interpretation. An *interpretation* for an OntoDLP program $\mathcal{P}$ is a set of ground atoms $I \subseteq B_{\mathcal{P}}$. A positive (negative) literal A (*not* A) is true w.r.t. I if $A \in I$ ($A \notin I$); otherwise it is false. A rule $r \in Ground(\mathcal{P})$ is satisfied w.r.t. I, if its head is true w.r.t. I or some body literal of r is false w.r.t. I.

An interpretation I for $\mathcal{P}$ is *well-typed* if it satisfies the following conditions:

- For each atom s *: string()*$\in I$, s is a (quoted) string in $U_{\mathcal{P}}$; *(string)*
- For each atom i *: integer()*$\in I$, i is an integer in $U_{\mathcal{P}}$; *(integer)*
- For each atom oid *:* $c(a_1:f_1,\ldots,a_n:f_n)\in I$, let $\sigma(c)=\{self:c,a_1:k_1,\ldots,a_n:k_n\}$ be the schema of c, then for all $i\in[1,n]$ there exists an instance for k_i identified by f_i which is true w.r.t. I; *(user class)*
- For each atom $r(a_1:f_1,\ldots,a_n:f_n)\in I$, let $\sigma(r)=\{a_1:k_i,\ldots,a_n:k_n\}$ be the schema of r, then for all $i\in[1,n]$ there exists an instance for k_i identified by f_i which is true w.r.t. I; *(relation)*
- For each atom $[L]:[c]()\in I$, L is a (possibly empty) sequence of constants such that for each element t in L there exists an instance for c identified by t which is true with respect to I; *(list)*
- For each atom $\{S\}:\{c\}()\in I$, S is a (possibly empty) set of constants such that for each element t in S there exists an instance for c identified by t which is true with respect to I; *(set)*

EXAMPLE 8

Given the ground program of Example 7, $I=\{ann:employee\ (name:\text{“}Ann\text{”},children:\{\},salary:5000),\{\}:\{person\}(),\{ann\}:\{person\}(),5000:integer(),\quad 4000:integer(),\text{“}Ann\text{”}:string(),\text{“}John\text{”}:string()\ \}$, is an interpretation, and we have that:

- $5000:integer()$, and $4000:integer()$ are true w.r.t. I; (integer)
- “*Ann*”: *string*(), and “*John*”: *string*() are true w.r.t. I; (string)
- $ann:employee(name:\text{“}Ann\text{”},children:\{\},salary:5000)$. is true w.r.t. I; (user class)
- $john:person(name:\text{“}John\text{”},children:\{ann\})$ (which is not in I) is false w.r.t. I (user class);
- $\{\}:\{person\}()$, and $\{ann\}:\{person\}()$ are true w.r.t. I; (set).
- $[ann]:[person]()$ (which is not in I) is false w.r.t. I (list);
- $ann:employee(name:\text{“}Ann\text{”},children:\{\},salary:ann)$. is false w.r.t. I, since it is not well-typed (violates correct typing on attribute salary), indeed *ann* is not an instance of *integer*.
- I is well-typed.

Basically, I gives a truth value for atoms (user-defined, as well as built-in, list and set atoms); moreover, it guarantees correct typing. ■

Model. A well-typed interpretation I for $\mathcal{P}$ is a model, if I satisfies every rule $Ground(\mathcal{P})$ and verifies the following conditions:

(1) If $s \preceq c$, c is a class with schema $\sigma(c)=\{self:c,a_1:k_1,\ldots,a_n:k_n\}$, and $oid{:}s(a_1:f_1,\ldots,a_n{:}f_n,\ldots,a_z{:}f_z)\in I$, then oid *:* $c(a_1:f_1,\ldots,a_n:f_n)\in I$. (A class gets also the instances of its subclasses.)

(2) If $s \preceq r$, r is a relation with schema $\sigma(r)=\{a_1:k_1,\ldots,a_n:k_n\}$, and $s(a_1:f_1,\ldots,a_n{:}f_n,\ldots,a_z{:}f_z)\in I$, then $r(a_1:f_1,\ldots,a_n:f_n)\in I$. (A relation gets also the tuples of its subrelations.)

(3) There are not two different class atoms in I $oid:c(A)$ and $oid:c(B)$ (both sharing the same identifier and belonging to the same class), such that $A=B$ (i.e. oid's uniquely identify an instance).

(4) If two different base class atoms in I $oid:c(A)$ and $oid:s(B)$ share the same identifier, then either $c \preceq s$ and $A \supseteq B$, or vice versa.

(5) If a collection class atom $oid : c(A)$ is in I, then oid is the identifier also of a base class atom in I.

A model I for $\mathcal{P}$ is an interpretation that satisfies all rules as in standard ASP, but it is endowed with some additional properties. Intuitively, each instance respects the types specified in the respective entity schema, each entity has also the instances if its subentities, object identifiers univocally determine instances (apart from inheritance), collection classes do not have proper oids as they re-classify existing class instances.

EXAMPLE 9
Given the schema specification of Example 1, consider the program:

$$\begin{aligned}\mathcal{P}^r_{ex} = \{\ & john : person(name : \text{“}John\text{”}, children : \{ann\}).\\ & ann : employee(name : \text{“}Ann\text{”}, children : \{\}, salary : 5000).\\ & ann : richPerson(name : \text{“}Ann\text{”}) :-\\ & \qquad ann : employee(name : \text{“}Ann\text{”}, salary : 5000), 5000 > 4000.\}\end{aligned}$$

and the following well-typed interpretations:

$$\begin{aligned}I_0 = \{\ & 5000 : integer(), 4000 : integer(), \text{“}Ann\text{”} : string(), \text{“}John\text{”} : string(),\\ & \{\} : \{person\}(), \{ann\} : \{person\}()\ \}\\ I_1 = I_0 \cup \{\ & john : person(name : \text{“}John\text{”}, children : \{ann\})\ \}\\ I_2 = I_1 \cup \{\ & ann : employee(name : \text{“}Ann\text{”}, children : \{\}, salary : 5000)\ \}\\ I_3 = I_2 \cup \{\ & ann : person(name : \text{“}Ann\text{”}, children : \{john\}),\ \{john\} : \{person\}()\ \}\\ I_4 = I_2 \cup \{\ & ann : person(name : \text{“}Ann\text{”}, children : \{\}),\ ann : richPerson(name : \text{“}Ann\text{”})\ \}\\ I_5 = I_4 \cup \{\ & john : employee(name : \text{“}John\text{”}, children : \{ann\}, salary : 4000)\ \}\end{aligned}$$

Then, only the last two are models for $\mathcal{P}^r_{ex}$, indeed: I_1 does not contain an instance for *ann* (thus, the second fact is not satisfied); I_2 violates condition (1), since $employee \preceq person$ holds, *ann* must be also instance of *person*; I_3 violates condition (4), since attribute children is not properly 'inherited' and, in addition, it does not satisfy the last rule (having true body and false head w.r.t. I_3); while, both I_4 and I_5 satisfy all the above-listed conditions. It is worth noting that, mistyped programs, where either object identity [i.e. conditions (3), (4)] or object origin [i.e. condition (5)] are violated, have no model. As an example, consider, the program $\mathcal{P}'_{ex} = \mathcal{P}_{ex} \cup \{john : person(name : \text{“}Jonny\text{”}, children : \{annette\})\}$, where *john* identifies two different instances. $\mathcal{P}'_{ex}$ has no model, indeed there exist no interpretation satisfying condition (3). ■

Answer set. A model M for $\mathcal{P}$ is minimal if no model N for $\mathcal{P}$ exists such that $N \subset M$.

EXAMPLE 10
Consider program $\mathcal{P}^r_{ex}$ and the two models I_4 and I_5 of $\mathcal{P}^r_{ex}$ reported in Example 9. Then, I_5 is not minimal ($I_4 \subset I_5$); while, I_4 is minimal; indeed there is no smaller subset of $B_{\mathcal{P}}$ that is a model for $\mathcal{P}^r_{ex}$.

It is worth noting that, minimality ensures the correctness of object inheritance. Indeed, an additional fact claiming that *john* is also an *employee* is true w.r.t. I_5; while, there is only one fact stating that *john* is an instance of *person* in $\mathcal{P}_{ex}$. ■

Given a ground OntoDLP program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I, $\mathcal{P}^I$ is called *reduct* of $\mathcal{P}$. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

An OntoDLP program $\mathcal{P}$ admits at most one answer set, because it contains only stratified rules [4, 32, 38] and axioms. If there is no answer set it can be said that $\mathcal{P}$ (and, thus the entire ontology specification) is *inconsistent*. This is the case when either the program is mistyped and contains some

statement that violates typing or object identity (Example 9), or some axiom cannot be satisfied (i.e. some axiom is violated).

EXAMPLE 11
Consider the program $\mathcal{P}_{ex}$ of Example 5, then it can be verified that interpretation I_4 of Example 10 is the answer set for $\mathcal{P}_{ex}$. Indeed, the reduct $Ground(\mathcal{P}_{ex}) = \mathcal{P}^r_{ex}$, and I_4 is a minimal model for it (see Example 10). Note also that, if the axiom $:- X : employee(salary : S)$ is added to $\mathcal{P}$, then $\mathcal{P}$ becomes inconsistent (since I_4 violates it). ■

Given a schema specification Σ and a consistent OntoDLP program on Σ, the instances specified by $\mathcal{P}$ are exactly those contained in the answer set of $\mathcal{P}$.

Querying. Given an OntoDLP program $\mathcal{P}$, a query is a tuple $\mathcal{Q} = \langle T, Conj \rangle$ where *Conj* is a conjunction of atoms and $T = \{X_1, \ldots, X_n\}$ $(n \geq 0)$ is the set of distinct variables occurring in *Conj*. We say that $\mathcal{Q}$ is ground when *Conj* is ground (and, hence $T = \emptyset$). Syntactically, a query is specified by writing *Conj*? (conjunction followed by question mark).

Let A be the answer set of $\mathcal{P}$, a ground query $\mathcal{Q}$ is true when *Conj* is true w.r.t. A, i.e. when all atoms in *Conj* are true w.r.t. I. If a query $\mathcal{Q}$ is non-ground, the answer to it is the set of all tuples $\langle t_1, \ldots, t_n \rangle$ so that when X_i by t_i is substituted in *Conj* the obtained ground conjunction is true w.r.t. A.

EXAMPLE 12
Consider the program $\mathcal{P}_{ex}$ of Example 11, the query $P : person(name : N, children : C)?$ has answer $\{\langle john, \text{“}John\text{”}, \{ann\}\rangle, \langle ann, \text{“}Ann\text{”}, \{\}\rangle\}$, indeed both $john : person(name : \text{“}John\text{”}, children : \{ann\})$ and $ann : person(name : \text{“}Ann\text{”}, children : \{\})$ are true w.r.t. the answer set of $\mathcal{P}_{ex}$. ■

3.4 Reasoning on ontologies

An important construct of the language are *reasoning modules*, which allows one to reason on top of OntoDLP ontologies. Basically, reasoning modules are *disjunctive* programs. We now define auxiliary atoms and disjunctive rules by extending the notion of atom and rule given in Section 3.2.

Let P be a set of *auxiliary predicate* symbols such that $P \cap E = \emptyset$. An *auxiliary atom* is an expression $p(t_1, \ldots, t_n)$, where $p \in P$ and $t_1, \ldots, t_n$ are terms. Auxiliary atoms are standard atoms that do not have any typing constraint on attributes. Moreover, since attributes does not have names the order of terms is meaningful, like in standard ASP. A *disjunctive rule* is an expression of the form:

$$a_1 \vee \ldots \vee a_n :- b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m. \quad n, m \geq 0$$

where, a set of *auxiliary* atoms $a_1, \ldots, a_n$ is allowed in the head, and $b_1, \ldots, b_m$ are atoms, and $m \geq k \geq 0$. Let Σ be a schema specification and let $\mathcal{P}$ be an OntoDLP program on Σ, a *reasoning module* Π_m is a pair $\Pi_m = \langle m, \mathcal{P}_m \rangle$, where m is a reasoning module name, and $\mathcal{P}_m$ is a set of safe disjunctive rules called the definition of Π_m. Syntactically, a reasoning module Π_m is specified as ***module*** m $\{\mathcal{P}_m\}$, where the keyword ***module*** is followed by the name of the module, while the definition of Π_m is enclosed in curly brackets. The semantics of a reasoning module $\Pi_m = \langle m, \mathcal{P}_m \rangle$ is given by the answer sets of the program $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}_m$ obtained by the union of $\mathcal{P}$ with the definition Π_m. The notion of answer set for $\mathcal{P}'$ is obtained by adopting the definitions of universe, base, instantiation, interpretation, minimal model and answer set of Section 3.2, provided that, given an interpretation I, a disjunctive rule r is satisfied if *some* head atom is true w.r.t. I whenever all body literals are true

w.r.t. I. It is worth noting that, in this case, like in standard ASP, $\mathcal{P}'$ may admit several answer sets, each of which represents a possible solution for the problem solved by Π_m.

EXAMPLE 13
Consider the OntoDLP program $\mathcal{P}_{ex}$ of Example 5, and this module:

module *friend* $\{friend(X) \vee notFriend(X) :- X : employee().\}$.

Then, $\mathcal{P}'_{ex} = \mathcal{P}_{ex} \cup \mathcal{P}_{friend}$ has two answer sets, namely: $A \cup \{friend(\text{“}Ann\text{”})\}$ and $A \cup \{notFriend(\text{“}Ann\text{”})\}$, where A is the answer set of $\mathcal{P}_{ex}$. ■

4 OWL interoperability

As previously pointed out, 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 [3]. Based on these observations, our system supports a mechanisms for OWL interoperability. Actually, the approach of [14] was lifted to the OntoDLP framework, restricting the semantics and reducing the computational effort.

To enable the interfacing and import of existing OWL ontologies into OntoDLP, the so-called *OWL atoms* were introduced. OWL atoms can be used in rule bodies of OntoDLP's reasoning components and facilitate the evaluation of specific queries to an external OWL knowledge base. This allows one 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 for each OWL atom.

OWL atoms can be used in OntoDLP constructs wherever ordinary atoms are allowed. They can contain variables and are 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 input parameters of OWL atoms are known at runtime, such that they can be fully evaluated prior to any model computation procedure.

It is worth noting that, this choice (i.e. uni-directional flow of information) simplifies our approach compared to related frameworks, such as dl-programs or HEX-programs (Section 6). Indeed, there the original semantics of the formalisms has been modified to take the bidirectionality of the interface into account, resulting in more involved definition/implementation and more computationally expensive systems (again, see Section 6). In our formalism, all OWL atoms can be fully replaced first by ordinary OntoDLP atoms and then the semantics of OntoDLP can be applied as defined in Section 3.

OWL atoms. The types of queries that can be stated by an OWL atom is specified by the DL Implementation Group (DIG) DL Interface [6]. The DIG is a self-selecting assembly of researchers and developers associated with implementations of DL systems. The DIG Interface provides a specification for accessing DL reasoner functionality. It allows for submitting DL axioms (via the so-called TELL statement) as well as TBox and ABox queries (via the ASK statement), 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:

$$\#\texttt{OWL}[querytype, query, source]$$

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*, "*Truck SUV*", *"/data/loans.owl"*] is a purely boolean query, evaluating to true if the concepts *Truck* and *SUV* are disjoint in the specified OWL KB about loan data.

Moreover, the following rule imports all children classes of the concept *car_loans* of the specified OWL-KB:

$$typesOfCarLoans(X) :- \#\texttt{OWL}[children, \text{"?}X\ car\_loans\text{"}, \textit{"http://ex.org/loans.owl"}]$$

These children concept names instantiate the variable X in the respective rule. In order to be distinguishable from uppercase 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:

$$trusted(P) :- \#\texttt{OWL}[instances, \text{"}trusted\_employers(?X)\text{"}, \textit{"http://ex.org/loans.owl"}]$$

The difference to the previous example is the type of the return value. In previous example, concept names were imported, being translated to OntoDLP *names*, whereas here we import OWL individuals (members of a specific concept) into OntoDLP *constants*.

The following collection class gathers all trustable clients together with their credit score. Note that, the information about employers and trustable companies stems from OntoDLP itself, while the credit score information is derived from an external ontology.

$$\begin{aligned}&\textbf{\textit{collection class}}\ trustableClient(creditScore : integer).\\ &X : trustableClient(creditScore : S) :- X : client(creditScore : S),\\ &\#\texttt{OWL}[relatedIndividuals, \text{"}hasCreditScore(?X, ?S)\text{"}, \textit{"credit.owl"}],\\ &employedBy(X, C), trustableCompany(C).\end{aligned}$$

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.

$$\begin{aligned}&\textbf{\textit{mapping}}\ stocks\{\ \textit{'Amazon'}\quad \textit{'AMZ'}\\ &\qquad\qquad\quad \textit{'HewlettPackard'}\quad \textit{'HPQ'}\ \ldots\end{aligned}$$

If this mapping is specified in a query atom, each occurrence of *AMZ* in the query answer (i.e. data that comes from an OWL ontology) is translated to the name *Amazon* 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

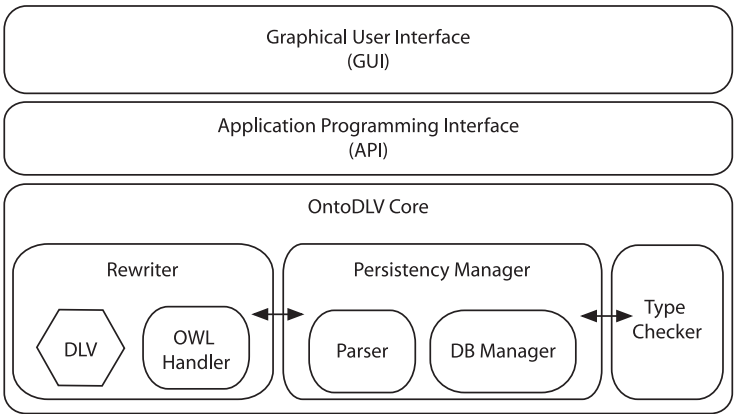

FIGURE 1. The OntoDLV architecture

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:

$$\#\texttt{OWL}[\mathit{relatedIndividuals}, \text{“}\mathit{performance}(?X, ?Y)\text{”}, \mathit{“market.owl”}]$$

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.

Optional mapping arguments can be used to refer to different namespaces in the OntoDLP part and the ontology part. If an individual matches such a namespace, but does not occur in the list of mapping-pairs, just its namespace-portion is replaced. Thus, namespace specifications simply are syntactic sugar, simplifying the replacement of common substrings within names of the OntoDLP universe and/or the ontology universe.

5 The OntoDLV system

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies.

The OntoDLV system is developed in Java by adopting industry-level development standards and ensures the stability and performances required by real-world applications. Here, an in-depth description is not given of all technical details underlying the implementation of OntoDLV, rather the main features of the system are presented. Moreover, at the end of this section we report some preliminary experiments for showing the efficiency of the implementation, and for outlining that OntoDLV scales well even when compared with one of the most used DL-systems.

The system architecture of OntoDLV, depicted in Figure 1, can be divided into three abstraction layers. The lowest layer, named *OntoDLV core* contains the components implementing the main functionalities of the system; above it, the API act as a facade for supporting the development of applications based on the core; while the GUI is the end-user interface of the system.

OntoDLV core. The OntoDLV system was conceived for efficiently handling real-world enterprise ontologies. This kind of ontology usually contains a large number of instances that are distributed

across several machines. To deal with this, the kernel of the system is equipped with a flexible persistency manager that is able to deal with large distributed ontologies. Indeed, ontologies can be stored transparently in a number of text files and/or database management systems. Text files in OntoDLP format are analyzed by the *Parser* module that builds in main memory an image of the ontology components it recognizes; while, the *DB Manager* module is able to manipulate ontology entities that are stored in mass-memory by exploiting relational databases. Transparency is obtained, on the one hand by exploiting a common abstract interface for ontology language components (i.e. classes, instances etc.); and, on the other hand, by exploiting the Hibernate framework [30] for Object/Relational mapping. While main memory storage of ontology components is straightforwardly obtained by implementing a library of concrete Java classes (compliant with the abstract interface), the mass-memory storage has been implemented by mapping each concrete Java class to a corresponding representation on a relational database. In this way, database tables are exploited for storing instances and tuples (which in a real-world ontology represent the largest part of the information) in an efficient way. The persistency manager builds a global view of the distributed ontology which can be then exploited by the other components of the kernel, namely: *Type Checker* and *Rewriter*.

The *Type Checker* module verifies the admissibility (Section 3) of the loaded ontology. This process is implemented efficiently by exploiting a number of indexing caching techniques which are garbage-collector friendly (i.e. caches and indexes are retained in main-memory while the system has sufficient space). It is important to say that, if the loaded ontology contains some admissibility problem (e.g. a class is declared twice) the type checker builds a precise description of the problem. This information can be exploited by external applications, and in particular the user interface of OntoDLV relies on this feature for helping the ontology design during the development process.

Another important component of the OntoDLP core is the *Rewriter* module. The *Rewriter* is responsible for the evaluation of both ontology and reasoning modules, which is carried out by invoking the DLV system [32]. DLV is a state-of-the art ASP system that has been shown to perform efficiently on both hard and 'easy' (having polynomial complexity) problems. Since DLV cannot directly deal with ontologies, they must be rewritten in an 'equivalent' ASP program in order to be evaluated. The rewriting algorithm works by exploiting an enhanced version of the algorithm described in [39], which is able to deal with the broader set of constructs supported by OntoDLP. In particular, in this process each class (resp. relation) of the ontology is transformed in a predicate discarding attribute names, while suitable rules are added for handling instance inheritance, and OntoDLP rules are flattened to remove complex terms. Importantly, *ontologies* are translated into an equivalent (stratified) ASP program that is solved by DLV in polynomial time (under data complexity). The import of OWL data is performed during the rewriting phase by the *OWL Handler* submodule which exploits the DIG interface (a general interface towards DL [5] knowledge bases). We used an existing DIG API [46], which is built on Java XMLBeans, providing a straightforward mapping from Java classes to the native DIG XML format. The queries represented by the OWL import atoms are translated into DIG and submitted to an external OWL reasoner, such as Racer Pro [21], Pellet [37] or FaCT++ [45], all three of which are capable of being accessed via DIG. More specifically, the ontologies specified in the import atoms are first translated into DIG themselves and then loaded into the DL reasoner prior to the actual queries.[14] The mapping of individual names (if applicable) is carried out before the query submission and after retrieval of the reasoner reply. The OWL atoms in the rule bodies are replaced by ordinary OntoDLP atoms, while the query result is translated into

[14]This step is necessary, since the currently available DIG version 1.1 does not support the import of an existing ontology in another format, such as OWL/Resource Description Framework (RDF).

according ground atoms. Hence, after this stage, the OntoDLP program is augmented by the imported OWL data and does not contain OWL import atoms anymore.

The rewriting is crucial for the efficiency of the OntoDLV system, and it has been implemented by exploiting a number of optimization and caching techniques. More in detail, when a reasoning task is requested the rewriter selects the minimum set of constructs to be translated and caches it (to avoid multiple rewritings). For instance, if we query the ontology asking for the number of employees named John the procedure rewrites only the employee class (thus avoiding to pass the whole ontology to the reasoner).

OntoDLV API. In order to enable third parties to develop their own knowledge-based applications on top of OntoDLV, we developed a complete application programming interface named OntoDLV API [16]. 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.

OntoDLV GUI. The end user exploits the system through an easy-to-use and intuitive visual development environment called *GUI*, which is built on top of the *OntoDLV API*. The OntoDLV GUI was designed to be simple for a novice to understand and use, and powerful enough to support experienced users. The *GUI* combines a number of specialized visual tools for authoring, browsing and querying a OntoDLP ontology. Queries can be created by exploiting both a text and a graphical 'QBE-like' interface, and results are presented in an intuitive way.

Experiments. An experimental analysis was carried out in order to measure the performance of OntoDLV, and in particular we assessed the efficiency of the rewriting process.

Two different kinds of ontologies were considered, the first set is made of four automatically generated LUBM ontologies of increasing size (number of instances). Actually, LUBM is a benchmark developed at the Lehigh University for testing performance of ontology management and reasoning systems and has been already exploited for testing DL systems performance [20, 35]. Importantly, those LUBM ontologies could be encoded in OntoDLP by preserving the original semantics by exploiting a tool described in [15].

The second set of instances, called HILEX, contains some real-world OntoDLP ontologies. Those instances were courteously provided by the company Exeura s.r.l. and solve a text classification problem with the Hilex system (Section 7).

The experiments were performed on a Centrino Duo 2 Ghz machine with 3GB of RAM, and 80GB 7200 rpm HD running the JDK 1.6.0_06 by Sun (with heap size up to 512 MB) on Windows XP.

The execution time spent by OntoDLV were measured for solving the ontology materialization task, together with the time spent for rewriting the original ontology in standard ASP. In addition, in order to have an idea about the scalability of OntoDLV when compared with with other ontology systems based on DLs, we have run the LUBM instances on the DL system Pellet [37][15] performing the realization task (which corresponds to ontology materialization in OntoDLV). In order to obtain more reliable results each test was processed seven times and here is reported the average time spent by the systems.

In Figure 2, the data collected during the experimental analysis are reported regarding both LUBM and HILEX ontologies. It can be noticed that the rewriting procedure is very efficient, indeed it always amounts to less than the 6% of the total time spent by OntoDLV [Figure 2(a)]. Indeed, if the most significant instance (LUBM 4) is considered only 8.9 s over 198.2 s have been spent for rewriting the

[15] We could not run Pellet on HILEX instances since they contain logic rules.

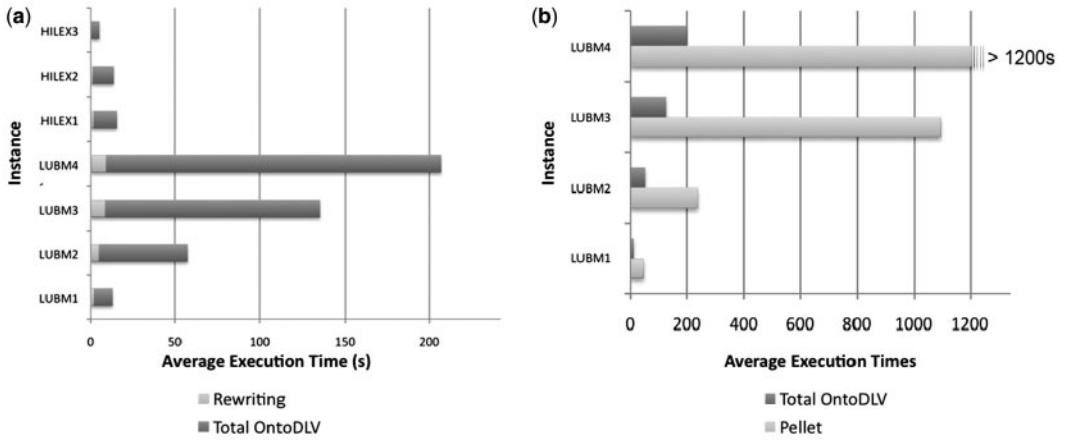

FIGURE 2. Average execution times on both LUBM and HILEX instances

ontology. Concerning the comparison with Pellet [Figure 2(b)], OntoDLV appears to scale well on LUBM instances. Note that, OntoDLV always employed less time than the competitor. In particular, on LUBM 3 OntoDLV took only 126.8 s while Pellet required up to 970.5 s, and the latter could not 'realize' LUBM4 in 1200 s, while our system took only 198.2 s.

It is worth pointing out that these results give us an idea of the performance of OntoDLV when compared with other systems, and have to be considered only indicative; however, a comprehensive comparison is outside of the aim of this article.

6 Related work

There are several languages and systems related to OntoDLP that have been proposed in the literature. Those languages have different characteristics and can be grouped as follows: *Datalog extensions*, *Frame-based languages*, *Semantic Web languages*. OntoDLV is compared with each family of related works in a different paragraph. Finally, OntoDLV is compared with the most closely related system: DLV^+ [39].

Datalog extensions. In the field of databases there are several languages and systems similar to OntoDLP which have been quite successful and positively accepted in the literature (see e.g. [9, 18, 33, 34]), even if they were based on less powerful logic programming languages. Among them, the COMPLEX system [18], which extends Datalog with object-oriented constructs, is the most similar to OntoDLV. COMPLEX and OntoDLP share some similarities in the object-oriented model (like, eg. the attribute inheritance mechanism). However, the latter features a more rich set of modelling constructs, like collection classes, lists and sets. Moreover, COMPLEX supports normal (non-disjunctive) stratified programs, which are a restricted fragment of the logic language of OntoDLP (supporting also disjunction and aggregate functions), thus resulting less expressive than the latter.

Frame-based languages. A popular logic-based language supporting most aspects of object-oriented and frame-based languages is F-Logic [31]. The idea behind F-logic is to exploit a logic programming paradigm for developing intelligent information systems. A main implementation of F-logic is the Flora-2 system [50] which is devoted to Semantic Web reasoning tasks. Flora-2 integrates F-Logic

with other novel formalisms such as HiLog [8] (a logical formalism that provides higher-order and meta-programming features in a computationally tractable first-order setting) and Transaction Logic [2] (that provides a logical foundation for state changes and side effects in a logic programming language). Comparing OntoDLP with F-Logic, we note that the latter has a richer set of object oriented features (e.g. class methods), but it misses some important constructs of OntoDLP like disjunctive rules, which increase the knowledge modelling ability of the language. Concerning system-related aspects, important advantages of OntoDLV (w.r.t. Flora-2) are: the presence of both a tight integration with DL [5] systems and a graphical development environment. The first one simplifies the development of application embedding OWL ontologies while the second one simplifies the interaction with OntoDLV for both the end user and the knowledge engineer.

Semantic web. A number of ontology languages has been proposed in the field of the Semantic web, the most popular of which (RDF [49] and OWL [43]) have been already proposed as a recommendation by the W3C.

RDF is a simple assertional logical language which allows for the specification of binary properties. It has an extended version, called RDFS (RDF Schema), which supports the notions of class and property, and provides mechanisms for specifying domains and ranges of properties, and taxonomies. Compared with OntoDLP, RDF(S) features a richer data-type library, but, it does not provide any way to extract new knowledge from the asserted one (there are no 'inference rules' in RDFS).

OWL is based on RDFS and, in general, allows one to express complex statements about the domain of discourse (OWL is undecidable in general) [43]. The largest decidable subset of OWL, called OWL-DL, coincides, basically, with $\mathcal{SHOIN}(\mathbf{D})$, an expressive DL [5]. Basically, OWL is based on classical logic (there is a direct mapping from $\mathcal{SHOIN}$ to first order logic) and, consequently, it adopts different semantic assumptions like the OWA, and Multiple Name Assumption. As previously pointed out, those assumptions make OWL much more suitable than OntoDLP for the semantic web (the web is an open environment), but, at the same time, they make OWL inappropriate for enterprise ontologies. This is because an enterprise ontology describes the knowledge of specific aspects of the 'closed world' of the enterprise. Thus, OntoDLP supporting both CWA and UNA results more appropriate than OWL for representing enterprise ontologies. Moreover, OntoDLP, i.e. based on ASP, natively supports 'rules' which are missing in OWL, and are considered an important tool for ontology reasoning [19, 27, 43].

Our work is also related with the attempt of combining OWL with rules for the Semantic Web, since OntoDLV natively supports an interoperability mechanism with OWL. In this field, a lot of effort has been done for finding an appropriate solution (see [3] for an excellent survey). The proposals range from the reduction of DL to logic programming [22, 29, 36, 40, 41, 44, 47], to the integration of rules in a DL specification [19, 26, 27]. The major problems existing in the interaction of rules and DLs with strict semantic integration is retaining decidability (which is, instead, ensured in our framework) without loosing ease of use and expressivity.

Our approach is a modified version of the framework presented in [14], which falls is in the 'middle' of the range of proposals. Indeed, in [14] a novel type of logic programs, so-called *dl-programs* was introduced, which allows for an information exchange between the program and an OWL ontology. dl-programs combine ASP with the DLs. In this approach, ASP works at the 'rule layer', while OWL/RDF Schema flavors would keep their purpose of description languages, in the underlying 'ontology layer'. From the rule layer point of view, ontologies are dealt with as an external source of information whose semantics is treated independently. Basically, information exchange is obtained by allowing the so-called *dl-atoms* (which contain queries to the description logic knowledge base) in the rules bodies. Non-monotonic reasoning and rules are allowed in a decidable setting, as well as

arbitrary mixing of closed and open world reasoning. In a subsequent proposal [13], this framework has been extended to a more general type of external interface, allowing for a bidirectional flow of information with any type of external source of computation (and, thus also description logics reasoners). These so-called HEX-*programs* were first presented in [13].

The OWL interface of OntoDLP extends the idea of dl-programs by allowing for querying multiple ontologies within the same program. Moreover, it supplies a wide range of ontology queries instead of the limited choice of queries in a dl-atom. On the other hand, it is more restricted than HEX-atoms, which have an arbitrary set of input and output terms. However, it differs from both dl-programs as well as HEX-programs by providing only unidirectional information flow, i.e. knowledge is only imported into the OntoDLP program. This restriction significantly simplifies the semantics of the interface. Due to the import-only type of the interface, the truth values of OWL import atoms can be entirely determined before any OntoDLP model computation. Thus, no recursion between the program and the external ontology is possible, which reduces the computational effort significantly compared to the two related approaches.

DLV$^+$. OntoDLV is rooted in the *DLV*$^+$ system [39], but, compared to its predecessor, it brings many relevant extensions, optimizations and enhancements.

As far as the language is concerned, OntoDLP supports a richer set of modelling constructs than *DLP*$^+$ (the language of *DLV*$^+$):

- classes and relations in *DLP*$^+$ are only extensional, while OntoDLP allows one to define entities intentionally (i.e. by means of rules) by exploiting collection classes (which enable objects reclassification) and intensional relations (allowing for a compact definition of relationships);
- inheritance in *DLP*$^+$ is confined to (base) classes, while all OntoDLP entities (including relations and intentional entities) support this useful modelling tool;
- OntoDLP supports lists ad sets which have no counterpart in *DLP*$^+$;
- OntoDLP natively supports an interoperability mechanism with OWL, which makes it suitable also for the development of Semantic Web applications.

Moreover, the OntoDLP language has a direct model-theoretic semantics, while the meaning of DLP+ programs was specified merely by a rewriting technique [39].

On the system side, OntoDLV, conceived for dealing with real-world applications, is more solid and better engineered than *DLV*$^+$, and it supports a number of features which have no counterpart in *DLV*$^+$, namely:

- a powerful persistency layer, which supports ontology storage in both filesystem and commercial DBMS. This feature allows one to work also with large (real-world) enterprise ontologies, which can be efficiently stored in relational databases;
- a rich API, which allows one to develop third party applications embedding OntoDLP ontologies. The OntoDLV API let the developer exploit *all* the features of the system in a comfortable way, while *DLV*$^+$ has a only limited input-output handling interface.
- OWL handling through the DIG interface which connects OntoDLV with the most popular DL reasoners;
- a more advanced Graphical Interface, which extends the one of *DLV*$^+$ with all the new features of both language and system.

In summary, OntoDLV supports both a richer language and more useful functionalities than *DLV*$^+$; moreover, it has been implemented by following industry-level development standards.

Those features makes OntoDLV much more stable, powerful, and suitable for developing real-world applications than its predecessor DLV^{+}.

7 Current applications and conclusion

This article has presented OntoDLP, an extension of (disjunctive) ASP with relevant object-oriented constructs, including classes, objects, (multiple) inheritance, lists and sets. We have described the syntax and semantics of OntoDLP and shown its usage for ontology representation and reasoning. The semantic features of the language, like CWA and UNA, its rich set of tools for ontology specification, combined with an expressive reasoning language, make OntoDLP very suitable for dealing with Enterprise/Corporate ontologies. Moreover, it supports a powerful interoperability mechanism with OWL, allowing one to simultaneously deal with both OWL and OntoDLP ontologies.

Importantly, a concrete implementation of the language has been provided: the OntoDLV system. OntoDLV features both an advanced persistency-management system, an API, and a GUI. This way, both the novice and the 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.

OntoDLV is a powerful tool 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* [42], an advanced tool for semantic information-extraction from unstructured or semi-structured documents; *OLEX* (OntoLog Enterprise Categorizer System) [10], a system developed by Exeura s.r.l. (http://www.exeura.it) for text classification (roughly, sets of documents are automatically classified by the system w.r.t. a given ontology by using suitable reasoning modules); and 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.

As far as future work is concerned, the language could be enriched with more powerful modelling constructs, and the system extended in order to carry out the evaluation of large ontologies in mass memory. Moreover, a comprehensive experimental evaluation is being set up for comparing the performance of OntoDLV with other related systems, and in particular with DL reasoners.

Further details and documentation about OntoDLV are available on the web at http://www.mat.unical.it/ricca/downloads/rt-ontodlp.zip.

Acknowledgements

This work has been 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', and by the EC NoE REWERSE (IST 506779) and the Austrian Science Fund (FWF) project P17212-N04.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, Massachusetts, 1995.

[2] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and C. Teodor Przymusinski. HiLog: a foundation for higher-order logic programming. *JLP*, **15**, 187–230, 1993.

[3] G. Antoniou, C. V. Damsio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, and P. F. Patel-Schneider. Combining rules and ontologies. *Asian survey*, **13-D3**, 2005.

[4] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Proceedings of the Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1988.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, Cambridge, 2003.

[6] S. Bechhofer, R. Möller, and P. Crowther. The dig description logic interface. In *Proceedings of the Description Logics, CEUR Workshop*, pp. 21–29. CEUR-WS.org, 2003.

[7] F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, **50**, 333–361, 2007.

[8] W. Chen, M. Kifer, and D. S. Warren. Hilog: a foundation for higher-order logic programming. *JLP*, **15**,187–230, 1993.

[9] W. Chen and D. S. Warren. C-logic of complex objects. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 369–378. ACM Press, 1989.

[10] R. Curia, M. Ettorre, S. Iiritano, and P. Rullo. Textual document per-processing and feature extraction in OLEX. In *Proceedings of Data Mining 2005*, Skiathos, Greece, 2005.

[11] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In *Proceedings of the IJCAI 2003*, pp. 847–852, Acapulco, Mexico, 2003.

[12] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM TODS*, **22**,364–418, 1997.

[13] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer set programming. In *Proceedings of the IJCAI05*, pp. 90–96. Professional Book Center, 2005.

[14] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proceedings of the KR2004*, pp. 141–151. AAAI Press, 2004.

[15] L. Gallucci, G. Grasso, N. Leone, and F. Ricca. Interoperability mechanisms for ontology management systems. In *Proceedings of CILC'07, S. Agata di Messina, 21–22 Giugno 2007*, 2007. Informal Proceedings. Available at http://aleph.unime.it/cilc2007

[16] L. Gallucci and F. Ricca. Visual querying and application programming interface for an ASP-based ontology language. In *Proceedings of the SEA'07 Workshop, Arizona, USA, 2007*, pp. 56–70. CEUR-WS.org, 1998.

[17] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *NGC*, **9**, 365–385, 1991.

[18] S. Greco, N. Leone, and P. Rullo. COMPLEX: an object-oriented logic programming system. *IEEE TKDE*, **4**, 344–359, 1992.

[19] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logics. In *Proceedings of the WWW2003, Budapest, Hungary*, pp. 48–57. ACM Press, 2003.

[20] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. In *Third International Semantic Web Conference (ISWC 2004)*. Springer-Verlag, New York, Inc., 2004.

[21] V. Haarslev and R. Möller. Racer: a core inference engine for the semantic web, In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools*, Y. Sure and O. Corcho, eds, pp. 27–36. CEUR-WS.org, 2003.

[22] S. Heymans and D. Vermeir. Integrating semantic web reasoning and answer set programming. In *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd International ASP'03 Workshop*. M. D. Vos and A. Provetti, eds, CEUR-WS.org, 2003.

[23] I. Horrocks. DAML+OIL: a description logic for the semantic web. *IEEE Bulletin of the Technical Committee on Data Engineering*, **25**, 4–9, 2002.

[24] I. Horrocks. DAML+OIL: a reason-able web ontology language. In *Proceedings of the 8th International Conference on Extending Database Technology, Advances in Database Technology (EDBT 2002)*, C. S. Jensen *et al*., eds, Vol. 2287 of *Lecture Notes in Computer Science*, Springer, 2002.

[25] I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, D. Fensel *et al*., eds, Vol. 2870 of *Lecture Notes in Computer Science*, Springer, 2003.

[26] I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *Proceedings of the 13th international conference on World Wide Web, (WWW 2004)*, pp. 723–731. ACM Press, 2004.

[27] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: a semantic web rule language combining owl and ruleml, 2004. W3C Member Submission. Available at http://www.w3.org/Submission/SWRL/ (Last accessed on 21 July 2008).

[28] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: the making of a web ontology language. *Journal of Web Semantics*, **1**, 7–26, 2003.

[29] U. Hustadt, B. Motik, and U. Sattler. Reducing shiq-description logic to disjunctive datalog programs. In *Proceedings of the KR2004*, pp. 152–162, Canada, 2004.

[30] Inc. Red Hat. The hibernate framework for object/relational persistence. 2007. Available at http://www.hibernate.org/ (Last accessed on 21 July 2008).

[31] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, **42**,741–843, 1995.

[32] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, **7**, 499–562, 2006.

[33] M. Liu, G. Dobbie, and T. W. Ling. Logical foundation for deductive object-oriented databases. *Symposium on Principles of Database Systems*, **27**, 117–151, 2002.

[34] D. Maier. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, pp. 6–26. ACM Press, 1986.

[35] Mindswap. Pellet performance, 2003. Available at http://www.mindswap.org/2003/pellet/performance.shtml (Last accessed on 21 July 2008).

[36] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *Journal of Web Semantics*, **3**,41–60, 2005.

[37] B. Parsia and E. Sirin. Pellet: an OWL DL reasoner. In *Third International Semantic Web Conference (ISWC 2004) – Posters Track*. Springer-Verlag, New York, Inc., 2004.

[38] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Proceedings of the Foundations of Deductive Databases and Logic Programming*, pp. 193–216, Morgan Kaufmann Publishers Inc., San Francisco, CA, 88.

[39] F. Ricca and N. Leone. Disjunctive logic programming with types and objects: the DLV$^+$ system. *Journal of Applied Logics*, **5**, 545–573, 2007.

[40] R. Rosati. Dl+log: tight integration of description logics and disjunctive datalog. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, P. Doherty *et al*., eds, pp. 68–78. AAAI Press, 2006.
[41] R. Rosati. Integrating ontologies and rules: semantic and computational issues. In *Proceedings of the Reasoning Web*, pp. 128–151. Springer-Verlag, New York, Inc., 2006.
[42] M. Ruffolo, N. Leone, M. Manna, D. Sacca', and A. Zavatto. Exploiting ASP for semantic information extraction. In *Proceedings of the ASP05*, Bath, UK, 2005.
[43] M. K. Smith, C. Welty, and D. L. McGuinness. OWL web ontology language guide. W3C candidate recommendation. 2003. Available at http://www.w3.org/TR/owl-guide/ (Last accessed on 21 July 2008).
[44] T. Swift. Deduction in ontologies via asp. In *LPNMR*, pp. 275–288, 2004.
[45] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: system description. In *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR 2006)*, U. Furbach and N. shankar, eds, Vol. 4130 of *Lecture Notes in Computer Science*, Springer, 2006.
[46] D. Turi. The DIG interface project. 2007. http://dig.sourceforge.net/ (Last accessed on 21 July 2008).
[47] K. Van Belleghem, M. Denecker, and D. De Schreye. A strong correspondence between description logics and open logic programming. In *Proceedings of the ICLP*, pp. 346–360, MIT Press, Massachusetts, 1997.
[48] W3C. The word wide web consortium. 1994. Available at http://www.w3.org (Last accessed on 21 July 2008).
[49] W3C. The resource description framework. 2006. Available at http://www.w3.org/RDF/ (Last accessed on 21 July 2008).
[50] G. Yang, M. Kifer, and C. Zhao. 'flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web.' In *Proceedings of the CoopIS/DOA/ODBASE*, pp. 671–688. Springer-Verlag, New York, Inc., 2003.

Appendix A: The Ontology Web Language (OWL)

OWL [43] is an ontology language for the Semantic Web, developed by the W3C [48] Web Ontology Working Group. It became a W3C Recommendation in February 2004 and is understood by the industry and the web community as a web standard. Built on top of XML and RDF(S), OWL further extends the ability of stating facts and class/property hierarchies.

The logical underpinning of OWL are DLs [5], which can be seen as fragments of first-order logic. A DL theory is commonly divided into the so-called TBox and ABox. The TBox contains the terminological axioms of the theory (corresponding to FO formulas), constituting the ontology's schema. The extensional information is established by the ABox by assertions about concept and role membership of individual instances. Query answering is defined as first-order entailment taking into account all models of TBox ∪ ABox. The atomic building blocks of DLs are *concepts* (unary relations) and *roles* (binary relations). They support inheritance and can be composed to express more complex terms. Moreover, *Value restrictions* can be used to describe cardinality constraints between concepts and roles. The semantics of DLs corresponds to a set-theoretic approach, where concepts are interpreted as sets of individuals and roles as pairs of individuals. The domain of such

an interpretation is possible infinite and adheres to the open-world assumption, in contrast to logic programming formalisms.

The naming of specific DL languages usually corresponds to the constructors they provide (in addition to the basic ones like concept union, concept disjunction, etc.). For example, in the case of $\mathcal{SHOIN}(\mathbf{D})$ these are: $\mathcal{S}$ Role transitivity, $\mathcal{H}$ Role hierarchy, $\mathcal{O}$ Nominals ('one-of'-constructor), $\mathcal{I}$ Role inverses; $\mathcal{N}$ Unqualified number restrictions; D Datatypes.

The language OWL provides the three increasingly expressive sublanguages *OWL Lite*, *OWL DL* and *OWL Full*, where OWL DL basically corresponds to DAML+OIL [23, 24]. The languages OWL Lite and OWL DL are essentially very expressive DL with an RDF/XML syntax and an abstract frame-like syntax [28].[16] In fact, as shown by [25], ontology entailment in OWL Lite and OWL DL reduces to knowledge base (un)satisfiability in the DL $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, respectively. The particular datatypes used in OWL are taken from RDF and XML Schema Datatypes. OWL Lite, being closely related to $\mathcal{SHIF}(\mathbf{D})$, prohibits unions and complements, restricts intersections to the implicit intersections in the frame-like class axioms, limits all embedded descriptions to concept names, does not allow individuals to show up in descriptions or class axioms, and limits cardinalities to 0 or 1. It therefore represents a subset of OWL DL, reducing its expressivity and hence its complexity. $\mathcal{SHOIN}(\mathbf{D})$ has a time complexity of NEXP for central reasoning problems, which is in $\mathcal{SHIF}(\mathbf{D})$ reduced to EXP in the worst case.

Received 2 January 2008

[16] Since an OWL ontology is in principle just an RDF graph, it can also be represented by RDF triples and hence be written in a variety of different syntactic forms.