Università della Calabria
Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XIX ciclo

Tesi di Dottorato

A DLP-Based System for Ontology Representation and Reasoning

Lorenzo Gallucci

Coordinatore
Prof. Domenico Talia

Supervisore
Prof. Nicola Leone

DEIS

DEIS - DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
ING-INF/05

Dedicated to Silvia, Matteo and Celeste
You are my lifeblood, my air, my energy.

Abstract

In the last few years, the need for knowledge-based technologies is emerging in several application areas. Industries are now looking for *semantic* instruments for knowledge-representation and reasoning. In this context, *ontologies* (i.e., abstract models of a complex domain) have been recognized to be a fundamental tool; and the World Wide Web Consortium (W3C) has recommended OWL [58] as a standard language for ontologies.

Some semantic assumptions of OWL, like Open World Assumption and non-Unique Name Assumption, make sense for the Web, but they are unsuited for Enterprise ontologies, that are specifications of information of business enterprises, which often evolve from relational databases, where both CWA and UNA are adopted.

The subject of this thesis is $\mathcal{O}ntoDLV$, a system based on Disjunctive Logic Programming (DLP) for the specification and reasoning on enterprise ontologies.

$\mathcal{O}ntoDLP$, the language of the system, overcomes the above-mentioned limitations of OWL, it adopts both CWA and UNA avoiding ”semantic clash” to enterprise databases.

$\mathcal{O}ntoDLP$ extends DLP with all the main ontology constructs including classes, inheritance, relations and axioms. The language is strongly typed, and includes also complex type constructors, like lists and sets.
Importantly, $\mathcal{O}ntoDLV$ supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information also from OWL Ontologies and to reason on top of that by exploiting $\mathcal{O}ntoDLP$ powerful deduction rules. The system is endowed with a powerful Application Programming Interface and is already used in a number of real-world applications, including agent-based systems and information extraction applications.

Rende (CS),
November 2007

Lorenzo Gallucci

Acknowledgements

A Ph.D. thesis work is a complex matter; but, it can also become an exciting challenge, if one finds the right people to work with.

This is what happened to me when I first met Nicola Leone.

He is a formidable researcher, an honest person, and first of all he is for me a **sincere friend**. It has been definitely a pleasure to work with him.

Then, I was blessed to work with the $\mathcal{O}ntoDLV$ team.

Thanks to Tina $Dell'$Armi, because she never gives up; most important, she never backs away hard work; the helm could never be in better hands, captain.

Thanks to Giovanni Grasso, because he is kind, but wise and determined; I can't remember how many times you saved me when I was wrong.

Thanks to Francesco Ricca, because he can see the point in someone other's thought; you taught me that this is the only way to achieve great results.

Thanks to Roman Schindlauer, because he is crisp, though modest; it is a rare match, definitely.

I want also thank Francesco Calimeri and Susanna Cozza, the team of "DLV with External Sources of Computation".

Thanks to Giorgio Terracina and Claudio Panetta, the team of DLV^{DB}.

Contents

1 Introduction

1.1 Motivation

In the last few years, the need for knowledge-based technologies is emerging in several application areas. Industries are now looking for *semantic* instruments for knowledge-representation and reasoning. In this context, *ontologies* (i.e., abstract models of a complex domain) have been recognized to be a fundamental tool; and the World Wide Web Consortium (W3C) has already provided recommendations and standards related to ontologies, like, e.g., RDF(s) [64] and OWL [58]. In particular, OWL has been conceived for the Semantic-Web, with the goal to enrich web pages with machine-understandable descriptions of the presented contents (the so-called Web ontologies). OWL is based on expressive Description Logics (DL)[6]; distinguishing features of its semantics are the adoption of the Open World Assumption and the non-uniqueness of the names (the same individual can be denoted by different names).

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

The enterprise ontology of a food-distribution company stores its pasta suppliers and their respective production branches in the relation depicted in Table 1.1 (of the company database).

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

Table 1.1. The Supplier-Branch table.

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

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

Similar scenarios are frequent when we deal with enterprise ontologies. In these cases logic programming languages like DLP, strongly relying on CWA and UNA, are definitely more appropriate than OWL. Disjunctive Logic Programming [30], 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, DLP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [23]. However, DLP is, somehow, a "low-level" formalism for ontologies since in its classical formulation it does not directly support the most common ontology concepts like classes, inheritance, individuals, etc. Moreover, DLP systems are far away from comfortably enabling the development of industry-level applications, mainly because they miss important tools for supporting users and programmers. In particular, friendly user interfaces are missing, and there is a lack of advanced Application Programming Interfaces (API) for implementing applications on top of DLP systems.

1.2 Contribution of the Thesis

This thesis work describes $\mathcal{O}ntoDLV$, a DLP-based system for knowledge modeling and advanced knowledge-based reasoning, which addresses all the above-mentioned issues.

On the one hand, $\mathcal{O}ntoDLV$ overcomes the semantic clash to database semantics of OWL, thanks to the adoption of CWA and UNA of DLP.

On the other hand, $\mathcal{O}ntoDLV$ enhances DLP with the most relevant ontology constructs.

Indeed, $\mathcal{O}ntoDLV$ implements a powerful logic-based ontology representation language, called $\mathcal{O}ntoDLP$, which is an extension of DLP with all the main ontology constructs including classes, relations, inheritance, and axioms. $\mathcal{O}ntoDLP$ is strongly typed, and includes also complex type constructors, like lists and sets.

Importantly, $\mathcal{O}ntoDLV$ supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information also from OWL Ontologies and to exploit this information in $\mathcal{O}ntoDLP$ ontologies [1].

Moreover, $\mathcal{O}ntoDLV$ allows the development of complex applications in a user-friendly visual environment; and it seamlessly integrates the DLV system [40] exploiting the power of a stable and efficient DLP solver. The system is also able to seamlessly switch to DLV^{DB} [61], taking advantage of relational database capabilities.

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

The main contributions of the thesis can be grouped in three categories:

- **Language**. We defined a powerful language, called $\mathcal{O}ntoDLP$, for the representation and reasoning on enterprise ontologies. The language complies with the semantic assumption of databases and is endowed with interoperability mechanisms to OWL.
- **Algorithms**. We designed a bunch of algorithms and optimization techniques for the efficient implementation of the proposed language.
- **System and implementation**. We implemented the $\mathcal{O}ntoDLV$ system, fully supporting the $\mathcal{O}ntoDLP$ language. The system is endowed with a friendly visual interface and powerful API for the development of knowledge-based applications.

Both the language ($\mathcal{O}ntoDLP$) and the system ($\mathcal{O}ntoDLV$) have features that make them well suited for managing enterprise ontologies. Among the features of the language, we recall:

- Ontology constructs, such as classes, relations, (multiple) inheritance: essential to build a domain representation;
- Axioms: a way to enforce properties that must hold in a consistent domain;
- Modular programming: logic programs defining reasoning tasks may be split over different reasoning modules;
- Rich set of data types: $\mathcal{O}ntoDLP$ can handle strings, integers, decimal numbers and dates. In addition to simple operations, also aggregate functions can be applied;
- Complex types (e.g., lists and sets): the language has the ability to build and explore complex, nested data structures and use them as values;

[1] It is well known that rule-based inference systems are needed by OWL applications [34, 36].

- Objects reclassification support (collection classes), intensionally defined relations: entities which are defined by specifying the properties they hold, rather than explicitly enumerated;
- Meta reasoning capabilities: the language offers capabilities to reason on the structure of the ontology, rather than on the data contained therein;
- OWL interoperability mechanisms at language level: $\mathcal{O}ntoDLP$ supports a kind of *OWL atoms*, able to gather knowledge from elsewhere stored OWL ontologies

The system also features:

- Application Programming Interface (API): all $\mathcal{O}ntoDLV$ functionalities are accessible through an easy-to-use programming interface that exposes concepts at a higher level;
- Modular architecture, based on an extensible ontology storage engine: the system can use (and mix) different storage engines (on filesystem, on a relational database, etc.), for different parts of the ontology;
- Mass-memory query execution: $\mathcal{O}ntoDLV$ supports the direct execution of queries in mass memory on (possibly remote) relational databases;
- OWL interoperability mechanism at system level: $\mathcal{O}ntoDLV$ has the ability to import (from OWL) and export (to OWL) any ontology, keeping entailment compatibility for a significant fragment of OWL language;

1.3 Organization

The rest of this work is organized as follows:

In the first part, we focus on the characteristics of the language, $\mathcal{O}ntoDLP$.

In Chapter 2 we introduce some preliminaries about the domain of logic programming in general and disjunctive logic programming in particular. Chapter 3 describes the language $\mathcal{O}ntoDLP$, highlighting its base characteristics, the ability to define intensional entities, enhanced data types, lists and sets, name extensions, and external knowledge gathering constructs. The ability to reason on structure of the ontology, called *meta reasoning*, is described in Chapter 4.

The mechanisms for achieving interoperability with OWL are illustrated in Chapter 5; we describe therein import/export facility and the ability to reason on top of OWL ontologies, via *OWL atoms*.

In the second part we describe the system, $\mathcal{O}ntoDLV$.

Architecture and capabilities of the system are described in Chapter 6.

Base principles and features of Application Programming Interface are taken into account in Chapter 7.

Eventually, in the third part, Chapter 8 compares our work with a number of languages and systems, related to $\mathcal{O}ntoDLP$ and $\mathcal{O}ntoDLV$, already proposed in the literature.

Chapter 9 concludes this thesis.

Part I

The language: $\mathcal{O}ntoDLP$

2

Disjunctive logic programming (DLP)

In this chapter, we provide a formal definition of the syntax and the semantics of Disjunctive Logic Programming (DLP), which our $\mathcal{O}ntoDLP$ language is built on. For further background, see [44, 23, 30].

2.1 Syntax

A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *literal* is a *positive literal* p or a *negative literal* not p, where p is an atom.

A *disjunctive rule* (*rule*, for short) $\mathcal{R}$ is a formula

$$a_1 \texttt{ v } \cdots \texttt{ v } a_n \text{ :- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m. \tag{2.1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \texttt{ v } \cdots \texttt{ v } a_n$ is called *head* of $\mathcal{R}$, while the conjunction $b_1, \cdots, b_k$, not $b_{k+1}, \cdots,$ not b_m is the *body* of $\mathcal{R}$. We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(\mathcal{R})$ the set of the body literals. In particular, $B(\mathcal{R}) = B^+(\mathcal{R}) \cup B^-(\mathcal{R})$, where $B^+(\mathcal{R})$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(\mathcal{R})$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and we usually omit the " :- " sign.

An *(integrity) constraint* is a rule without head literals (i.e. $n = 0$)

$$\text{:- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m. \tag{2.2}$$

A *disjunctive logic program* (often simply DLP program) $\mathcal{P}$ is a finite set of rules (possibly including integrity constraints), and $Rules(\mathcal{P})$ denotes the set of rules (including integrity constraints) in $\mathcal{P}$. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*, and a v-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *normal logic program.*

A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe, if each of its rules is safe, and in the following we will only consider safe programs.

A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. A ground program is also called a *propositional* program.

Given a literal l, let $\text{not}.l = a$ if $l = \text{not } a$, otherwise $\text{not}.l = \text{not } l$, and given a set L of literals, $\text{not}.L = \{\text{not}.l \mid l \in L\}$.

Example 2.1. For example consider the following program:

$$\begin{aligned}
&r_1 : a(X) \vee b(X) \ {:\!-}\ c(X,Y), d(Y), \text{not } e(X).\\
&r_2 : \ {:\!-}\ c(X,Y), k(Y), e(X), \text{not } b(X).\\
&r_3 : m \ {:\!-}\ n, o, a(1).\\
&r_4 : c(1,2).
\end{aligned}$$

r_1 is a disjunctive rule s.t. $H(r_1) = \{a(X), b(X)\}$, $B^+(r_1) = \{c(X,Y), d(Y)\}$, and $B^-(r_1) = \{e(X)\}$; r_2 is a constraint s.t. $B^+(r_2) = \{c(X,Y), k(Y), e(X)\}$, and $B^-(r_2) = \{b(X)\}$; r_3 is a ground positive (non-disjunctive) rule s.t. $H(r_3) = \{m\}$ $B^+(r_3) = \{n, o, a(1)\}$, and $B^-(r_3) = \emptyset$; r_4 is a fact (note that $:\!-$ is omitted).

Relevant Classes of Programs

In this paragraph, we introduce syntactic classes of disjunctive logic programs with interesting properties. First we need the following:

Definition 2.2. Functions $||\ || : B_{\mathcal{P}} \rightarrow \{0, 1, \ldots\}$ from the Herbrand Base $B_{\mathcal{P}}$ to finite ordinals are called *level mappings* of $\mathcal{P}$.

Level mappings give a useful technique for describing various classes of programs.

Definition 2.3. A disjunctive logic program $\mathcal{P}$ is called *(locally) stratified* [5, 50], if there is a level mapping $||\ ||_s$ of $\mathcal{P}$ such that, for every rule r of $Ground(\mathcal{P})$[1],

1. for any $l \in B^+(\mathcal{R})$, and for any $l' \in H(r)$, $||l||_s \leq ||l'||_s$;
2. for any $l \in B^-(\mathcal{R})$, and for any $l' \in H(r)$, $||l||_s < ||l'||_s$.
3. for any $l, l' \in H(r)$, $||l||_s = ||l'||_s$.

Example 2.4. Consider the following two programs.

$$\mathcal{P}_1 = \{p(a) \vee p(c) :\!-\text{not } q(a). \ ; \ p(b) :\!-\text{not } q(b).\}$$

$$\mathcal{P}_2 = \{p(a) \vee p(c) :\!-\text{not } q(b). \ ; \ q(b) :\!-\text{not } p(a).\}$$

It is easy to see that program $\mathcal{P}_1$ is stratified, while program $\mathcal{P}_2$ is not. A suitable level mapping for $\mathcal{P}_1$ is the following:

[1] See 2.2 for a definition of $Ground(\mathcal{P})$

$$||p(a)||_s = 2 \; ||p(b)||_s = 2 \; ||p(c)||_s = 2$$
$$||q(a)||_s = 1 \; ||q(b)||_s = 1 \; ||q(c)||_s = 1$$

As for $\mathcal{P}_2$, an admissible level mapping would need to satisfy $||p(a)||_s < ||q(b)||_s$ and $||q(b)||_s < ||p(a)||_s$, which is impossible.

Another interesting class of problems consists of head-cycle free programs.

Definition 2.5. A program $\mathcal{P}$ is called *head-cycle free (HCF)* [10, 11], if there is a level mapping $|| \; ||_h$ of $\mathcal{P}$ such that, for every rule r of $Ground(\mathcal{P})$,

1. for any $l \in B^+(\mathcal{R})$, and for any $l' \in H(r)$, $||l||_h \leq ||l'||_h$;
2. for any pair $l, l' \in H(r)$ $||l||_h \neq ||l'||_h$.

Example 2.6. Consider the following program $\mathcal{P}_3$.

$$\mathcal{P}_3 := \{a \vee b. \;\; ; \;\; a \text{:-} \, b.\}$$

It is easy to see that $\mathcal{P}_3$ is head-cycle free; an admissible level mapping for $\mathcal{P}_3$ is given by $||a||_h = 2$ and $||b||_h = 1$. Consider now the program

$$\mathcal{P}_4 = \mathcal{P}_3 \cup \{b \text{:-} \, a.\}$$

$\mathcal{P}_4$ is not head-cycle free, since a and b should belong to the same level by Condition (1) of Definition 2.5, while they cannot by Condition (2) of that definition. Note, however, that $\mathcal{P}_4$ is stratified.

Another characterization of the HCF property is given in term of the so called *dependency graph*.

Definition 2.7. With every ground program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (N, E)$, called the *dependency graph* of $\mathcal{P}$, in which (i) each atom of $\mathcal{P}$ is a node in N and (ii) there is an arc in E directed from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b and a appear in the head and positive body of r, respectively.

The graph $DG_{\mathcal{P}}$ singles out the dependencies of the head atoms of a rule r on the positive atoms in its body.[2]

Example 2.8. Consider the programs

$$\mathcal{P}_5 = \{a \vee b. \;\; ; \;\; c \text{:-} \, a. \;\; ; \;\; c \text{:-} \, b.\}$$
$$\mathcal{P}_6 = \mathcal{P}_5 \cup \{d \vee e \text{:-} \, a. \;\; ; \;\; d \text{:-} \, e. \;\; ; \;\; e \text{:-} \, d, \text{not } b.\}.$$

The dependency graph $DG_{\mathcal{P}_5}$ of $\mathcal{P}_5$ is depicted in Figure 2.1 (a), while the dependency graph $DG_{\mathcal{P}_6}$ of $\mathcal{P}_6$ is depicted in Figure 2.1 (b).

[2] Note that negative literals cause no arc in $DG_{\mathcal{P}}$.

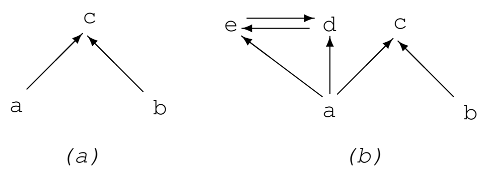

Fig. 2.1. *Graphs (a) $DG_{\mathcal{P}_4}$, and (b) $DG_{\mathcal{P}_5}$*

Definition 2.9. *[10, 11]* A program $\mathcal{P}$ is *head-cycle-free* (*HCF*) iff there is no rule r in $\mathcal{P}$ such that two atoms occurring in the head of r are in the same cycle of $DG_{\mathcal{P}}$.

Example 2.10. Considering Example 2.8, the dependency graphs given in Figure 2.1 reveal that program $\mathcal{P}_4$ is HCF and that program $\mathcal{P}_5$ is not HCF, as rule $d \vee e \text{:-} \, a.$ contains in its head two atoms belonging to the same cycle of $DG_{\mathcal{P}_5}$.

Another refinement of this property is given in term of the *components* of the dependency graph.

Definition 2.11. A *component* C of a dependency graph DG is a maximal subgraph of DG such that each node in C is reachable from any other. The *subprogram* of C consists of all rules having some atom from C in the head. An atom is non-HCF if the subprogram of its component is non-HCF.

2.2 Semantics

The semantics of a disjunctive logic program is given by its stable models [49], which we briefly review in this section.

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible ground atoms that can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_{\mathcal{P}}$.

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

Stable Models

For every program $\mathcal{P}$, we define its stable models using its ground instantiation in two steps: First we define the stable models of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define stable models of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\mathrm{not}\ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I.

Let $\mathcal{R}$ be a ground rule in $\mathcal{P}$. The head of $\mathcal{R}$ is *true* w.r.t. I if exists $a \in H(r)$ s.t. a is true w.r.t. I (i.e., some atom in $H(r)$ is true w.r.t. I). The body of $\mathcal{R}$ is *true* w.r.t. I if $\forall \ell \in B(r)$, ℓ is true w.r.t. I (i.e. all literals on $B(r)$ are true w.r.t I). The body of $\mathcal{R}$ is *false* w.r.t. I if $\exists \ell \in B(r)$ s.t. ℓ is false w.r.t I (i.e., some literal in $B(r)$ is false w.r.t. I). The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

Interpretation I is *total* if, for each atom A in $B_{\mathcal{P}}$, either A or $\mathrm{not}.A$ is in I (i.e., no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $\mathcal{R} \in \mathcal{P}$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is a *stable model* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

Example 2.12. Consider the positive programs:

$$\mathcal{P}_1 = \{a \vee b \vee c.\ ;\ \ {:-}\, a.\}$$
$$\mathcal{P}_2 = \{a \vee b \vee c.\ ;\ \ {:-}\, a.\ ;\ \ b \,{:-}\, c.\ ;\ \ c \,{:-}\, b.\}$$

The stable models of $\mathcal{P}_1$ are $\{b, \mathrm{not}\ a, \mathrm{not}\ c\}$ and $\{c, \mathrm{not}\ a, \mathrm{not}\ b\}$, while $\{b, c, \mathrm{not}\ a\}$ is the only stable model of $\mathcal{P}_2$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all rules $\mathcal{R} \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules.

A stable model of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is a stable model of $\mathcal{P}^X$.

Example 2.13. Given the (general) program

$$\mathcal{P}_3 = \{$$
$$a \vee b \,{:-}\, c.\ ;$$
$$b \,{:-}\, \mathrm{not}\ a, \mathrm{not}\ c.\ ;$$
$$a \vee c \,{:-}\, \mathrm{not}\ b.$$
$$\}$$

and the interpretation $I = \{b, \mathrm{not}\ a, \mathrm{not}\ c\}$, the reduct $\mathcal{P}_3^I$ is $\{a \vee b \,{:-}\, c., b.\}$. I is a stable model of $\mathcal{P}_3^I$, and for this reason it is also a stable model of $\mathcal{P}_3$. Now consider $J = \{a, \mathrm{not}\ b, \mathrm{not}\ c\}$. The reduct $\mathcal{P}_3^J$ is $\{a \vee b \,{:-}\, c.\ ;\ \ a \vee c.\}$ and it can be easily verified that J is a stable model of $\mathcal{P}_3^J$, so it is also a stable model of $\mathcal{P}_3$.

2.3 Some DLP Properties

In this section, we recall some important properties of (ground) DLP programs.

Definition 2.14. Given an interpretation I for a ground program $\mathcal{P}$, we say that a ground atom A is *supported* in I if there is a *supporting* rule $r \in ground(\mathcal{P})$, i.e. the body of r is true w.r.t. I and A is the only true atom in the head of r.

Proposition 2.15. [45, 42, 8] If M is a stable model of a program $\mathcal{P}$, then all atoms in M are supported.

Example 2.16. Consider the program $\mathcal{P}_2$ of Example 2.12, and its stable model $M = \{b, c, \text{not } a\}$. We have that M contains two atoms (b and c), and both of them are supported, in fact: the rule b :- c supports b, and the rule c :- b supports c.

Another important property of stable models is related to the notion of *unfounded set* [63, 42].

Definition 2.17. Let I be a (partial) interpretation for a ground program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an unfounded set for $\mathcal{P}$ w.r.t. I if, for each $a \in X$ and for each rule $r \in \mathcal{P}$ such that $a \in H(r)$, at least one of the following conditions holds: (i) $B(r) \cap \text{not}.I \neq \emptyset$, (ii) $B^+(r) \cap X \neq \emptyset$, (iii) $(H(r) - X) \cap I \neq \emptyset$.

Let $\mathcal{II}$ denote the set of all interpretations of $\mathcal{P}$ for which the union of all unfounded sets for $\mathcal{P}$ w.r.t. I is an unfounded set for $\mathcal{P}$ w.r.t. I as well[3]. Given $I \in \mathcal{II}$, let $GUS_{(I)}$ (the *greatest unfounded set* of $\mathcal{P}$ w.r.t. I) denote the union of all unfounded sets for $\mathcal{P}$ w.r.t. I.

Proposition 2.18. [42] If M is a total interpretation for a program $\mathcal{P}$. M is a stable model of $\mathcal{P}$ iff $\text{not}.M = GUS_{(M)}$.

2.4 Computational Complexity

In this section, we recall the computational complexity of Disjunctive Logic Programming. We first provide some preliminaries on the complexity theory. Then, we define the main computational problems under consideration and illustrate their precise complexity.

2.4.1 The Polynomial Hierarchy

We assume that the reader is familiar with the concepts of NP-completeness and complexity theory and so we provide only a very short reminder of the complexity classes of the Polynomial Hierarchy which are relevant to this chapter. For further details, the reader is referred to [48].

[3] While for non-disjunctive programs the union of unfounded sets is an unfounded set for all interpretations, this does not hold for disjunctive programs (see [42]).

The classes Σ_k^P, Π_k^P, and Δ_k^P of the *Polynomial Hierarchy* (PH, cf. [38]) are defined as follows:

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathrm{P}$$

$$\text{and for all } k \geq 1,\ \Delta_k^P = \mathrm{P}^{\Sigma_{k-1}^P}, \Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}, \Pi_k^P = \text{co-}\Sigma_k^P,$$

where NP^C denotes the class of decision problems that are solvable in polynomial time on a nondeterministic Turing machine with an oracle for any decision problem π in the class C. In particular, NP $= \Sigma_1^P$, co-NP $= \Pi_1^P$, and $\Delta_2^P = \mathrm{P}^{\mathrm{NP}}$.

The oracle replies to a query in unit time, and thus, roughly speaking, models a call to a subroutine for π that is evaluated in unit time.

Observe that for all $k \geq 1$,

$$\Sigma_k^P \subseteq \Delta_{k+1}^P \subseteq \Sigma_{k+1}^P \subseteq \mathrm{PSPACE}$$

where each inclusion is widely conjectured to be strict. By the rightmost inclusion above, all these classes contain only problems that are solvable in polynomial space. They allow, however, a finer-grained distinction among NP-hard problems that are in PSPACE.

2.4.2 Complexity of the Main DLP Decision Problems

Three important decision problems, corresponding to three different reasoning tasks, arise in the context of Disjunctive Logic Programming:

Brave Reasoning. Given a program $\mathcal{P}$, and a ground atom A, decide whether A is true in some stable model of $\mathcal{P}$ (denoted $\mathcal{P} \models_b A$).
Cautious Reasoning. Given a program $\mathcal{P}$, and a ground atom A, decide whether A is true in all stable models of $\mathcal{P}$ (denoted $\mathcal{P} \models_c A$).
Stability Checking. Given a program $\mathcal{P}$, and a set M of ground literals as input, decide whether M is a stable model of $\mathcal{P}$.

We summarize the complexity of these decision problems for ground (i.e., propositional) DLP programs; we shall address the case of non-ground programs at the end of this section.

An interesting issue is the impact of syntactic restrictions on the logic program $\mathcal{P}$. Starting from normal positive programs (without negation and disjunction), we consider the effect of allowing the (combined) use of the following constructs:

- stratified negation (not_s),
- arbitrary negation (not),
- head-cycle free disjunction ($\mathtt{v_h}$),
- arbitrary disjunction ($\mathtt{v}$).

Given a set X of the above syntactic elements (with at most one negation and at most one disjunction symbol in X), we denote by DLP[X] the fragment of DLP where the elements in X are allowed. For instance, DLP[$\mathtt{v_h}, \mathrm{not}_s$] denotes the fragment allowing head-cycle free disjunction and stratified negation.

	$\{\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not}\}$
$\{\}$	P	P	NP
$\{\mathtt{v_h}\}$	NP	NP	NP
$\{\mathtt{v}\}$	Σ_2^P	Σ_2^P	Σ_2^P

Table 2.1. The Complexity of Brave Reasoning in fragments of DLP

	$\{\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not}\}$
$\{\}$	P	P	co-NP
$\{\mathtt{v_h}\}$	co-NP	co-NP	co-NP
$\{\mathtt{v}\}$	co-NP	Π_2^P	Π_2^P

Table 2.2. The Complexity of Cautious Reasoning in fragments of DLP

	$\{\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not}\}$
$\{\}$	P	P	P
$\{\mathtt{v_h}\}$	P	P	P
$\{\mathtt{v}\}$	co-NP	co-NP	co-NP

Table 2.3. The Complexity of Stability Checking in fragments of DLP

The complexity of Brave Reasoning and Cautious Reasoning from ground DLP programs are summarized in Table 2.1 and Table 2.2, respectively. Table 2.3, shows the results on the complexity of Stability Checking.

The rows of the tables specify the form of disjunction allowed; in particular, $\{\}$ = no disjunction, $\{\mathtt{v_h}\}$ = head-cycle free disjunction, and $\{\mathtt{v}\}$ = unrestricted (possibly not head-cycle free) disjunction. The columns specify the support for negation. In detail, $\{\}, \{\mathrm{not}_s\}$ and $\{\mathrm{not}\}$ denotes positive programs, negation as failure and arbitrary negation respectively. Each entry of the table provides the complexity of the corresponding fragment of the language, in terms of a completeness result. For instance, $(\{\mathtt{v_h}\}, \{\mathrm{not}_s\})$ is the fragment allowing head-cycle free disjunction and stratified negation. The corresponding entry in Table 2.1, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under polynomial time (and in fact LOGSPACE) reductions. All results have been proved in [23, 31, 24, 21, 13]. Furthermore, not all complexity results in the quoted papers were explicitly stated for LOGSPACE reductions, but can be easily seen to hold from (suitably adapted) proofs.

Looking at Table 2.1, it can be seen that limiting the form of disjunction and negation reduces the respective complexity. For disjunction-free programs, brave reasoning is polynomial on stratified negation, while it becomes NP-complete if unrestricted (nonmonotonic) negation is allowed. Brave reasoning is NP-complete on head-cycle free programs even if no form of negation is allowed. The complexity jumps one level higher in the Polynomial Hierarchy, up to Σ_2^P-complexity, if full disjunction is allowed. Thus, disjunction seems to be harder than negation, since the full complexity is reached already on positive programs, even without any kind of negation.

Table 2.2 contains results for cautious reasoning. One would expect its complexity to be symmetric to the complexity of brave reasoning, that is, whenever the complexity of a fragment is C under brave reasoning, one expects its complexity to be co-C under cautious reasoning (recall that co-P $=$ P, and co-$\Sigma_2^P = \Pi_2^P$).

Surprisingly, there is one exception: While full disjunction raises the complexity of brave reasoning from NP to Σ_2^P, full disjunction alone is not sufficient to raise the complexity of cautious reasoning from co-NP to Π_2^P. Cautious reasoning remains in co-NP if default negation is disallowed. Intuitively, to disprove that an atom A is a cautious consequence of a program $\mathcal{P}$, it is sufficient to find *any model* M of $\mathcal{P}$ (which need not be a stable model or a minimal model) which does not contain A. For not-free programs, the existence of such a model guarantees the existence of a subset of M which is a stable model of $\mathcal{P}$ (and does not contain A).

The complexity results for Stability Checking, reported in Table 2.3, help us to understand the complexity of reasoning. Whenever Stability Checking is co-NP-complete for a fragment F, the complexity of brave reasoning jumps up to the second level of the Polynomial Hierarchy (Σ_2^P). Indeed, brave reasoning on full DLP programs suffers from two sources of complexity:

(s_1)the exponential number of model "candidates",
(s_2)the difficulty of checking whether a candidate M is stable (the minimality of M can be disproved by an exponential number of subsets of M).

Now, disjunction (unrestricted or even head-cycle free) or unrestricted negation preserve the existence of source (s_1), while source (s_2) exists only if full disjunction is allowed (see Table 2.3).

As a consequence, e.g., the complexity of brave reasoning is the highest (Σ_2^P) in the fragments preserving both the two sources of complexity (where both full disjunction and unrestricted negation are allowed). The complexity goes down to the first level of PH if source (s_2) is eliminated; avoiding source (s_1) the complexity falls down to P, as (s_2) is automatically eliminated. Finally, we note that HCF disjunction preserves the tractability of the Stability Checking

We close this section with briefly addressing the complexity and expressiveness of non-ground programs. A non-ground program $\mathcal{P}$ can be reduced, by naive instantiation, to a ground instance of the problem. The complexity of this ground instantiation is as described above. In the general case, where $\mathcal{P}$ is given in the input, the size of the grounding $Ground(\mathcal{P})$ is single exponential in the size of $\mathcal{P}$. Informally, the complexity of Brave Reasoning and Cautious Reasoning increases accordingly by one exponential, from P to EXPTIME, NP to NEXPTIME, Σ_2^P to NEXPTIME$^{\mathrm{NP}}$, etc. For disjunctive programs and certain fragments of DLP, complexity results in the non-ground case have been derived e.g. in [23, 24]. For the other fragments, the results can be derived using complexity upgrading techniques [23, 32].

3

A DLP-based language for ontology representation and reasoning: $\mathcal{O}ntoDLP$

The role of a knowledge representation language is to capture domain knowledge and provide a commonly agreed upon understanding of a domain. The specification of a common vocabulary defining the meaning of terms and their relations, usually modeled by using primitives such as concepts organized in taxonomies, relations, and axioms is commonly called an *ontology*.

In this chapter, we present $\mathcal{O}ntoDLP$, an extension of Disjunctive Logic Programming (DLP) aimed at representing ontologies and enabling reasoning tasks on them.

Organization

The remainder of this chapter is structured as follows. In Sec. 3.1, we present an overview of the $\mathcal{O}ntoDLP$ language with respect to extensional ontology entities (i.e., entities whose instances are defined explicitly and independently).

Defining properties of instances is crucial to domain modeling; we describe the choices available for properties' types in Sec. 3.2.

Importantly, $\mathcal{O}ntoDLP$ allows the usage of entities described by rules, rather than by an explicit enumeration of instances; *intensional entities* are described in Sec. 3.3.

Axioms and queries are presented in Sec. 3.4, together with reasoning modules. In reasoning modules unstratified negation and disjunction are permitted, thus enhancing complexity up to Σ_2^P, allowing for direct encoding of complex problems.

3.1 Extensional ontology entities

An ontology in $\mathcal{O}ntoDLP$ can be specified by means of *base classes*, and *base relations*.

In the following subsections we describe these fundamental constructs; for a better understanding, we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

3.1.1 Base classes

One of the most powerful abstraction mechanism for the representation of a knowledge domain is *classification*, i.e., the process of identifying object categories (*classes*), on the basis of the observation of common properties (*class attributes*).

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

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*. Those classes can be defined in $\mathcal{O}ntoDLP$ as follows:

$$\textbf{class}\ person.$$

$$\textbf{class}\ animal.$$

$$\textbf{class}\ food.$$

$$\textbf{class}\ place.$$

The simplest way to declare a class is, hence, to specify the class name, preceded by the keyword **class**[1]. However, when we recognize a class in a knowledge domain, we also identify a number of properties or attributes which are defined for all the individuals belonging to that class.

A class attribute can be specified in $\mathcal{O}ntoDLP$ by means of a pair *(attribute-name : attribute-type)*, where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

For instance, we can enrich the specification of the class *person* by the definition of some properties that are common to each person: the name, age, father, mother, and birthplace.

Note that many properties can be represented by using alphanumeric strings and numbers. To this end, $\mathcal{O}ntoDLP$ features the built-in classes *string* and *integer*, respectively representing the class of all alphanumeric strings and the class of integer numbers. Additionally, types like decimals, dates plus "container types" are available; they are discussed later in this chapter, in Sec. 3.2.1 and 3.2.3.

Thus, the class person can be better modeled as follows:

$$\begin{array}{l}\textbf{class}\ person(\\ \quad name : string,\\ \quad age : integer,\\ \quad father : person,\\ \quad mother : person,\\ \quad birthplace : place).\end{array}$$

[1] The keyword **class** may be, in fact, preceded by keyword **base**; however, the latter is often omitted, for brevity.

Note that this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e., objects made of other objects[2]. It is worth noting that attributes model the properties that *must* be present in all class instances; properties that, *might* be present or not should be modeled, as will be shown later, by using relations[3].

In the same way, we could enrich the specification of the other above mentioned classes in our domain by adding some attributes. For instance, we could have a name for each *place*, *food* and *animal*, an age for each animal etc.

class $place(name : string).$

class $food(name : string,\ origin : place).$

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

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

Next section illustrates how we represent individuals in $\mathcal{O}ntoDLP$.

3.1.2 Objects

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

Each individual in $\mathcal{O}ntoDLP$ belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, we declare that "Rome" is an instance of the class *place* as follows:

$$rome : place(name : "Rome").$$

Note that, when we declare an instance, we immediately give an oid to the instance (in this case is *rome*), and a value to the attributes (in this case the *name* is the string "Rome").

The oid *rome* can now be used to refer to that place (e.g., when we have to fill an attribute of another object). Suppose that, in our *living being* domain, there is a person (i.e., an instance of the class person) whose name is "John". John is 34 years old, lives in Rome, his father and his mother are identified by *jack* and *ann* respectively. We can declare this instance as follows:

[2] Unnamed aggregation of objects may be defined also via container types, see Sec. 3.2.3

[3] In other words, an attribute $(n : k)$ of a class c is a total function from c to k; while partial functions from c to k can be represented by a binary relation on (c, k).

$$
\begin{array}{l}
john : person(\\
\quad name : "John", \\
\quad age : 34, \\
\quad father : jack, \\
\quad mother : ann, \\
\quad birthplace : rome).
\end{array}
$$

In this case, “*john*” is *the object identifier* of this instance, while “*jack*”, “*ann*”, and “*rome*” are suitable oid’s respectively filling the attributes *father*, *mother* (both of type *person*) and *birthplace* (of type *place*).

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

3.1.3 Inheritance

Another relevant abstraction tool in the the field of knowledge representation is the *specialization/generalization* mechanism, allowing to organize concepts of a knowledge domain in a taxonomy. This is obtained in the object-oriented languages by using the well-known mechanism of inheritance.

Inheritance is supported by $\mathcal{O}ntoDLP$, and class hierarchies can be specified by using the special binary relation isa.

For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in $\mathcal{O}ntoDLP$ as follows:

$$
\begin{array}{l}
\textbf{class}\ student\ \textbf{isa}\ person(\\
\quad code : string, \\
\quad school : string, \\
\quad tutor : person). \\
\\
\textbf{class}\ employee\ \textbf{isa}\ person(\\
\quad salary : integer, \\
\quad skill : string, \\
\quad company : string, \\
\quad tutor : employee).
\end{array}
$$

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

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

For example, consider the following two instances of *student* and *employee*:

$$\begin{array}{l} al : student(name : "Alfred", \\ \quad age : 20, \\ \quad father : jack, \\ \quad mother : betty, \\ \quad birthplace : rome, \\ \quad code : "100", \\ \quad school : "Cambridge", \\ \quad tutor : hanna). \end{array}$$

$$\begin{array}{l} jack : employee(name : "Jack", \\ \quad age : 54, \\ \quad father : jim, \\ \quad mother : mary, \\ \quad birthplace : rome, \\ \quad salary : 1000, \\ \quad skill : "Javaprogrammer", \\ \quad company : "SUN", \\ \quad tutor : betty). \end{array}$$

They are automatically considered also instances of person as follows:

$$\begin{array}{l} al : person(name : "Alfred", \\ \quad age : 20, \\ \quad father : jack, \\ \quad mother : betty, \\ \quad birthplace : rome). \end{array}$$

$$\begin{array}{l} jack : person(name : "Jack", \\ \quad age : 54, \\ \quad father : jim, \\ \quad mother : mary, \\ \quad birthplace : rome). \end{array}$$

Note that we do not need to assert the above two instances, both *al* and *jack* are automatically considered instances of *person*.

In $\mathcal{O}ntoDLP$ there is no limitation on the number of superclasses (i.e., multiple inheritance is allowed). Thus, a class can be a specialization of any number of classes, and, consequently, it inherits all the attributes of its superclasses.

As an example, consider the following declaration:

$$\begin{array}{l} \textbf{class}\ stud\_emp\ \textbf{isa}\ \{student, employee\}(\\ \quad workload : integer). \end{array}$$

So, the class *stud_emp* (exploiting multiple inheritance) is a subclass of both *student* and *employee*. Note that, the attribute tutor is defined in both *student*, with type *student*, and *employee* with type *employee* [4].

In this case, the attribute *tutor* will be taken only once in the scheme of *stud_emp*, but it is not intuitive what type will be taken for it.

This tricky situation is dealt with by applying a simple criterion. The type of the "conflicting" attribute *tutor* will be *employee*, which is the "intersection" (somehow in the sense of instance sharing) of the two types of the tutor attribute (*person* and *employee*). This choice is reasonably safe, and guarantees that all instances of *stud_emp* are correct instances of both *student* and *employee*.

We complete the description of inheritance recalling that there is also another built-in class in $\mathcal{O}ntoDLP$, which is the superclass of all the other classes (whether they are user defined or built-in) and is called $object$ (or $\top$). An immediate subclass of $object$ is $individual$, which is the common superclass of all base classes (see Sec. 3.2.4).

3.1.4 Base relations

A fundamental feature of a knowledge representation language is the capability to express relationships among the objects of a domain. This can be done in $\mathcal{O}ntoDLP$ by means of *(base) Relations*.

Relations are declared like classes: the keyword **relation**[5] (instead of **class**) precedes a list of attributes.

As an example, the relation *friend*, which models the friendship between two persons, and the relation *lived* containing information about the places where a person lived can be declared as follows:

$$\textbf{relation}\ friend(pers1 : person, pers2 : person).$$

$$\textbf{relation}\ lived(per : person, pla : place, period : string).$$

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

In particular, to assert that a person, say "john", lived in Rome for two years we write the following logic fact:

$$lived(per : john, pla : rome, period : "twoyears").$$

[4] We acknowledge that is quite unnatural that the tutor of a student employee is an employee. Actually we made this choice to show an important feature of the language.

[5] The keyword **relation** may be, in fact, preceded by keyword **base**; however, the latter is often omitted, for brevity.

We call this assertion a tuple of the relation *lived*. Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an object identifier).

It is worth noting that $\mathcal{O}ntoDLP$ base relations support inheritance, like base classes do.

As an example, we could define the relation *trustedFriend* as a subrelation of *friend*, as follows:

$$\textbf{relation}\ trustedFriend\ \textbf{isa}\ friend(trustLevel : integer).$$

Instances of relation *trustedFriend* are thus also seen as instances of *friend*, while the opposite is not true.

3.2 Typing system

$\mathcal{O}ntoDLP$ features a number of useful types and type constructors; in the following, we introduce them.

First, built-in classes provided are analyzed in Sec. 3.2.1. We describe naming system, together with reserver names and namespace concept, in Sec. 3.2.2.

Container classes, i.e., lists and sets, are explained in Sec. 3.2.3.

The complete type hierarchy of $\mathcal{O}ntoDLP$ is finally shown in Sec. 3.2.4.

3.2.1 Built-in classes

$\mathcal{O}ntoDLP$ provides the built-in classes depicted in Figure 3.1. Classes are organized in an isa hierarchy rooted in $object$.

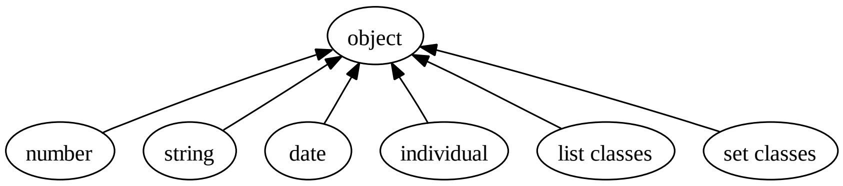

Fig. 3.1. builtin class hierarchy

The direct subclasses of $object$ are:

- $string$: strings of characters
- $date$: Gregorian dates (without time)
- $individual$: common root of all user defined base classes (3.1.1)
- $number$: numerals
- container types (lists and sets, 3.2.3)

Thus, each class instance is also an instance of $object$ (indirectly), which therefore is the most general type in $\mathcal{O}ntoDLP$. $object$ is the ”universe” of the language: any legal value is necessarily the object identifier of some class instance, whose type is a subclass of $object$. In the following, we illustrate the above classes, describing their extensions and properties.

Strings

Strings are sequences of Unicode characters enclosed in double quotes (”). Some Unicode characters cannot be used, namely the control characters:

- Ctrl-C (Unicode 0003)
- Ctrl-D (Unicode 0004)
- Ctrl-E (Unicode 0005)

Besides, there are some special (control) characters that can be used in strings, provided they are properly encoded, as follows:

- single quotes $\rightarrow$ \’
- double quotes $\rightarrow$ \”
- horizontal tabulation $\rightarrow$ *t*
- newline $\rightarrow$ *n*
- carriage return $\rightarrow$ *r*
- backslash $\rightarrow$ \ \

$\mathcal{O}ntoDLP$ number classes and their hierarchy

Numbers are organized in a class hierarchy whose root is the class $number$. Like $object$ class, $number$ does not have any proper values; its values come from its subclasses. Actual numeric classes are of the form $decimal[n, m]$, where n and m represent, the maximum number of integer (resp. decimal) digits of the elements of this data type ($n \geq 1$, $m \geq 0$). In other words, $decimal[n, m]$ is the set of numbers having **at most** n integer digits and **at most** m decimal digits. $decimal$ is used as synonymous for $decimal[9, 9]$.

Moreover, each number class has two subclasses, which form a complete partition over the number class’s domain into positive (and zero) values and negative values. In particular:

- $number$ has two direct subclasses: $positive\ number$ and $negative\ number$
- each $decimal[X, Y]$ class has two direct subclasses: $positive\ decimal[X, Y]$ and $negative\ decimal[X, Y]$;
- each $integer[X]$ class has two direct subclasses: $positive\ integer[X]$ and $negative\ integer[X]$.

Also, $positive\ decimal[n, 0]$ (resp. $negative\ decimal[n, 0]$) is aliased to $positive\ integer\ [n]$ (resp. $negative\ integer[n]$) . Decimal classes are organized in a hierarchy, where direct children of $decimal[n, m]$ are $decimal[n-1, m]$ and $decimal[n, m-1]$; the lowest classes of the hierarchy (i.e., the leaves of the isa tree) are the classes of the form $decimal[n, 0]$. Classes of the former kind collect integer numbers and can be equivalently denoted by $integer[n]$. The top decimal classes, i.e., the decimal classes which are direct subclasses of $number$, are the decimal classes of the form $decimal[n, m]$ such that $n + m = 18$, $n \leq 9$ and $m \leq 9$ (thus, the only top decimal class is $decimal[9, 9]$). Note that: $decimal[X, Y]$ is a (direct or indirect) subclass of $decimal[X1, Y1]$ if both $X \leq X1$ and $Y \leq Y1$ hold; the intersection of (the values of) any pair of classes $decimal[X, Y]$ and $decimal[X1, Y1]$ is given precisely by the set of values in $decimal[min(X, X1), min(Y, Y1)]$.

These rules combine into a number hierarchy, depicted in Figure 3.2.

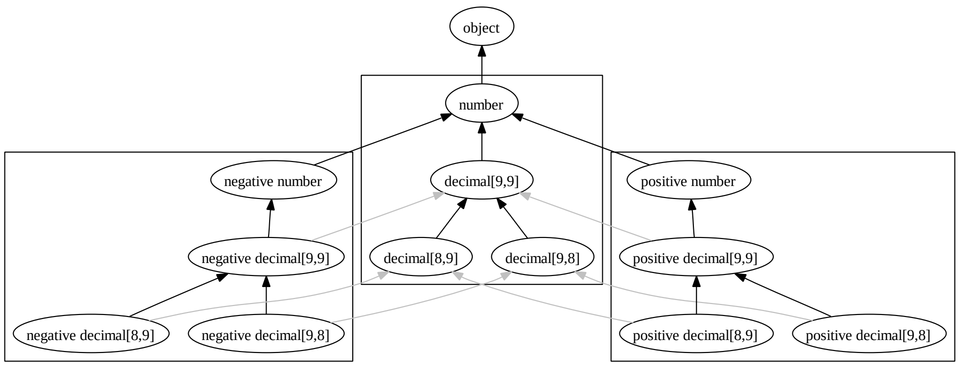

Fig. 3.2. Number classes hierarchy

It is worth noting that each number is uniquely represented in a canonical form, where superfluous zeroes in head and tail are omitted. For instance, 3.4100 is represented by 3.41 (the user is allowed to write 3.4100 for syntactic sugar, but the number is represented in its canonical form 3.41 for $\mathcal{O}ntoDLP$). Also '.' is omitted if there is no decimal digit (3.0 is represented by 3 in canonical form). The proper instances of $positive\ decimal[n, m]$ are the *positive* numbers having precisely n integer digits and m decimal digits in their canonical form. Conversely, proper instances of $negative\ decimal[m, n]$ are the *negative* numbers, having precisely n integer digits and m decimal digits in their canonical form. Zero (0) is a special case and is considered as being part of $positive\ decimal[1]$ class. Thus, each number is a proper value of precisely one class. Some examples:

- 892.23 (which is the same as 892.230, 892.2300, etc.) is a proper instance of $positive\ decimal[3, 2]$
- 28.3 is a proper member of $positive\ decimal[2, 1]$
- 0.75 is a proper member of $positive\ decimal[1, 2]$
- -0.75 is a proper member of $negative\ decimal[1, 2]$
- 3 (the same as 3.0, 3.00, 3.000, etc.) is a proper member of $positive\ decimal[1, 0]$ (equivalent to $positive\ integer[1]$)

Due to the number hierarchy, each of these values is also a (non-proper) instance of other classes. For example:

- 892.23 is an instance of $positive\ decimal[m, n]$ and of $decimal[m, n]$ (for each $m \geq 3$ and $n \geq 2$)
- 28.3 is an instance of $positive\ decimal[m, n]$ and of $decimal[m, n]$ (for each $m \geq 2$ and $n \geq 1$)
- and so on ...

In general, each numeric type can be:

- used in comparison operators inside rules;
- used with built-in operators such as $+$, $*$, $-$:

Dates

Dates are Gregorian dates, enclosed between two symbols "^". A value of the *date* class has the following format:
^$\langle year\rangle\langle month\rangle\langle day\rangle$^
For example, *^20070512^* is a valid date, whereas *^200705 ^* is not, due to missing day-of-month. Dates do not include neither a time, nor a time zone.

3.2.2 Namespaces and reserved names

A known drawback of DLP is the impossibility to use names that either do not start with a lowercase letter or contain "strange" characters (such as white spaces, slashes, etc.). $\mathcal{O}ntoDLP$ overcomes this problem by allowing "complex names", composed by any sequence of Unicode characters, enclosed in single quotes ('). For example, *'Piazza dei Bruzi'* is a valid name in $\mathcal{O}ntoDLP$ and it can be used as follows:
'Piazza dei Bruzi': place (name:"Piazza dei Bruzi").

Note that the character ', used to enclose complex names is different than ", used for strings (see "Strings" paragraph in 3.2.1). For example, *"Piazza dei Bruzi"* is a value of type , whereas *'Piazza dei Bruzi'* is the object identifier of a descendant of class $individual$. A name in $\mathcal{O}ntoDLP$ is seen as split into two components: the *namespace* and the *local name*, separated by the symbol "/". For example, the name *'people/jack'* has *jack* as local name and *people* as namespace. If multiple slashes occur in a name, only the last one (from left to right) matters as a separator. Therefore, the name *'http://www.example.org/people/jack'* has *jack* as local name and *http://www.example.org/people* as namespace[6].

When a name has no slashes, it is considered as being declared in the default namespace; so, *'jack'* is an $\mathcal{O}ntoDLP$ name whose local name is *jack* and whose namespace is equal to default one. Additionally, single quotes around a name in the default namespace may also be omitted, provided that it is regular enough to avoid confusion with other language elements, such as variables names and keywords. In

[6] Note that the separator does not belong to either the namespace or the local name.

particular, a ”regular” name starts with a lowercase letter and does not contain any special characters (such as spaces or slashes); thus, e.g., regular name *myName* can be written also as *’myName’*, whereas in *’my spaced name’* single quotes could not be omitted. The default namespace can be changed using the following declaration:

#namespace *’<new default namespace>’*

For example:

#namespace *’http://www.example.org/ontology/core’*

changes the default namespace to *http://www.example.org/ontology/core* for each simple name used in the ontology. When an explicit namespace declaration is missing, the default namespace is the empty string; so, for example, in absence of a namespace declaration, *’simpleName’*, *’/simpleName’* [7] and *simpleName* are quite equivalent. Even when a default namespace declaration is present, names can belong to different namespaces, using a prefix (*alias*). An alias can be declared as follows:

#alias *prefix = ’namespaceDefinition’*

Valid declarations of aliases are the following:

#alias *core = ’http://www.example.org/ontology/core’*

#alias *living_being = ’http://www.example.org/ontology/domain of the living being’*

Aliases can be used to compose names, prefixing a local name with an alias name and the symbol ”::”. For example:

’living_being::person’

is fully equivalent to:

’http://www.example.org/ontology/domain of the living being/person’

[7] *’/simpleName’* is obtained by concatenating a namespace equal to the empty string and a local name equal to *simpleName*; remember that a slash (/) is always required to separate namespace from local name.

though the former is dependent on the declaration of alias *living_being*, while the latter is not dependent on anything. In this sense, the former syntax for names is called *"relative"*, while the latter is called *"absolute"*. It should be noted that, in order to avoid ambiguities, a local name cannot contain the symbols "::" and "/", while a namespace cannot contain the symbol "::". Some control characters cannot be used in names, namely:

- Ctrl-C (Unicode 0003)
- Ctrl-D (Unicode 0004)
- Ctrl-E (Unicode 0005)

Besides, some special (control) characters can be used in names, provided they are properly encoded, as follows:

- single quotes $\rightarrow$ \'
- double quotes $\rightarrow$ \"
- horizontal tabulation $\rightarrow$ $\backslash t$
- newline $\rightarrow$ $\backslash n$
- carriage return $\rightarrow$ $\backslash r$
- backslash $\rightarrow$ $\backslash\backslash$

It is useful to remember that some keywords (such as built-in classes and relations names) belong to *any* conceivable namespace, including the empty one. Thus, they are reserved everywhere and cannot be used outside its original meaning. Reserved names include:

- the names of all classes that are built-in, namely:
  $object$, $individual$
  $decimal$, $'positive\ decimal'$, $'negative\ decimal'$
  $integer$, $'positive\ integer'$, $'negative\ integer'$
  $number$, $'negative\ number'$, $'positive\ number'$
  $string$, $date$
- the names of meta-classes and relations [8], namely:
  $'base\ class'$, $'base\ relation'$, $'collection\ class'$, $'intensional\ relation'$
  $class$, $relation$
  isa , $isaClosure$
  $hasAttribute$

3.2.3 Container classes: lists and sets

In this section we illustrate *container classes*, i.e. the $\mathcal{O}ntoDLP$ constructs allowing the representation of groups of objects. In particular, we show how the definition of classes of type *list* and *set* allows to build terms at an arbitrary nesting level.

[8] Meta classes and relations are explained in detail in chapter 4

Lists as containers

In general, a *list* is an *ordered* group of values that accepts multiple copies of the same value. For each class C, in $\mathcal{O}ntoDLP$ there is an implicitly defined class "list of C" (referred as $[C]$). The instances of this class are all the possible lists having instances of the class *C* as elements. The implicit definition of the list classes allows declaring attributes having a list of objects as type. For example, we can define a class *person*, adding the attribute *sons* with type list of *person*, as follows:

class *person (name: string, age: integer, birthplace: place, sons : [person]).*

Lists can appear inside the predicates of a rule as terms (see also Sec. 3.3.1). For example, the predicate *firstBorn* associating each person to his first-born son can be defined as [9]:

firstBorn(X,Y) :- X: person(sons: [Y|R]).

In this case sons list is partitioned in two parts: *Y* represents the first element of the list (denoted as *list head*) and *R*, the sublist obtained eliminating the first element from the complete list (denoted as *list tail*)[10] .

Using built-in functions, lists can be manipulated in an efficient way (avoiding the use of recursion for enumerating list elements). The implicit definition of class $[object]$ allows the representation of list anyhow complex (nested lists and/or lists containing heterogeneous elements).

Sets as containers

In a similar way, $\mathcal{O}ntoDLP$ allows to define *sets*, which are collection of objects whose order does not matter; unlike *lists*, *sets* do not accept multiple copies of a given value.

Thus, for each class C, in $\mathcal{O}ntoDLP$ there exists a class "set of C" (referred as $\{C\}$). The instances of this class are all the possible sets having instances of the class *C* as elements.

Sets can be used as type of an attribute, like in:

class *person (name: string, age: integer, birthplace: place, cars :* $\{car\}$*).*

[9] In this example, we are supposing that the sons list is ordered by age.
[10] Since we do not care about what the tail is, we could also use substitute *R* for _, thus writing:

firstBorn(X,Y) :- X: person(sons: [Y|_]).

Sets can also be used inside the predicates of a rule as terms. For example, the predicate *twoCars* identifying people having two cars can be defined as:

twoCars(X) :- X: person(cars: {Car1, Car2}).

3.2.4 The complete hierarchy

We are now able to look at the complete type hierarchy of $\mathcal{O}ntoDLP$. The common superclass, called $object$, has six different built-in subclasses, namely:

- *individual*, common superclass of all base classes
- *date*, a non-derivable built-in class, for Gregorian dates
- *number*, the root of number classes hierarchy
- *string*, another non-derivable built-in class, representing strings
- *[object]*, the generic "list of objects" (and superclass of all list classes)
- $\{object\}$, the generic "set of objects" (and superclass of all set classes)

Moreover, all the list classes are part of the $\mathcal{O}ntoDLP$ inheritance hierarchy; basically, the class $[a]$ is a subclass of class $[b]$ if class a is a subclass of class b. For example, the class $[student]$ is subclass of $[person]$ if *student* is subclass of *person*.

This rule also applies to $object$: since every class is a subclass of $object$, given a class a it holds certainly that $[a]$ is a subclass of $[object]$. Moreover, $object$ is a special class with respect to inheritance. In fact, inheritance relations involving container classes always relate container classes at the same nesting depth, with only two exceptions, regarding $object$ class:

- a list of objects is an object ($[object]\ \ isa\ \ object$)
- a set of objects is an object ($\{object\}\ \ isa\ \ object$)

A property of container classes spring from these rules; since $object$ is superclass of all classes, $[object]$ is superclass of $[[...everyotherclass...]]$, and, in particular, of $[[object]]$. Thus, the hierarchy relation between $object$ and $[object]$ generates an *infinite* hierarchy of list classes having increasing depths. Of course, this also applies to $\{object\}$, e.g., $object$ is superclass of $object$, etc.

Combining both hierarchy dependencies, we have that $[object]$ is also superclass of $[\{object\}]$, e.g., "a list of sets of objects is also a list of objects", since a set of objects is an object in itself. Similarly, we can say that $\{object\}$ is also superclass of $\{[object]\}$, e.g., "a sets of lists of objects is also a set of objects", since a list of objects is an object in itself.

The class hierarchy springing from these rules is, in fact, unlimited; a scheme that depicts how it could be generated is in Figure 3.3.

While an $\mathcal{O}ntoDLP$ ontology can use classes in any depth level in the hierarchy, a given ontology uses only a limited subset of the classes, thus defining a finite "view" of the unlimited hierarchy.

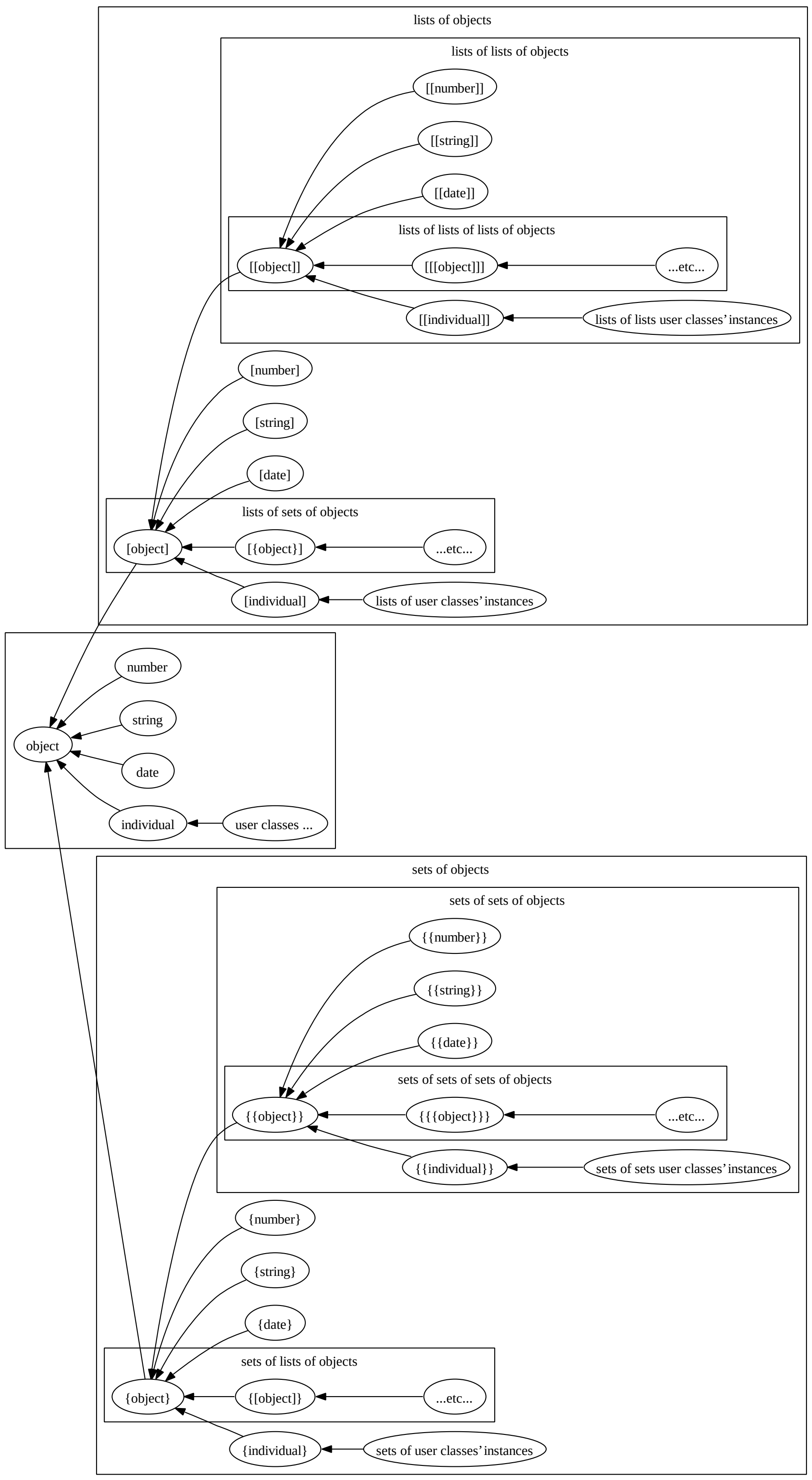

Fig. 3.3. $\mathcal{O}ntoDLP$ class hierarchy, with containers

An example

As an example, let's consider a small ontology with four classes *a*, *b*, *c*, and *d*, depicted in Fig. 3.4.

class a.
class $b\ isa\ \{a\}$.
class $c\ isa\ \{a\}$.
class $d\ isa\ \{b,\ c\}$.

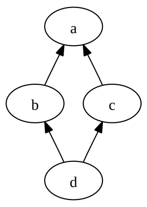

Fig. 3.4. A simple class hierarchy

From these declarations, the class hierarchy scheme in Fig. 3.4 is instantiated into a class hierarchy whose first levels are shown in Fig. 3.5.

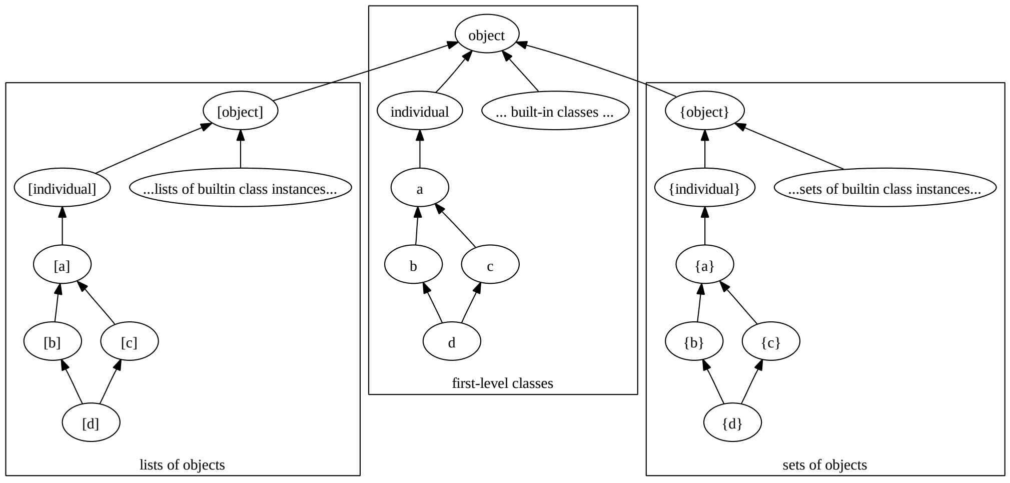

Fig. 3.5. The class hierarchy $\mathcal{O}ntoDLP$ "sees", an extension of the previous one

3.3 Intensional ontology entities

In addition to base classes, base relations and their instances, $\mathcal{O}ntoDLP$ offers to the user the possibility to define knowledge by means of other constructs that exploit the inferential features of the language. Logical rules are, in fact, the basis which these constructs are built on. In particular, $\mathcal{O}ntoDLP$ modeling constructs defined by means of logical rules include:

- *intensional relations* : relations whose instances can be declared explicitly as well as inferred through rules (as opposed to base relations, whose instances have to be enumerated);
- *collection classes* : a mean to collect object identifiers defined elsewhere and assign them a set of attributes, either via explicit declaration of values or through rules.

In the next sections we illustrate the above language constructs. In Sec. 3.3.1, we present the basic building blocks of these constructs: *logical terms*, *arguments*, *literals* and *rules*. Then, we show how base class instances (see 3.1) can be reclassified by using *collection classes*. Similarly, we illustrate how *intensional relations* can be defined by inferring tuples by means of logical rules.

Other constructs based on logical rules are described in Sec. 3.4.

3.3.1 Building blocks: terms, arguments, literals

Since $\mathcal{O}ntoDLP$ extends the Disjunctive Logic Programming, it inherits the capability to express *logical rules*, built on *literals*; the latter use *logical terms*.

Logical terms

Logical terms are the ”atoms” of the inferential process; in $\mathcal{O}ntoDLP$, a *term* can be:

- a *variable*, either having a ”regular” name starting with an uppercase letter (such as *X*, *Zed*, etc.), or an anonymous one expressed as _ (whose meaning is ”do not care”);
- a *constant value* of one of the aforementioned classes (namely, a $decimal$ value, a $date$, a $string$, a *name* or a *list/set* of values, possibly nested);
- a *complex term* whose values are later determined to be the object identifiers of entities satisfying some properties (this allows navigating ontology right from terms);
- a *list enumeration term*, composed of other terms grouped in a *list* (such as *[First, Second, Third]*);
- a *set enumeration term*, composed of other terms grouped in a *set* (such as $\{$*OneMember, ASecondMember, AThirdMember*$\}$);
- a *head/tail list term*, used to decompose lists into first element and the next ones (such as *[Head*$|$*Tail]*).

Variables and constant values are simple terms, in that they are not built on top of other terms, whereas *complex terms*, *list enumeration terms* and *head/tail list terms* can use general terms in their syntax. Terms can be combined at arbitrarily deep nesting levels, resulting in complex expressions, such as in:

- *employee(worksIn: company(foundedInYear: 2002))* all employees that work in companies whose founding year is *2002* [complex on complex on constant value];
- *[[1,2], [3,4]]* a list whose elements are two lists of numbers (*[1,2]* and *[3,4]*) [list on list on constant values];
- *[X, Y, Z]* a list whose elements are variables, may turn out to represent any three-element list depending on the actual values of *X, Y, Z*[11] [list on variables];
- {*X, Y, Z*} a set whose elements are variables, may turn out to represent any three-element set depending on the actual values of *X, Y, Z* [set on variables];
- *[company(foundedInYear: 2002), company(foundedInYear: 2006)]* a list of two elements, where the former is an object identifier of companies founded in 2002 and the latter is oid of companies founded in 2006; this may turn out to represent many lists, depending on the number of combinations of companies having these properties [list on complex on constant value].

Arguments

An argument is the occurrence of a term linked to some schema. It may specify a name (such as $age : X$) or not (e.g., just 42). While unnamed arguments are exactly the kind of arguments supported in plain DLP, the former are an extra feature of $\mathcal{O}ntoDLP$, which allows pointing out an attribute on the basis of its name, rather than on its positioning among the others. For that reason, lists of arguments having a name are said to be in *non-positional notation*, whereas if no one of the arguments in the list has a name before it, the list itself is said to be in *positional notation*. Note that a list of argument cannot mix both named and unnamed arguments; it would be considered as a syntax error.

Literals

Literals are the building blocks of many reasoning constructs (including rules); they are heavily based on terms and arguments. Supported literals are of the following kinds:

- *simple literals*, referring to knowledge contained into the ontology
- *aggregate literals*, expressing ways to aggregate knowledge using accumulation operators
- *built-in literals*, expressing knowledge about some built-in relations (such as value comparisons or arithmetics)

A list of literals separated by commas is a *conjunction*; a list of simple literals separate by $\vee$ symbol is a *disjunction*.

[11] Note that some or all of *X, Y, Z* may be lists or sets themselves; thus, the proposed term would match e.g., *[[1, 2], 3, a]*.

Simple literals

Simple literals refer to knowledge contained in the ontology; the general syntax is

$$[\textit{oid:}]\ \textit{name}\ (arg_1, ..., arg_n)$$

where:

- *oid* is a term whose value will be (if it is a variable) or dictates (if it is a constant term), the object identifier of entity referred to (valid only if it is a base class or a collection class)
- *name* refers to an entity defined in the ontology (such as a *base class*, a *base relation*, a *collection class* or an *intensional relation)* or an auxiliary predicate
- arg_1, ..., arg_n are arguments referring to attributes of the entity

Moreover, *name* can be also the name of a built-in class, such as *decimal*, *integer*, *date*, *string*, etc.; this *special* kind of simple literal can be used to test for membership of some values to a given class. For example,

X: string()

can be used to test that X belongs to class $string$[12], but cannot give values to X (in particular, it cannot be used to enumerate all strings in Herbrand's base). When the class name is *individual*, though, the simple literal allows also enumerating: e.g.,

Someone: individual()

enumerates through all base class instances.

Aggregate literals

Values of some of types can also be "aggregated" with operators like #`count`, #`min`, #`max`, #`sum`[13].

Table 3.1 reports which aggregate operator can be used, on the basis of the type of the aggregated variables type; each combination of type and aggregation operator is marked with a symbol:

- **NA**: Not Applicable, this combination does not make sense
- **S**: Supported, this combination is feasible
- **NS**: Not supported, this combination, while feasible, would lead to highly inefficient computation and is thus unsupported (forbidden) at system level

In particular:

[12] A literal of kind would falsify the body of a rule in which *X: string()* appears, in each ground instantiation where X is not a $string$.

[13] For more information on DLP with aggregates (DLP), please refer to Appendix A

	#min	#max	#sum	#count
string	**NA**	**NA**	**NA**	**S**
date	**S**	**S**	**NA**	**S**
integer	**NS**	**NS**	**S**	**S**
positive integer	**S**	**S**	**S**	**S**
negative integer	**S**	**S**	**S**	**S**
decimal	**NS**	**NS**	**S**	**S**
positive decimal	**S**	**S**	**S**	**S**
negative decimal	**S**	**S**	**S**	**S**
individual and its descendants	**NA**	**NA**	**NA**	**S**

Table 3.1. Supported aggregates with respect to aggregation variable type.

- #count is applicable and supported on any type (one can always count ”how many” exist of a given thing, this does not imply computing an ”accumulation” of data, but only a count of elements);
- #sum is applicable and is supported on any numeric type (i.e., any built-in type except for *string* and *date*);
- sign-homogeneous integer numeric types (i.e., *positive integer* and *negative integer*) support both #min and #max;
- also sign-homogeneous non-integer numeric types (i.e., *positive decimal* and *negative decimal*) support both #min and #max;
- numeric types with mixed sign could support #min and #max, but this would lead to an overly complicated rewritten DLP program, with bad performances, so those combinations are recognized at system level as invalid.

Built-in literals

Built-in literals refer to relations that are built-in in the system, such as ordering between numerals, or arithmetic. The *name* of the built-in literal is the name of the relation queried for. In particular, the following built-in relations are available:

- arithmetic operators: $+$, $*$
- equality check: $=$ (also $==$)
- comparison: $>$, $>=$, $<$, $<=$, $<>$ (also $!=$)

Moreover, a number of built-in exist for manipulating lists and sets, e.g., getting or checking the cardinality of a list, getting first or last element, etc.

For example, the following rule computes in the predicate *numberOfChildren*, the number of sons of a person:

numberOfChildren (P,X) :- P: person(children: L), #length(L, X).

Rule

A rule is a conjunction of literals that implies a disjunction of simple literals. An example of logical rule in $\mathcal{O}ntoDLP$ is the following:

male(X) $\vee$ *female(X) :- X:person().*

The disjunction *male(X)* $\vee$ *female(X)* is the **rule head** and it is followed by the conjunction *X:person()*, also said the **rule body**. The meaning of this rule is:
a person can be male or female
Another example of rule is the following:

bornInUSA(X) :- X:person(birthplace:B), locatedIn(B, "USA").

This rules states that a person is born in USA if its birthplace is located in a country named "USA". In this case the rule is not disjunctive, since the head is given by only one literal.
Note that in a rule body literals can appears also negated. For example, the rule

*unreachableByTrain(X) :- X: place(),**not** Y: 'RailwayStation'(placedIn:X).*

is used to state that a place is unreachable by train if it does not have a train station.

3.3.2 Instances reclassification: collection classes

In $\mathcal{O}ntoDLP$ one can define a special kind of classes, named *collection classes*, which allow to represent collections of heterogeneous objects belonging to different classes.

Indeed, collection classes are rather different from base classes, mainly for the way their instances are defined.

Instances of base classes have to be explicitly defined and they cannot be instances of other classes at the same time, unless those classes are super-classes of the class in which these instances have born.

On the other hand, collection classes instances are given through the union of subsets of instances of pre-existing classes, i.e., they spring from a re-classification of instances already defined in other classes (both user defined classes and built-in classes such as integer, string, etc.), and they can be at the same time instances of other unrelated collection classes.

In particular, while a class instance's identifier must be defined from scratch and attributes of a base class instance have to be defined explicitly, a collection class instance's identifier has to be an existing object identifier already used to identify a class instance, and, together with the collection class instance's attributes, has to be determined using suitable logic rules.

Declaring a collection class may be useful whenever it is required to refine, using some criterion, a classification of existing instances.

For example, suppose that the class *musicalInstrument* and the relation *plays* that links an instance of *person* to an instance of *musicalInstrument* have been defined as follows:

class *musicalInstrument (name: string).*
relation *plays(pers: person, inst: musicalInstrument).*

One can afterward define the collection class *musician* as follows:

```
collection class musician (name:string)
{
X: musician(name:N) :- plays(X,_), X:person(name:N).
}
```

This way one specifies how to compute instances of collection class *musician* through a simple logic rule, which translates the basic notion that a musician is a person that plays a musical instrument.

Declaring a collection class thus needs the definition of a schema (like with, e.g., base classes) and the specification of a set of logic rules. These rules determine how instances and values of their attributes are computed.

In the above-mentioned example, instances of collection class *musician* are a subset of instances of *person* class. Classes which instances are taken from are named "generating classes"; so, *person* is a generating class for *musician*. Moreover, the value of each attribute present in the schema of the collection class is computed through the given inference rules, like happens, in the example, for the musician's name.

Now consider to redefine the collection class of the above example as follows:

```
collection class musician (name:string, instrument:musicalInstrument)
{
X: musician(name:N, instrument:Y) :- plays(X,Y), X:person(name:N).
}
```

Noting that the relation *plays* of the example is *"n:m"* (i.e., a person might play more than one instrument or none, a musical instrument might be played by many people, or none), we can have that two different set of attributes values are computed for a given object identifier (e.g., *jos: musician(name: "josh", violin)* and *jos: musician(name: "josh", viola)*).

In this case the collection class itself is considered not valid.

Let's examine another example, in order to show how one can compute the value of an attribute using aggregates.

We define collection class *busyPerson* as (informally) the set of people that have at least two jobs.

```
collection class busyPerson(jobCount: integer)
{
X: busyPerson (jobCount: Z) :- X:person(), #count{Y:hasJob(X,Y) }=Z, Z>=2.
}
```

Note that attribute *jobCount* is computed using aggregate `#count` in the instances definition rule.

It is important to note that rules in a collection class definition have to be normal stratified rules, that they may define and use auxiliary predicates[14] and that they admit, in rules heads, a predicate whose name is equal to the collection class being defined. Constraint (i.e., , rules without head, whose body must always be false) are not allowed[15].

A special case of collection class definition is the enumeration of its instances, through body-less rules, named *facts*.

```
collection class threeStringedInstruments()
{
violin: threeStringedInstruments ().
viola: threeStringedInstruments ().
cello: threeStringedInstruments ().
}
```

The same principle allows one to use built-in classes as generating classes in facts.

For example:

```
collection class firstThreeIntegers()
{
1: firstThreeIntegers ().
2: firstThreeIntegers ().
3: firstThreeIntegers ().
}
```

and:

[14] The scope of auxiliary predicates is limited to definition of collection class
[15] If consistency conditions for a collection class are needed, one can use axioms, see 3.4.1.

collection class *weekday()*
{
"monday": weekday ().
"tuesday": weekday ().
"wednesday": weekday ().
"thursday": weekday ().
"friday": weekday ().
"saturday": weekday ().
"sunday": weekday ().
}

While in the examples seen to this point, each collection class has an unique generating class, this is not necessarily true.

For example, given the class:

class *voice(kind: string).*

which models the classification of voices in opera (tenor, soprano, etc.), one can define the collection class:

collection class *orchestraElement()*
{
X: orchestraElement () :- X: musicalInstrument ().
X: orchestraElement () :- X:voice().
}

Which specify the set of elements of an orchestra, encompassing both musical instruments and voices. Collection classes may be organized into taxonomies (like classes and relations), using isa relation.

For example:

collection class *veryBusyPerson* ***isa*** $\{busyPerson\}$ ()
{
X: veryBusyPerson (jobCount: Z) :- X:person(), #count{*Y:hasJob(X,Y)* }*=Z, Z>=10.*
}

Inheritance semantics for *collection classes* is slightly different than the one adopted for classes. In fact, two base classes may share instances only if there exists an inheritance path between them (i.e., one is subclass, direct or indirect, of the other), while a pair of collection classes may share instances no matter they are linked by means of inheritance or not. However, like for classes, if B is a sub-collection class of A, all instances of B are also instances of A.

Anyway, please note that hierarchies of collection classes and hierarchy of classes must remain disjoint (a class and a collection class cannot be involved in an inheritance relation).

In general:

- different hierarchies of collection classes (trees of them, linked by isa) are allowed;
- hierarchies of collection classes are disjoint w.r.t. classes;
- collection classes do not have a common super-collection classes, while classes have the class $individual$ as common super-class;
- collection classes can be used to define type of attributes[16].

3.3.3 Intensional relations

Defining a relation implies, like for classes, defining the schema and its instances (tuples).

However, often it could be more comfortable to define instances ”intensionally”, i.e., using some logic rules to specify how to compute rather than enumerating them. This can be accomplished in $\mathcal{O}ntoDLP$ by using an *intensional relation*. An example of intensional relation is *sonInLaw*, defined as follows:

intensional relation *sonInLaw* (*son* : *person, father* : *person*)
{
sonInLaw(X, Y) :- X: person(partner: Z), Z: person(father: Y).
}

Here, the definition of the tuples of the intensional relation *sonInLaw* is accomplished by a rule asserting that, if Y is the father of X's partner, then X is son-in-law of Y.

Another example of intensional relation is:

intensional relation *cousin(p1: person, p2: person)*
{
cousin(*X,Y*)*:- X* : *person*(*father* : *V*)*, V* : *person*(*father* : *Z*)*,*
Y : *person*(*father* : *W*)*, W* : *person*(*father* : *Z*)*, X != Y, V !=W.*
}

The above-mentioned intensional relations *sonInLaw* and *cousin* are similar to well-known ”views” in relation databases; however, it is important to note that $\mathcal{O}ntoDLP$ interpretation of view concept is more powerful than relation databases one, since, for example, $\mathcal{O}ntoDLP$ allows for recursion in rules.

As an example of recursive intensional relation, let's consider the classical *ancestor* example:

[16] In this case, checking for type correctness implies computing extension of a collection class, i.e., instances.

intensional relation *ancestor(desc: person, anc: person)*
{
ancestor(A,X):-A : person(father : X).
ancestor(A,X):-A : person(father : Y), ancestor(Y,X).
}

3.4 Axioms, queries and reasoning modules

Apart from defining intensional entities, literals and logical rules may be used in other very important $\mathcal{O}ntoDLP$ constructs:

- *axioms*, a mean to assess properties that must hold in a consistent view of the domain;
- *reasoning modules*: the language components endowing $\mathcal{O}ntoDLP$ with powerful reasoning capabilities coming from DLP.
- *queries*, which give the possibility of extracting knowledge contained in the ontology;

In Sec. 3.4.1 we illustrate *axioms*, whereas reasoning modules are described in Sec. 3.4.2. Usage of queries in $\mathcal{O}ntoDLP$ is depicted in Sec. 3.4.3.

3.4.1 Axioms and Consistency

We said before that the structural representation of a knowledge domain is obtained in $\mathcal{O}ntoDLP$ by specifying classes and relations; however, in general, this information is not enough to obtain a correct description of the domain. Often, we need to impose constraints asserting additional conditions that hold in the domain.

These assertions are modeled in $\mathcal{O}ntoDLP$ by means of *axioms*.

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

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

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

It is clear that the information about the company of an employee (recall that there is an attribute company in the scheme of the class *employee*, see Sec. 3.1.3) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we state the following axiom:

$$(1) : X2 : employee(company : C) \ {::-} \ colleague(emp1 : X1, emp2 : X2), \\ X1 : employee(company : C)$$

The above axiom states that, if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

Note that $\mathcal{O}ntoDLP$ axioms are different from logic rules, they do not derive new knowledge, but they are only used to model sentences that must be always true, like integrity constraints. Importantly, axioms are distinguished by rules because they are declared by using the symbol $::-$ instead of $:-$. Moreover, $\mathcal{O}ntoDLP$ also supports a syntax which is very close to the one used in logic programming for constraints. In the previous example, we could equivalently write:

$$\begin{aligned}(2):\ & ::-\ colleague(emp1:X1, emp2:X2), X1:employee(company:C),\\ & \text{not} X2:employee(company:C)\end{aligned}$$

Note that it is always possible to write an axiom as a constraint.

3.4.2 Reasoning modules

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

Reasoning modules are the language components endowing $\mathcal{O}ntoDLP$ with powerful reasoning capabilities coming from DLP. Basically, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules in $\mathcal{O}ntoDLP$ are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem in a unique definition identified by a name). Moreover, it is possible to define *derived* predicates having a “local scope” without giving a scheme definition. This gives the possibility to exploit a form of modular programming, because it becomes possible to organize logic programs in a simple kind of library.

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

$$\begin{aligned}&\textbf{module}\ (shyFriends)\{\\ &\quad youngAndShy(N) :- P:person(name:N, age:A), A < 18,\\ &\qquad\qquad \#count\{F:friend(pers1:P, pers2:F)\} < 10.\\ &\}\end{aligned}$$

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

We now show another example demonstrating that the reasoning power of $\mathcal{O}ntoDLP$ can be exploited also for solving complex real-world problems.

Given our living being ontology, we want to compute a project team satisfying the following restrictions (i.e., we want to solve an instance of *team building problem*):

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

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

$$\begin{array}{l}\textbf{class}\ project(numEmp : integer,\\ \quad numSk : integer,\\ \quad budget : integer,\\ \quad maxSal : integer).\end{array}$$

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

```
module (teamBuilding){
    /*r*/   $inTeam(E,P) \vee outTeam(E,P)$ :- $E : employee()$,
                                            $P : project()$.

    /*$c_1$*/  :- $P : project(numEmp : N)$,
             not#count{$E : inTeam(E,P)$} $= N$.

    /*$c_2$*/  :- $P : project(numSk : S)$,
             not#count{$Sk :\ E : employee(skill : Sk)$,
             $inTeam(E,P)$} $\geq S$.

    /*$c_3$*/  :- $P : project(budget : B)$,
             not#sum{$Sa, E :\ E : employee(salary : Sa)$,
             $inTeam(E,P)$} $\leq B$.

    /*$c_4$*/  :- $P : project(maxSal : M)$,
             not#max{$Sa :\ E : employee(salary : Sa)$,
             $inTeam(E,P)$} $\leq M$.
}
```

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

Concluding, reasoning modules isolate a set of logic rules and constraints conceptually related, they exploit the expressive power of disjunctive logic programming allowing to perform complex reasoning tasks on the information encoded in an ontology.

3.4.3 Querying

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

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

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

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

For instance, consider the reasoning module *shyFriends* defined in the previous section, we can ask, using queries:

- whether the number of people who are "young and shy" and were born in Rome is less than ten

$$\#count\{X : youngAndShy(X), X : person(birthplace : rome)\} < 10?$$

- whether there is a person whose name is "Jack" and is "young and shy":

$$youngAndShy(person(name : "Jack"))?$$

4

Meta reasoning

$\mathcal{O}ntoDLP$ offers powerful meta-reasoning features, giving the possibility to access the information contained in the ontology schemas. In other words, $\mathcal{O}ntoDLP$ allows one for the extraction of knowledge contained in the ontology structure itself; this meta-information can be obtained through some special classes and relations, defined by the system and usable for querying, as well as for any other reasoning purpose, even in combination with other literals.

In Sec. 4.1 we depict the *meta hierarchy*, showing some of its features; then, in Sec. 4.2, we present some usage examples.

4.1 The meta hierarchy

The definition of built-in classes and relations, which are necessary to perform meta-reasoning, starts from the enrichment of the $\mathcal{O}ntoDLP$ type hierarchy by means of the type $meta$, representing all the entities appearing in the ontology.

Recall that $\mathcal{O}ntoDLP$ classes include:

- *base classes*, i.e., subclasses of $individual$ (see Sec. 3.1.1);
- *builtin classes*, like e.g., $date$, $string$ and $number$ (see Sec. 3.2.1);
- *container classes*, i.e., lists and sets (see Sec. 3.2.3);
- *collection classes* (see Sec. 3.3.2).

Relations include:

- *base relations* (see Sec. 3.1.4);
- *intensional relations* (see Sec. 3.3.3).

$\mathcal{O}ntoDLP$ mimics this structure of entities in a parallel hierarchy of "special" classes, rooted at $meta$. *Meta-classes* are shown in Fig. 4.1 and organized as follows:[1]:

[1] Please note that most of the syntax used herein would not be acceptable in a regular, user-defined $\mathcal{O}ntoDLP$ ontology; meta classes are built automatically from the system.

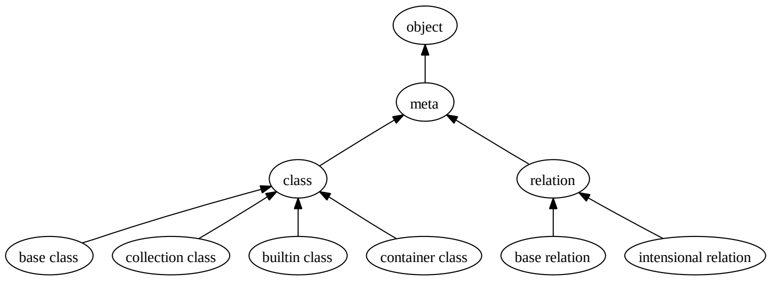

Fig. 4.1. $\mathcal{O}ntoDLP$ meta class hierarchy

- ***class 'base class'***(), has an instance *c: 'base class'*() if *c* is the name of a base class existing in the ontology. It also holds that *individual: 'base class'()*, since *individual* is the ancestor of all user-defined base classes;
- ***class 'collection class'***(), has an instance *cc: 'collection class'*() if *cc* is the name of a collection class existing in the ontology;
- ***class 'container class'***(), has an instance *cc: 'container class'*() if *cc* is the name (like, e.g., *[integer]*, *[myclass]*, {*individual*}, {*[object]*}, etc.) of a container class existing[2] in the ontology;
- ***class 'builtin class'***(), has an instance *bc: 'builtin class'*() if *bc* is the name of a builtin class;
- ***class 'base relation'***(), has an instance *r: 'base relation'*() if a relation named *r* exists in the ontology;
- ***class 'intensional relation'***(), has an instance *ir: 'intensional relation'*() if an intensional relation named *ir* exists in the ontology;

Applying inheritance, we have that *class* has an instance *c: class()* whenever *c* is the name of a *base class*, a *builtin class*, a *collection class* or a *container class*. Moreover, since $object$ and any of its descendant are all classes, the meta-class named *class* has also an instance for each of the classes defined in $\mathcal{O}ntoDLP$ and not included in the above listing. In particular, instances of meta-class *class* are also:

- $object$, i.e., it holds that *object: class()*
- the name of all meta-classes themselves, i.e., it holds that:
  meta: class().
  class: class().
  'base class': class().

Note also that *'meta'* is equivalent to *meta*, while for writing names having spaces single quotes are needed. So, *'intensional relation'* is not equivalent to *intensional relation* (without quotes); occurrence of the latter, also in an otherwise legal user program, would be a syntax error. More on name syntax can be found in Sec. 3.2.2.

[2] Existence, in this context, means that the given container class appears somewhere in the ontology; since a container class may be used *only* when declaring attributes, an "existing" container class *must* be the type of an attribute declared in some entity (see Sec. 3.2.4).

'collection class': class().
'builtin class': class().
'container class': class().
relation: class().[3]
'base relation': class().
'intensional relation': class().

While the double occurrence of *class* in *class: class()* may seem ambiguous at first glance, it is not the case. In fact, when a name like *class* it is used both in contexts where values (i.e., instances of classes) are accepted, such as before the colon ':', and in contexts where types (i.e., name of classes) are expected, such as between the colon ':' and the first open parenthesis '(', the context *matters* and it drives proper recognition of the meaning for a given name.

In a similar way, we have that instances of *'base relation'* and *'intensional relation'* both contribute, via projection, to their common superclass, *relation*. In other words, *relation* has an instance *r: relation()* if *r* is the name of a *base relation* or an *intensional relation*.

Both instances of meta-class *class* and instances of meta-class *relation* are collected in their common super-class, *meta*.

Meta-classes are also involved in some system-defined relations, called *meta-relations*. They are:

- ***relation isa***(*subentity*: $meta$[4], *superentity*: $meta$) has a tuple *isa*(*subentity*: *e1, superentity:e2*) if *e1* is a *class* or a *relation* that is a direct specialization [5] of *e2* that is a *class* or a *relation*, respectively;
- ***intensional relation isaClosure*** (*subentity*: *meta*, *superentity*: *meta*) has a tuple *isaClosure* (*subentity*: *e1*, *superentity* : *e2*) if *e1* is a *class* or a *relation* that is a specialization of *e2* that is a *class* or a *relation*, respectively;
- ***relation hasAttribute***(*entity*: *meta*, *name*: *string*, *position*: *integer*, *type*: *class*) has a tuple *hasAttribute*(*entity*: *e, name*: "*attrName*", *attributePosition*: *pos*, *type*: *attrType*) for each attribute of *e*[6] in position *pos* (starting from 1), having name *attrName* and type *attrType*.

Note that $isaClosure$ can be thought of as:

[3] Please pay attention to this; we are not saying that a relation is a class (which would be absurd), instead that *relation* is the object identifier of an instance of meta-class *class*.
[4] In the isa meta-relation, $meta$ individuals are projected only on the user defined classes.
[5] *e1* is a direct specialization of *e2* if it does not exist an entity *e3* such that *e3* is a direct specialization of *e2* and *e1* is a direct specialization of *e3*
[6] Note that, though attribute *entity* is declared as being of type *meta*, it will take values only among instances of *meta-classes* corresponding to entities, i.e., *base classes*, *base relations*, *collection classes*, *intensional relations*

intensional relation *isaClosure(subentity: meta, superentity: meta)*
{
isaClosure(Sub, Super) :- isa(Sub, Super).
isaClosure(Sub, Super) :- isa(Sub, IntermediateSuper), isaClosure(IntermediateSuper, Super).
}

4.2 Using meta classes and meta relations

In this section we depict what can be done by using meta classes and meta relations.

Example 4.1. If we have an example ontology:

> ***class****'living being' (name: string).*
> ***class****person****isa***{*'living being'*} *(age: 'positive integer').*

Then system builds a corresponding meta-knowledge base, that has a fixed part (which does not change with respect to the actual ontology), composed of:

- the definition of meta classes (the *meta hierarchy*);
- instances of meta classes;
- the definition of meta relations.

Another portion of the meta-knowledge is driven by user declarations, though, and in this example it will contain:

> *'living being':* ***class*** *().*
> *person:* ***class*** *().*
> ***isa****(sub: 'person', super: 'living being').*
> ***hasAttribute****(e: 'living being', name: "name", pos: 1, type: string).*
> ***hasAttribute****(e: person, name: "name", pos: 1, type: string).*
> ***hasAttribute****(e: person, name: "age", pos: 2, type: 'positive integer').*

Example 4.2. If we change the previous ontology as:

> ***class*** *'living being' (name: string).*
> ***class*** *person* ***isa*** {*'living being'*} *(age: 'positive integer', friends: [person]).*

Then, the meta-knowledge available to $\mathcal{O}ntoDLP$ will be:

'living being': ***class*** *().*
person: ***class*** *().*
isa*(sub: person, super: 'living being').*
'[person]': 'container class'().
hasAttribute*(e: 'living being', name: "name", pos: 1, type: string).*
hasAttribute*(e: person, name: "name", pos: 1, type: string).*
hasAttribute*(e: person, name: "age", pos: 2, type: 'positive integer').*
hasAttribute*(e: person, name: "age", pos: 3, type: [person]).*

Note that addition of a container-class valued attribute (*friends*) triggered both a new fact in *hasAttribute* meta-relation and a new instance of *'container class'* meta class, whose object identifier is *[person]* (the same used in the value of attribute named *type*).

Meta reasoning

Since *meta*, the root of all meta-hierarchy is also a class (it descends from *object*), its instances can legally appear everywhere an instance of *object* (i.e., a value) is accepted. Thus, the above-mentioned meta classes and relations can be also used in every component of $\mathcal{O}ntoDLP$ based on reasoning. For example, we can use meta classes and relations to express queries as follows:

- *hasAttribute(entity: E, type: company)?* asks for all the entities having an attribute of type *company*;
- *C: 'base class'()?* asks for all the user define base classes;
- *isaClosure(subentity: web_programming_language, superentity:E)?* asks for all the superclasses (direct or indirect) of class *web_programming_language*.

Let's look at some more examples about meta reasoning.

Example 4.3. If we have the ontology:

class *furniture'().*
class *'modern furniture'* ***isa*** *{furniture} ().*
class *'classic furniture'* ***isa*** *{furniture} ().*
class *office ().*
class *'styled office'* ***isa*** *{office} (style:* ***class****).*

And we want to constrain attribute *style* of class *'styled office'* to accept, in actual instances of the latter, only subclasses of *furniture*, we can add the axiom:

/(a)*/ ::- StyledOffice: 'styled office'(style: StyleClass),* ***not isaClosure****(sub: StyleClass, super: furniture).*

Thus, instances:

```
'office #1': 'styled office'(style: 'classic furniture').
'office #2': 'styled office'(style: 'modern furniture').
```

Would pass axiom checking, whereas:

```
'bad office #1': 'styled office'(style: individual).
'bad office #2': 'styled office'(style: office).
```

would be marked as invalid, making the ontology inconsistent.

Note that, since *isaClosure* is not reflexive, i.e., for each class or relation *x*, it never holds *isaClosure(x, x)*, instances of *'styled office'* having the value *furniture* for attribute *style* would be also marked invalid by the axiom (a).

If this is undesired behavior, and *furniture* has to be considered as a legal value, one can always first process interesting tuples from *isaClosure* using a collection class or an intensional relation, then use the latter in a (modified) axiom.

For example, using a collection class we can extract the object identifiers of all *furniture* subclasses, plus *furniture* class itself, with a code like this:

```
collection class 'valid furniture class'()
{
furniture: 'valid furniture class'().
F: 'valid furniture class'() :- O: 'valid furniture class'(), isa(F, O).
}
```

The axiom could then be changed to:

```
::- StyledOffice: 'styled office'(style: StyleClass), not StyleClass: 'valid furniture class'().
```

5

OWL Interoperability

As discussed in Chapter 1, $\mathcal{O}ntoDLP$ is more suitable than OWL for Enterprise Ontologies, while OWL has been conceived for describing and sharing information on the Web (i.e., to deal with Web ontologies). However, it may happen that enterprise systems have to share or to obtain information from the Web; thus, from inside an enterprise ontology, one may need to access and query an external OWL ontology for specific purposes. At the same time, it is well known that Semantic-Web applications may need to integrate rule-based inference systems, to enhance their deductive capabilities. Based on these observations, our system supports some mechanisms for OWL interoperability[1].

The first mechanism is an import (from OWL) and export (to OWL) facility, whose principles are sketched in Sec. 5.1 and Sec. 5.2; in Sec. 5.3 we define two important properties of import/export facility, guaranteeing syntactic and semantic equivalence for a language fragment.

In the last section, 5.4, we explain a further mechanism, *OWL Atoms*, that enables $\mathcal{O}ntoDLP$ ontologies to reason on top of (even multiple) OWL ontologies.

5.1 Importing OWL in $\mathcal{O}ntoDLP$

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

5.1.1 OWL Thing ($\top$) and OWL Nothing ($\bot$).

The OWL universal class *Thing* corresponds to the $\mathcal{O}ntoDLP$ class *individual* (because both are the set of all individuals). Conversely, in $\mathcal{O}ntoDLP$ we cannot directly express the empty class $\bot$, but we approximate it as follows:

class 'Nothing'. ::− X:*Nothing()*.

[1] In this chapter we assume the reader to be familiar with semantics and Description Logics syntax of OWL.

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

Please note that the name of the class *Nothing* is written as *'Nothing'* in $\mathcal{O}ntoDLP$ syntax. This is mandatory, since names starting with a capital letter and not enclosed by single quotes could be confused, at syntactic level, with variables. See Sec. 3.2.2 for more on this matter.

5.1.2 Atomic classes and class axioms ($C, C \sqsubseteq D$).

Atomic classes are straightforwardly imported in $\mathcal{O}ntoDLP$. For example, we write: **class** $Person()$to import the specification of the atomic class *Person*.

Inclusion axioms directly correspond to the *isa* operator in $\mathcal{O}ntoDLP$. Thus, the statement $Student \sqsubseteq Person$ (asserting that student is a subclass of person) is imported by writing: **class** 'Student' **isa** 'Person'.

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

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

Another class axiom provided by OWL, called *disjointWith*, asserts that two classes are disjoint. We approximate this behavior by using an axiom in $\mathcal{O}ntoDLP$. For example:

$Man \sqcap Woman \sqsubseteq \bot$

in represented in $\mathcal{O}ntoDLP$ using the axiom:

::– X: *'Man'()*, X:*'Woman'().*

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

5.1.3 Enumeration classes $\{a_1, ..., a_n\}$.

A class can be defined in OWL by exhaustively enumerating its instances (no individuals exist outside the enumeration).

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

$RGB \equiv red, green, blue$

we will import it in $\mathcal{O}ntoDLP$ by using a collection class in this way:

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

and we also add to the resulting ontology, the axiom $::-$ #`count` $\{$ *X: X:* [2] *'RGB'()* $\} > 3$. in order to correctly fix the number of admissible instances of the class.

5.1.4 Properties and Restrictions ($\forall$, $\exists$, $\lesseqgtr nR$).

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

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

collection class *'Parent'* $\{$
X: *'Parent'() :- hasChild(X,Y)*, Y: *'Person'().* $\}$

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

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

$HappyFather \sqsubseteq \forall hasChild.RichPerson$

In practice, an individual is an happy father if all its children are rich. The above statement can be imported in $\mathcal{O}ntoDLP$ in the following way:

collection class *'RichPerson'* $\{$
Y: *'RichPerson'() :- hasChild(X,Y)*, X: *'HappyFather'().* $\}$

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

African $\equiv \exists$ *bornIn.africa*

where *africa* is a specific individual representing the mentioned continent. To import it in $\mathcal{O}ntoDLP$, we write:

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

[2] The first *X* indicates the aggregation variable of #`count`, the second one is the variable of class literal for class *'RGB'*, which will assume, during grounding, the value of all object identifiers of *'RGB'* instances.

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

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

$ShyPerson \equiv \leq 5 hasFriend$

To import it in $\mathcal{O}ntoDLP$ we write:

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

Note that, the aggregate function $\#$`count` (see [51]) allows one to infer all the individuals having less than (or exactly) five friends.

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

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

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

when we import this in $\mathcal{O}ntoDLP$ we obtain:

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

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

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

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

intensional relation *hasChild (domain: 'Parent', range: 'Person')* $\{$
  $hasChild(X,Y) \text{:-} hasParent(Y,X).$ $\}$
intensional relation *hasParent (domain: 'Person', range: 'Parent')* $\{$
  $hasParent(X,Y) \text{:-} hasChild(Y,X).$ $\}$

Similarly, the transitive property *ancestor*, is imported in $\mathcal{O}ntoDLP$ as:

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

A classic example of symmetric property is the property *marriedWith*. We can import such a property into $\mathcal{O}ntoDLP$ as:

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

Moreover, OWL functional and inverse functional properties are encoded by using suitable $\mathcal{O}ntoDLP$ axioms. For example, consider the functional property *hasFather* and its inverse functional property *childOf*; they are imported in $\mathcal{O}ntoDLP$ as:

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

5.1.5 Intersection, Union and Complement ($\sqcap$, $\sqcup$, $\neg$).

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

$Woman \equiv Person \sqcap Female$

This expression is imported in $\mathcal{O}ntoDLP$ as:

collection class *'Woman'* ***isa*** $\{$ *'Person', 'Female'*$\}$*()* $\{$
X: *'Woman'()* :- X: *'Female'()*, X: *'Person'().* $\}$

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

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

$Parent \equiv Mother \sqcup Father$

and the following is the result of the import of this axiom in $\mathcal{O}ntoDLP$:

collection class *'Parent'* $\{$
X: *'Parent'()* :- X: *'Mother'().*
X: *'Parent'()* :- X: *'Father'().* $\}$

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

$InedibleFood \equiv \neg EdibleFood$

and, we import it in $\mathcal{O}ntoDLP$ by using *negation as failure* as follows:

collection class *'InedibleFood'* $\{$
X: *'InedibleFood(')* :- X: *individual ()*[3], not X: *'EdibleFood'().* $\}$

5.1.6 Individuals and datatypes.

The import of the ABox of a OWL ontology is straightforward; and actually, the A-Box assertions are directly imported in $\mathcal{O}ntoDLP$ facts. For example, consider the following

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

which are, thus imported in $\mathcal{O}ntoDLP$ as:

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

OWL makes use of the RDF(S) datatypes which exploit the XMLSchema datatype specifications[62].

$\mathcal{O}ntoDLP$ supports only a subset of XMLSchema datatypes; to import unsupported OWL datatypes, thus, we encode each datatype property filler in an $\mathcal{O}ntoDLP$ string that univocally represents its value.

5.2 Exporting $\mathcal{O}ntoDLP$ in OWL

In this section, we informally describe how an $\mathcal{O}ntoDLP$ ontology is exported in OWL by using some example.

5.2.1 Classes.

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

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

However, $\mathcal{O}ntoDLP$ class attributes do not have a direct counterpart in OWL, and we represent them introducing suitable properties and restrictions. Suppose that the class *Student* has an attribute *advisor* of type *Professor.* To export it in OWL, we first create a the functional property *advisor*, with *Student* as domain and *Professor* as range; and, then we export the class Student as $Student \sqsubseteq \forall advisor.Professor$.[4]

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

5.2.2 Relations.

We can easily export binary (base) relations and inheritance hierarchies in OWL, since the destination language natively supports them. In particular, isa statements are translated in inclusion axioms, and domain and range description allowed us to simulate the attributes. For relations having arity greater than two, we adopt the accepted techniques described in the W3C Working Group Note on n-ary Relations [47].[5]

5.2.3 Instances.

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

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

then we can export it in OWL as:

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

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

5.2.4 Collection classes and intensional relations.

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

$ancestor(X, Z) \mathrel{:-} ancestor(X, Y), ancestor(Y, Z).$

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

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

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

5.2.5 Axioms and Reasoning Modules.

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

5.3 Theoretical Properties

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

5.3.1 Syntactic Equivalence.

Let $import(O_{owl})$ and $export(O_{dlp})$ denote, respectively, the result of the application of our import and export strategies to OWL ontology O_{owl} and $\mathcal{O}ntoDLP$ ontology O_{dlp}.

Theorem 5.1. *Given a OWL DL ontology O_{owl}, and an $\mathcal{O}ntoDLP$ ontology O_{dlp} without class attributes and n-ary relations, we have that:*
(i) $export(import(O_{owl})) = O_{owl}$, and
(ii) $import(export(O_{dlp})) = O_{dlp}$.

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

A synopsis of transformations performed during OWL import/export is given in Table 5.1.

5.3.2 Semantic Equivalence.

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

Theorem 5.2. *Let Γ, Γ^R and Γ^L be the following sets of class descriptors: $\Gamma = \{A, B \sqcap C, \exists R.o\}$, $\Gamma^R = \Gamma \cup \{\forall R.C\}$, $\Gamma^L = \Gamma \cup \{\exists R.C, \sqcup\}$, and, let O_{owl} be an ontology containing only:*

$\mathcal{O}ntoDLP$	OWL DL	FOL
Base elements		
class a().	A	
class a() isa { b }.	$A \sqsubseteq B$	$\forall x.(A(x) \rightarrow B(x))$
relation r(…,…). (*binary*)	R	
relation r (…,…) isa { p }	$R \sqsubseteq P$	$\forall x, y.(r(x,y) \rightarrow p(x,y))$
class individual. (*built-in*)	$\top$	$(x = x)$
class 'owl::nothing'. ::- X: 'owl::nothing'().	$\bot$	$\neg(x = x)$
::- X: c1(), X: c2().	$C1 \sqcap C2 \sqsubseteq \bot$	
Collection class rules		
X: a() :- X: c(). X: c() :- X: a().	$A \equiv C$	$\forall x.(B(x) \rightarrow A(x))$ $\forall x.((A(x) \rightarrow B(x)))$
X: a() :- X: c1(), X: c2(). class a isa c1, c2().	$A \equiv C1 \sqcap C2$	$\forall x.(C1(x) \wedge C2(x) \rightarrow A(x))$ $\forall x.(\Lambda_{i \in [1,2]}(D(x) \rightarrow C_i(x)))$
X: a() :- X: c1(). X: a() :- X: c2().	$A \equiv C1 \sqcup C2$	$\forall x.((C1(X) \rightarrow A(X)) \wedge (C2(X) \rightarrow A(X)))$ $\forall x.(A(x) \rightarrow C1(x) \vee C2(X))$
X: a() :- X: individual(), not X: c().	$A \equiv \neg C$	$\forall x.(\neg A(x) \vee \neg B(x) \wedge (A(x) \vee B(x)))$
Y: c() :- r(X,Y),X: a().	$A \equiv \forall R.C$	$\forall x.\exists y.((R(x,y) \vee A(x)) \wedge (C(y) \rightarrow A(x)))$ $\forall x, y.((A(x) \wedge R(x,y)) \rightarrow C(y))$
X: a() :- r(X,Y),Y: c().	$A \equiv \exists R.C$	$\forall x.((A(x) \rightarrow R(x, f(x))) \wedge (A(x) \rightarrow C(f(x))))$ $\forall x, y.((C(y) \wedge R(x,y)) \rightarrow A(x))$
X: a() :- r(X, o). r(X, o) :- X: a().	$A \equiv \exists R.o$	$\forall x.(R(x,o) \rightarrow A(x))$ $\forall x.(A(x) \rightarrow R(x,o))$
a_1 : a() … a_n : a(). $::- \#\texttt{count}\{X : a()\} > n.$	$A \equiv \{a_1, ..., a_n\}$	$\forall x.((x = a_1 \vee \cdots \vee x = a_n) \rightarrow A(x))$ $\forall x.(x = a_1 \vee \cdots \vee x = a_n \vee \neg A(x))$
X: a() :- r(X,_), #count {Y: r(X,Y)} φ n. $\varphi \in \{=, <=, >=\}$	A $\equiv \varphi$ n $R.C$ $\varphi \in \{=, <=, >=\}$	
Intensional relation rules		
r(X, Y) :- S(X, Y). s(X, Y) :- r(X, Y).	$R \equiv S$	$\forall x, y.((S(x,y) \rightarrow R(x,y))$ $\forall x, y.((R(x,y) \rightarrow S(x,y)))$
X: c() :- r(X,_).	$\top \sqsubseteq \forall R.C$	$\forall x, y.(R(x,y) \rightarrow C(x))$
Y: c() :- r(_,Y).	$\top \sqsubseteq \forall R^-.C$	$\forall x, y.(R(x,y) \rightarrow C(y))$
r(X,Y) :- s(Y,X). s(X,Y) :- r(Y,X).	$R \equiv S^-$	$\forall x, y.((S(y,x) \rightarrow R(x,y))$ $\forall x, y.((R(y,x) \rightarrow S(x,y))$
r(X,Y) :- r(Y,X).	$R \equiv R^-$	$\forall x, y.((R(x,y) \rightarrow R(y,x))$
r(X,Z) :- r(X,Y), r(Y,Z).	$R^+ \sqsubseteq R$	$\forall x, y, z.((R(x,y) \wedge R(y,z) \rightarrow R(x,z))$
$::- r(X, \_), \#\texttt{count}\{Y : r(X,Y)\} > 1.$	$\top \sqsubseteq \leq 1.R$	
$::- r(\_, Y), \#\texttt{count}\{X : r(X,Y)\} > 1.$	$\top \sqsubseteq \leq 1.R^-$	
Instances		
o : c().	$o : C$	$C(o)$
r(o ,o1).	$< o, o1 >: R$	$R(o, o1)$

Table 5.1. Synopsis of OWL import/export facility

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

then import(O_{owl}) under the $\mathcal{O}ntoDLP$ semantic entails precisely the same consequences as O_{owl}.

Intuitively, the equivalence property holds because[6] the First Order Theories equivalent to the admitted fragment of OWL only contains Horn equality-free formulae whose semantics corresponds to the one of the produced logic program.

5.4 $\mathcal{O}ntoDLP$ reasoning on top of OWL ontologies

In the following, we describe how to import OWL knowledge into $\mathcal{O}ntoDLP$ ontologies and how this information can be exploited to write reasoning modules (and, thus, logic programs) that allow one to add rules and reason on top of OWL.

To enable the interfacing and import of existing OWL ontologies into the framework of $\mathcal{O}ntoDLP$, the so-called *OWL Atoms* have been introduced[7]. OWL Atoms can be used in rule bodies of $\mathcal{O}ntoDLP$ reasoning components and facilitate the evaluation of specific queries to an OWL knowledge base. This allows to import ABox data, like concept and role extensions, but also TBox information, like concept subsumption, ancestors and descendants. To comfortably handle the translation of names in this interfacing process, a *mapping* component can be specified.

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

5.4.1 OWL Atoms

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

[6] According to the approach of Borgida [12], also used in [34].
[7] Actually, we lifted the approach of [22] to the $\mathcal{O}ntoDLP$ framework.

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

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

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

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

The table in Figure 5.1 lists all possible query types and their according query syntax. `C` and `D` are here used as placeholders for concept names, `R` for a role name, and `I` for a name of an individual. `C`, `D`, and `I` must be ground. `?X` and `?Y` stand for variables.

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

For example, the OWL atom

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

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

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

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

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

The next rule imports the extension of a class into the $\mathcal{O}ntoDLP$ program:

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

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

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

query type	possible query	meaning
`allConceptNames`	`?X`	`?X` is instantiated with all concepts names
`allRoleNames`	`?X`	`?X` is instantiated with all role names
`allIndividuals`	`?X`	`?X` is instantiated with all individual names
`satisfiable`	`C`	the query atom is true if `C` is satisfiable
`subsumes`	`C D`	the query atom is true if `C` subsumes `D`
`disjoint`	`C D`	the query atom is true if `C` and `D` are disjoint
`parents`	`?X C`	`?X` is instantiated with all parent concepts of `C`
`children`	`?X C`	`?X` is instantiated with all children concepts of `C`
`ancestors`	`?X C`	`?X` is instantiated with all ancestor concepts of `C`
`descendants`	`?X C`	`?X` is instantiated with all descendant concepts of `C`
`equivalents`	`?X C`	`?X` is instantiated with all concepts that are equivalent to `C`
`rparents`	`?X R`	`?X` is instantiated with all parent roles of `R`
`rchildren`	`?X C`	`?X` is instantiated with all children roles of `R`
`rancestors`	`?X C`	`?X` is instantiated with all ancestor roles of `R`
`rdescendants`	`?X C`	`?X` is instantiated with all descendant roles of `R`
`instances`	`C(?X)`	`?X` is instantiated with all instances of `C`
`types`	`?X(I)`	`?X` is instantiated with all concepts subsuming the individual `I`
`relatedIndividuals`	`R(?X,?Y)`	`?X,?Y` is instantiated with all pairs of the ObjectProperty `R`
`toldValues`	`R(?X,?Y)`	`?X,?Y` is instantiated with all pairs of the DatatypeProperty `R`

Fig. 5.1. OWL atom query types

5.4.2 Name Mappings

Mappings ease the syntactic translation of constant names when they are imported into the $\mathcal{O}ntoDLP$ program. A mapping is defined via the `mapping` keyword. It is used like a module:

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

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

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

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

Mappings are not functional, hence they can be seen as $n : n$ relations. Consequently, one name can be mapped to multiple replacement names, which will all be inserted, and multiple names can be mapped to the same single replacement name. It is in the responsibility of the author of a mapping to consider the effect of such mappings on the Unique Name Assumption.

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

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

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

mapping persons {
    ‘*john*’ ‘*http://www.example.org/#Doe*’
    ‘*jane*’ ‘*http://www.example.org/#Smith*’
}
mapping family {
    ‘*johnny*’ ‘*http://www.example.org/#Doe*’
}

and the query atom above. Assume that the user specifically requested functional mappings. Considering the namespaces, two conflicting mappings for the external name ‘*http://www.example.org/#Doe*’ exist. Since the mapping *persons* is specified before *family*, the name will be translated into ‘*john*’.

Part II

The system: $\mathcal{O}ntoDLV$

6

The $\mathcal{O}ntoDLV$ System

$\mathcal{O}ntoDLV$ is a complete tool that supports the development of $\mathcal{O}ntoDLP$ ontologies and permits to specify, navigate, query and reason. It is a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The $\mathcal{O}ntoDLV$ system allows one for the development of complex applications and allows to perform advanced reasoning tasks in third-party applications, via an advanced Application Programming Interface. The $\mathcal{O}ntoDLV$ system seamlessly integrates the DLV system (together with DLV^{DB}) exploiting the power of a stable and efficient DLP solver.

In this chapter we illustrate the overall $\mathcal{O}ntoDLV$ architecture, and we highlight the features of the system together with its main components.

6.1 System Architecture

We now illustrate the $\mathcal{O}ntoDLV$ architecture, and present the main features of the system.

The system architecture of $\mathcal{O}ntoDLV$, depicted in Figure 6.1a, can be divided in three abstraction levels. The lowest level, named $\mathcal{O}ntoDLV$ *core* contains the components implementing the main functionalities of the system, namely: *Persistency Manager*, *Type Checker*, and *Rewriter*. The Persistency Manager provides all the methods needed to store and manipulate the ontology components. In particular, it exploits the *Parser* submodule to analyze and load the content of several $\mathcal{O}ntoDLP$ text files (together with a *Stream Module*, and a *RDB Module* submodule to implement data persistency on relational databases through Hibernate/JDBC.

The admissibility of an ontology is ensured by the *Type Checker* module which implements a number of type checking routines.

The *Rewriter* module translates $\mathcal{O}ntoDLP$ ontologies, axioms, reasoning modules and queries to an equivalent DLP program that runs on the DLV system [40] ; either the results or possible error messages are redirected to the *Persistency Manager*. Importantly, *ontologies* are translated into an equivalent (stratified) DLP program which is solved by DLV in polynomial time (under data complexity). More-

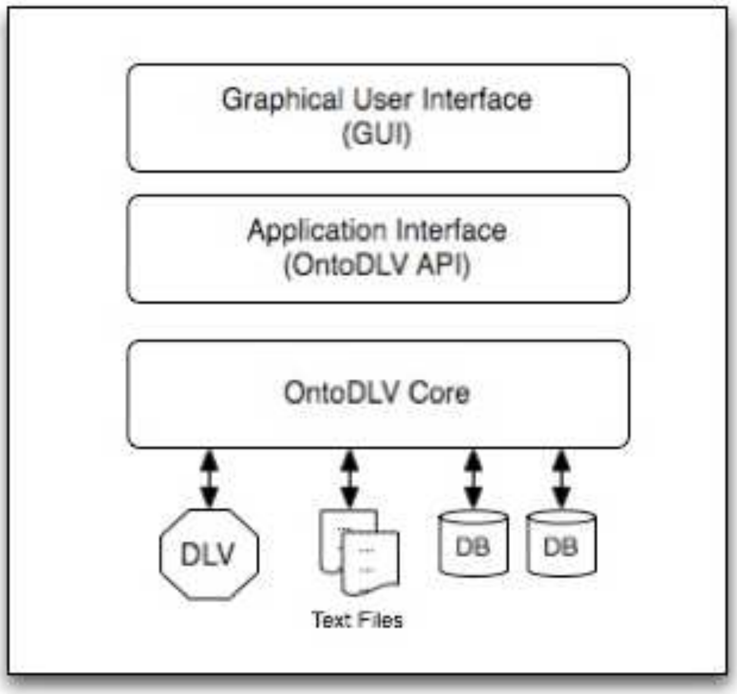

Fig. 6.1. The $\mathcal{O}ntoDLV$ architecture.

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

$\mathcal{O}ntoDLV$ system can exploit DLV^{DB}, rather than DLV, for computations than can take advantage of direct execution on relational databases.

6.2 Persistency

Persistency manager can handle different *storage engines*, represented as "repositories"; anyone can develop its own interface module towards a particular storage medium. Each *storing engine* is responsible for performing some high-level browsing operations (see Chap. 7) efficiently on its own, and is free to exploit whichever resources fit to achieve this goal.

$\mathcal{O}ntoDLV$ natively provides two implementations of the storage engine concept:

- the *Stream Module*
- the *RDB Module*

The two implementations both rely on a common base, named *Common Repository Layer*; to promote reuse, the $\mathcal{O}ntoDLP$ *Parser* has been developed as an independent module.

The parser module

We refrained from building a parser with fixed semantic actions for the $\mathcal{O}ntoDLP$ language, which would have blocked reuse; instead, we built a parser that is able to incrementally generate nodes of an Abstract Syntax Tree (AST). Text streams are converted to syntactic node streams (one AST per single syntactic construct), in a format suitable to feed other modules (e.g., , for bulk loading operations).

By using *JavaCC* parser generator (plus *JJTree* for AST generation), we are also able to associate each node of the AST with its underlying tokens (i.e., actual text and its coordinates). This, in turn, allows one for building other applications, like e.g., an $\mathcal{O}ntoDLP$ editor with syntax coloring capabilities, etc.

The *parser module* is also able to acquire the output of DLV output, which is then translated back into $\mathcal{O}ntoDLP$ constructs. In this case, no AST nodes are generated, since this functionality is used only internally.

The Common Repository Layer

This module has the main goal of ensuring a consistent behavior for each repository, either user-defined or system-defined. Speaking in terms of object-oriented language, the *Common Repository Layer* has the role of an *intermediate abstract class*. Here, we are referring to a classical scheme of development in object-oriented languages, where a concept, modeled at the highest level via a purely abstract class, is implemented through some steps, each reducing the abstraction level, until a concrete class is eventually defined. At each step only methods whose implementation is clear with respect to the particular abstraction level are defined, implementing the other in the levels beneath.

Following this scheme, *Common Repository Layer* defines abstract classes implementing some methods from the *storage engine* concept, leaving implementation of storage-specific issues to concrete storage modules, such as *Stream Module* and *RDB Module*.

The Stream Module

This module is a storage engine that uses stream-oriented resources, such as *files*, as the storage medium; it works entirely in main memory, which is used to keep the current content of the ontology.

The underlying stream (like, e.g., a *java.io.File*) can be synchronized in both ways with the ”ontology image”.

The latter is kept in an extremely tight format, in order to reduce memory allocation. Moreover, the content of ontology is also automatically enriched with a number of pre-computed information, such as indexes, to speed up access.

In fact, a goal of the module is to keep feasible the handling in main memory of huge ontologies.

To achieve this goal, *memoization* techniques, which are *garbage collector-friendly*, have been employed. Indeed, even when a memoized information is al-located[1], if Java garbage collector detects that memory occupied by that information is needed for other purposes, it can reclaim that space immediately.

Balance between the retention of all pre-computed information and a minimal usage of memory is thus automatic; it is regulated by the Java Virtual Machine itself.

This way, on machines with plenty of memory with respect to the size of ontology being handled, the *Stream Module* balances towards massive usage of pre-computed information, to enhance performance. On the contrary, when memory is scarce with respect to the size of the ontology, this module tends to keep in memory only a ”core” of information, smoothly degrading performance levels, without failing for, e.g., a dreaded *OutOfMemoryError*.

The RDB Module

RDB module stores ontology content in a database. Entities and reasoning constructs are stored as tuples of a fixed relational schema; the mappings between relational database tables and the object model is managed by the Hibernate framework for Object/Relation Mapping (ORM).

As opposed to intensional/schema part, instances of base classes and tuples are stored in a dynamic relational schema, managed by the module itself (via DDL operations) whenever a (ontology) schema change occurs.

While *Stream Module* needs to maintain some indexes in memory in order to speed up data access, *RDB Module* can get rid of any locally-stored information, leaving the burden of data indexing to database.

As said early in this section, each module is free to choose a plan to execute efficiently user-requested tasks; to do so, *RDB module* exploits SQL querying capabilities, building and executing even complex queries (depending on user requests), whose results are merged as needed and presented to the user of the module in a format that is common to other *storage engines*.

For example, the request to execute this expression (see Sec 7.1):

```
ontology.baseClassInstances().havingValue("a", 1974)
```

(”give me all instances of base classes that have an attribute named ’a’, whose value is 1974”)

is internally translated to multiple executions of the SQL query:

```
SELECT * FROM <X> WHERE a = 1974
```

where $< X >$ takes the names of tables, used for storing instances of base classes, which have an attribute named ”a”. Results are then merged in a final step, invisible to the user.

[1] It is worth noting that memoized information is always referenced via a *java.lang.SoftReference*, rather than using garbage collector-hostile *hard references*.

6.3 Type checking and rewriting

The ontology stored in a set of storage engines can be browsed in a global view in a transparent way. Type-checking and rewriting procedure indeed exploit this feature, in order to check the admissibility of an ontology, and to perform a ”translation” of $\mathcal{O}ntoDLP$ code in an ”equivalent” DLP encoding, respectively.

The rewriter module

The rewriter module performs translations from $\mathcal{O}ntoDLP$ code to DLP code, when a reasoning task is requested, taking into account the language extensions. In particular, among other things, the rewriter must efficiently:

- select the minimum set of constructs to be translated, depending on the kind of task requested;
- transform $\mathcal{O}ntoDLP$ values into DLP ones, ensuring that no clash happens between values of different types (extended names, strings, numbers, etc.), though they have to be mapped, in DLP, using only two domains, natural numbers and strings;
- translate operations on extended types to DLP, using native DLV operations or external built-ins, as needed;
- flatten extended constructs (like, e.g., complex terms), possibly creating intermediate predicates in generated DLP program;
- create rules corresponding to the various $\mathcal{O}ntoDLP$ constructs;
- create rules without body for the instances of entities.

The type checker

Type checking algorithms work on the whole ontology, in order to assess validation status of each ontology component.

The result of the *type checker* module is not a simple status assessment (i.e., ”valid” or ”not valid”) for the whole ontology; rather, the system is able to ideally ”split” the ontology in two parts, *valid* and *not valid*, allowing the user to work on both (which is useful during development of ontologies), or just on one of them.

Sophisticated checks are implemented; for example, in each entity the hierarchy chain is checked for inconsistencies, the type of attributes is controlled, etc. For intensional entities forming a group in a a dependency graph, a check for unstratified negation is also performed, using Tarjan’s algorithm. In general, the consistency between entity declarations and entity occurring in literals is checked.

Every status assessment contains details of type checking violations (”issues”) found, which is useful as *debugging* information for $\mathcal{O}ntoDLP$ developers.

6.4 Mass-memory query execution

DLV^{DB} is a version of DLV capable to evaluate normal stratified programs in mass memory (i.e., a relation database is exploited by DLV^{DB} in order to compute answer sets). DLV^{DB} requires to map each predicate of a program to a database table.

The translation process described for the *Rewriter module* is tailored for main memory execution. However, the $\mathcal{O}ntoDLV$ system is also able to take advantage of DLV^{DB} capabilities, if the ontology is stored on a relation database, thus realizing an *execution of queries on mass-memory*.

The rewriting procedure of $\mathcal{O}ntoDLV$ has been extended in order to produce a logic program plus the required mappings. DLV^{DB} leaves the output of the computation on the database, and $\mathcal{O}ntoDLV$ handles output accordingly. In particular, results are taken from the database instead of being rebuilt starting from output of DLV, but the technicalities of this process are hidden from the user of $\mathcal{O}ntoDLV$ system, which sees, through API, a query reasoning system.

7

$\mathcal{O}ntoDLV\ API$: embedding $\mathcal{O}ntoDLV$ in third-party applications

Integrating a complex reasoning engine in a system is not easy, especially when an ”impedance mismatch” exists between the reasoner and the third-party system needing integration.

We provided $\mathcal{O}ntoDLV$ with an Application Programming Interface (API), written in Java. The main goal of $\mathcal{O}ntoDLV\ API$ is to offer a set of classes and methods easy enough to be used by casual developer. This in turn requires to exploit basic, though fundamental, concepts of Java programming like *Collections*, *Iterators*. Exposing a minimal set of information is also crucial to leave enough room for internal optimizations.

In the following sections, we explore the functionalities of API for ontology browsing (Sec. 7.1), then we describe functionalities that control reasoning tasks execution (Sec. 7.2). A sample application illustrating API usage is eventually developed in Sec. 7.3.

7.1 Browsing an ontology

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 classes. It is worth noting that the $\mathcal{O}ntoDLV\ API$ is characterized by a rather high level of abstraction, and it is composed of a relatively rich set of Java interfaces, together with a single factory class (like, e.g., the JAXP API from Sun[1]). However, the extensive usage of standard Java components (e.g., both the interfaces $Collection$ and $Iterator$ play a central role) makes expert programmers rapidly familiar with the $\mathcal{O}ntoDLP$ API.

In the following, we describe the core components of $\mathcal{O}ntoDLV\ API$ and we sketch its working principles.

[1] Though the two APIs are aimed at rather different tasks, some of the design principles are in common, such the usage of ”abstract factory” pattern.

Core API Components

In the core part of the $\mathcal{O}ntoDLV\ API$ each language construct (class schema, relation schema, instance, etc.) has an associated Java concept describing it. To achieve a high level of abstraction we decided to use interfaces rather than classes, In particular, the available Java interfaces are:

- $BaseClass$ for base classes (see Sec. 3.1.1);
- $BaseClassInstance$ for instances of base classes (see Sec. 3.1.2)
- $BaseRelation$ for base relations (see Sec. 3.1.4)
- $Tuple$ for instances of base relations (see Sec. 3.1.4)
- $CollectionClass$ for collection classes (see Sec. 3.3.2)
- $IntensionalRelation$ for intensional relations (see Sec. 3.3.3)
- $Mapping$ for name mapping, used in OWL Atoms (see Sec. 5.4.2)
- $Query$ for $\mathcal{O}ntoDLP$ queries (see Sec. 3.4.3)
- $Axiom$ for axioms (see Sec. 3.4.1)
- $ReasoningModule$ for reasoning modules (see Sec. 3.4.2)

All the above-mentioned interfaces extend the common interface $Component$, which defines precisely the concept of an $\mathcal{O}ntoDLP$ language component.

Access points for ontology browsing

Concrete implementors of all the above component interfaces are accessible through an interface containing a set of browsing methods, called $ComponentBrowser$. In particular, each $ComponentBrowser$ provide a view of a set of components (either coming from a single storage engine) it ”owns”, and provides some methods that return lists of components, namely:

- $baseClasses()$, which returns a list of all base classes;
- $baseClassInstances()$, which returns a list of all instances of base classes;
- $baseRelations()$, which returns a list of all base relations;
- $tuples()$, which returns a list of all instances of base relations (tuples);
- $collectionClasses()$, which returns a list of all collection classes;
- $intensionalRelations()$, which returns a list of all intensional relations;
- $mappings()$, which returns a list of all name mappings;
- $queries()$, which returns a list of all queries;
- $axioms()$, which returns a list of all axioms;
- $modules()$, which returns a list of all reasoning modules.

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

```
  for (BaseClass cl: cb.baseClasses()) {
  \  System.out.println(cl);
}
```

It is worth noting that these lists are not "materializations" of the corresponding entities; they rather represent virtual "views" aggregating a set of objects, possibly coming from many sources (e.g., different physical storage [2]), and they are implementations of Java standard $Collection$s, which henceforth can be manipulated using well-known Java methods such as $add()$, $contains()$, $remove()$, etc.

Selectors

The principle exploited above, which is based on lists of $Components$, is applied to browse the content of schemas and instances. For example, the $BaseClass$ component has a method which returns the list of all superclasses of the given class object. Moreover, the lists returned by the browsing methods also provide the user the ability to perform *selections* over the set of objects through specialized methods. Those methods, called "selectors", return a list of the same kind as the one they were called on (cascading calls are allowed), but filtered on the basis of a given criterion. Lists and selectors form, in fact, an Embedded Domain Specific Language [26], based on Java, for accessing the ontology content.

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

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

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

```
BaseClassInstanceList specialInstances =
componentBrowser.baseClassInstances().havingValue(1974)
.havingValue("Rome");
```

Moreover, *havingValue()* may take an additional parameter, representing the name of the attribute to be matched.

As an example:

```
componentBrowser.baseClassInstances()
.havingValue(a, myValue)
```

selects all base class instances having an attribute "a" with specified value.

[2] As described in Sec. 6.2 $\mathcal{O}ntoDLV\ core$ supports both filesystem and database persistency, which are handled transparently by the API

Repositories and ontologies

The aforementioned $ComponentBrowser$ interface is implemented by two other interfaces:

- $Repository$, which represents, as a Java object, a physical source;
- $Ontology$, which is a integrated view of a set of contributions, coming from $Repositories$.

Ontology serves instead as a mixing point for lists of components provided by *Repositories*; e.g., when asked for a list of base classes, an *Ontology* returns a *composite* list aggregating lists coming from various storages. Moreover, composite lists do not evaluate selectors directly; instead, they take care of properly ”pushing down” selectors to underlying lists. This way, each *Repository* providing a list has the opportunity to independently execute associated selectors.

In fact, *Repositories* are responsible for translating navigational paths, built by chaining selectors, into an optimal query to underlying storage system.

The implementation of *Repository* in *RDB Module* (see Sec. 6.2) exploits SQL during selectors evaluation, whereas in *Stream Module* selectors are evaluated with respect to a set of indexes.

For example, if *ontology* integrates contributions from a *repositories r1* and *r2*, the following query, expressed via selectors:

```
BaseClassInstanceList result =
ontology.baseClassInstances().havingValue(myValue).
fromSchemas(ontology.baseClasses().named(namespace))
```

is *somewhat* equivalent to:

```
BaseClassList interestingClasses =
ontology.baseClasses().named(namespace);

BaseClassInstanceList result = ....
result.addAll(r1.baseClassInstances().
    havingValue(myValue).fromSchemas(interestingClasses));
result.addAll(r2.baseClassInstances().
    havingValue(myValue).fromSchemas(interestingClasses));
```

Except for that the internal translation performed by *Ontology* class gives a *BaseClassInstanceList* that is not a copy, it is *active*, i.e., components may still be modified or removed.

Ontology maintains also a central directory of objects, indexed by name; for example, to access a base class with a given name, no matter where it is stored, one can use:

```
ontology.findBaseClassByName(myName);
```

Project

Project is the main starting point of API; it:

- has a set of *Repositories*
- may create implementation of *Repository*, either stream-based or RDB-based;
- provides an integrated view through an *Ontology*
- has an *Engine* object, to control reasoning tasks (see Sec. 7.2)

Instances of *Project* may be created using *ProjectFactory*, which follows the *abstract factory* pattern.

7.2 Controlling reasoning tasks

Reasoning functionalities are accessible via the *Engine* interface, which allows one for selecting either DLV engine, or DLV^{DB}, by switching the *InvocationStrategy*.

Engine is a factory for *Invocation* objects, which serve as reasoning task control ”knobs”. Four kinds of reasoning tasks are available:

- *model computation*, a set of reasoning modules is selected for execution, all the models computed (encompassing only auxiliary predicates defined in modules) are returned as output;
- *querying*, a given query is submitted to the system, truth value (for ground queries) or variable binding satisfying the query are returned (may be execute in *brave* mode, or in *cautious* mode);
- *consistency check*, the ontology is checked for consistency with respect to axioms, violations are detected and pointed out as result of this task;
- *instance computation*, the actual extension of intensional entities is computed and cached on a repository-specific medium (either in main memory or in database tables).

For the first three kinds of reasoning tasks, *Engine* returns specialized implementations of *Invocation* interface (e.g., *ModelComputationInvocation*, *QueryInvocation*, *Consistency-CheckInvocation*). Each invocation must be provided an implementation of *OutputBuilder* interface, tailored to the specific task, which is responsible for accepting (and, possibly, storing) reasoning results.

In order to simplify usage of reasoning engine, when an *OutputBuilder* is not set, default implementations (whose names end with *OutputResultProvider*) are provided, which simply store the latest result in a suitable Java object for further reference.

As an example a *QueryInvocation* has, by default, a *QueryOutputResultProvider* that returns *QueryResult*s; the latter may be explored to gather results from query.

The last kind of invocation, is used internally, whenever the system has to maintain a snapshot of extension for each intensional entity. Access to the latter is mediated via a selector *instances()* on *CollectionClass* and *IntensionalRelation*, which triggers computation as needed.

7.3 A sample application

In this section, we show how to use $\mathcal{O}ntoDLV\ API$ by running an example. In particular, we describe a snippet of Java code which uses the API to deal with the living being ontology introduced in Section 3.1. We refrain from reporting all the technical details (package inclusions, main function declaration etc.), while we focus on the part of the code where the API methods are used. We report a program that executes the following four operations:

1. load a text file containing the living being ontology;
2. add some new data to the relation $friends$;
3. build the reasoning module $shyFriends$ described in Section 3.4.2;
4. perform the query *youngAndShy(X), X:person(name:”Jack”))?*, and print the obtained results in standard output.

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

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

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

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

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

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

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

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

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

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

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

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

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

Part III

Related work and conclusion

8

Related work

A number of languages and systems somehow related to $\mathcal{O}ntoDLP$ and $\mathcal{O}ntoDLV$ have been proposed in the literature. In this chapter we compare both the language and the system with other interesting work, highlighting differences and similarities.

Extensions of Datalog

Among systems descending from Datalog, COMPLEX [33] is the most closely related to $\mathcal{O}ntoDLV$; it supports the Complex-Datalog language, an extension of (non-disjunctive) Datalog with some concepts from the object-oriented paradigm. $\mathcal{O}ntoDLV$ and COMPLEX share a similar object-oriented model, however the language of the latter is less expressive than $\mathcal{O}ntoDLP$. In fact, COMPLEX supports normal (non-disjunctive) stratified programs only (its expressive power is confined to P), which are strictly less expressive than $\mathcal{O}ntoDLP$ language expressing even Σ_2^P-complete properties.

It is worth noting that other similar languages and systems have been quite successful and positively accepted in the literature (see e.g., [17, 1, 43]), even if they were based on less powerful logic programming languages.

Logic-related formalisms

Another popular logic-based object-oriented language is F-Logic [39], which includes most aspects of object-oriented and frame-based languages. F-logic was conceived as a language for intelligent information systems based on the logic programming paradigm. A main implementation of F-logic is the Flora-2 system [65] which is devoted to Semantic Web reasoning tasks. Flora-2 integrates F-Logic with other novel formalisms such as HiLog [16] (a logical formalism that provides higher-order and meta-programming features in a computationally tractable first-order setting) and Transaction Logic [3] (that provides a logical foundation for state changes and side effects in a logic programming language). Comparing $\mathcal{O}ntoDLP$ with F-Logic, we note that the latter has a richer set of object oriented features (e.g., class methods, while multi-valued attributes perfectly match $\mathcal{O}ntoDLP$ sets), but it misses some important constructs of $\mathcal{O}ntoDLP$ like disjunctive rules, which increase the knowledge modeling ability of the language. Concerning system-related aspects, two important advantages of $\mathcal{O}ntoDLV$ (w.r.t. Flora-2) are the availability of an Application Programming Interface, and the presence of a graphical development environment (not discussed here). The former eases the task of writing multi-platform Java applications on top of $\mathcal{O}ntoDLV$, while the latter simplifies the interaction with $\mathcal{O}ntoDLV$ for both the end user and the knowledge engineer. Moreover, interoperability mechanisms, which are available in $\mathcal{O}ntoDLV$, tend to lower the "knowledge gap" that has to faced when knowledge one wants to reason on is not encoded in the native formalism ($\mathcal{O}ntoDLP$).

Semantic Web

A couple of other formalisms for specifying ontologies have been recently proposed by W3C, namely, RDF/RDFS and OWL. The Resource Description Framework (RDF) [64] is a knowledge representation language for the Semantic Web. It is a simple assertional logical language which allows for the specification of binary properties expressing that a resource (entity in the Semantic Web) is related to another entity or to a value. RDF has been extended with a basic type system; the resulting language is called RDF Vocabulary Description Language (RDF Schema or RDFS). RDFS introduces the notions of class and property, and provides mechanisms for specifying class hierarchies, property hierarchies, and for defining domains and ranges of properties. Basically, RDF(S) allows for expressing knowledge about the resources (identified via URI), and features a rich data-type library (richer than $\mathcal{O}ntoDLP$), but, unlike $\mathcal{O}ntoDLP$, it does not provide any way to extract new knowledge from the asserted one (RDFS does not support any "rule-based" inference mechanisms nor query facilities).

The Ontology Web Language (OWL)[58] is an ontology representation language built on top of RDFS. The ontologies defined in this language consist of *concepts* (or classes) and *roles*(binary relations also called class properties). OWL has a logic based semantics, and in general allows to express complex statements about the domain of discourse (OWL is undecidable in general). The largest decidable subset of OWL, called OWL-DL, coincides, basically, with $\mathcal{SHOIN}(\mathbf{D})$, an expressive Description Logic (DL)[6]. OWL is based on classical logic (there is a direct mapping from $\mathcal{SHOIN}$to First Order Logic (FOL)) and, consequently, is quite different form $\mathcal{O}ntoDLP$, which is based on DLP. Compared to $\mathcal{O}ntoDLP$, OWL misses, for instance, *default negation*, *nonmonotonic disjunction*, and *inference rules*. "Rules", in particular, are considered an indispensable tool for enabling agents to reason about the knowledge represented in an ontology [58, 36]

The approach we took in this respect is, somehow, connected with the effort of combining OWL with rules for the Semantic Web (see [4] for an excellent survey). One of the major problems existing in the interaction of rules and description logics with strict semantic integration is retaining decidability (which is, instead, ensured in our framework) without loosing easy of use and expressivity. For instance, the SWRL[36] approach is undecidable; while, in the so-called DL-safe rules [46] a very strict safety condition is imposed to retain decidability. Notably, this safety condition has been recently weakened in some works [53, 52] thus obtaining a more flexible environment. However, the goal of the above-mentioned approaches is different from the one achieved in this paper.

Indeed, one may use $\mathcal{O}ntoDLP$ rules to reason on top of OWL ontologies; in particular, by means of our OWL Atoms (see Sec. 5.4, that can appear in the rules' bodies, one can query OWL ontologies and apply the powerful $\mathcal{O}ntoDLP$ reasoning mechanisms on the resulting knowledge.

In addition, we have provided also a mechanism to import simple OWL ontologies in $\mathcal{O}ntoDLP$ and to export $\mathcal{O}ntoDLP$ ontologies to OWL. We have singled out language fragments where the semantic equivalence is guaranteed, along the line of the approach proposed by B.N. Grosof et al. [34], who defined the so-called Description Logic Programs, to combine logic programming with DL.

The techniques exploited to obtain our import transformation are similar to (even if more pragmatic and less general than) the ones used for reducing description logics to logic programming (see [9, 59, 35, 37]).

DLV$^+$

Previous works on DLP extensions for ontology representation and reasoning have been carried out at the Mathematics Department of University of Calabria [14, 41, 51], and have led to the development of the DLV$^+$ prototype [51]. $\mathcal{O}ntoDLV$ finds its roots in the DLV$^+$ work, but, compared to DLV$^+$, $\mathcal{O}ntoDLV$ brings many relevant extensions, optimizations, and enhancements.

First, there is a number of additions to the language; compared to DLP$^+$, $\mathcal{O}ntoDLP$ additionally features:

- objects reclassification support through collection classes (see Sec. 3.3.2);
- intensionally defined relations (see Sec. 3.3.3), hierarchical relations (see Sec. 3.1.3);
- meta reasoning: the language handles seamlessly reasoning on the schema as well as on data (see Chap. 4);
- container types (i.e., lists and sets): the user is allowed to build and explore complex data structures, with unlimited nesting level, and use them as regular values (see Sec. 3.2.3);
- richer set of data types: the language can natively handle a wider range of types (e.g., dates, decimals, signed integers), both in simple operations and using aggregates (see Sec. 3.2.1);
- support for externally defined built-in predicates;
- wider range of acceptable names for named components (e.g., entities and individuals), support for namespaces (see Sec. 3.2.2);
- OWL interoperability mechanisms at language level: a kind of "import atoms" is supported, so as to making $\mathcal{O}ntoDLP$ able to reason on top of OWL ontologies (see Sec. 5.4).

Importantly, the system itself is fitted with new capabilities; compared to DLV$^+$, $\mathcal{O}ntoDLV$ features also:

- Application Programming Interface (API): **all** $\mathcal{O}ntoDLV$ functionalities are accessible through an easy-to-use programming interface that exposes concepts at a higher level (see Chap. 7);
- modular architecture, featuring a pluggable ontology storage engine: the system can use (and mix) different storage engines (e.g., on filesystem, on a relational database, etc.), for different parts of the ontology, still retaining an unified, virtual view of concepts defined in the ontology (see Sec. 6.2);
- mass-memory query execution: $\mathcal{O}ntoDLV$, rooted on DLV system, can switch seamlessly to DLV^{DB} engine for direct execution of queries on relation databases (see Sec. 6.4);
- OWL interoperability mechanism at system level: $\mathcal{O}ntoDLV$ has the ability to import from OWL and export to OWL (see Sec. 5.1, 5.2) ontologies, keeping entailment compatibility for a significant fragment of OWL language (see Sec. 5.3);

These enhancements over DLV$^+$ make $\mathcal{O}ntoDLV$ well-suited for the development of industrial applications.

9

Conclusion

In this thesis work, we have presented $\mathcal{O}ntoDLP$, an extension of disjunctive logic programming with relevant constructs for advanced knowledge modeling, including classes, objects, (multiple) inheritance, intensional entities, lists, sets, extended types and meta-reasoning. We have described the syntax of $\mathcal{O}ntoDLP$ and shown its usage for ontology representation and reasoning.

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

In particular, ”OWL Atoms” allow for mixing knowledge coming from OWL into logical rules; thus, $\mathcal{O}ntoDLP$ can reason on top of an OWL ontology.

Importantly, we have provided a concrete implementation of the language: the $\mathcal{O}ntoDLV$ system. It combines powerful type-checking mechanism and flexible storing of ontologies for fast ontologies specification and error detection.$\mathcal{O}ntoDLV$ features also an Application Programming Interface (API), in order to ease exploiting of the system for solving problems and developing real-world applications based on $\mathcal{O}ntoDLP$. The system is built on top of DLV (a state-of-the art DLP system), and DLV^{DB} (a version of DLV featuring a fragment of DLP with direct execution of reasoning tasks on DBMS).

We proposed also *pragmatic* approach to the problem of interoperability between $\mathcal{O}ntoDLP$ and OWL. In particular, we designed and implemented in the $\mathcal{O}ntoDLV$ system two transformation facilities, which are able to *import* an OWL ontology in $\mathcal{O}ntoDLP$, and *export* an $\mathcal{O}ntoDLP$ specification in an OWL one. We shown also that *semantic equivalence* between the original OWL ontology and the obtained $\mathcal{O}ntoDLP$ ontology is guaranteed for a fragment of both languages.

The $\mathcal{O}ntoDLV$ system has already been exploited for the development of some advanced real-world applications. We close the section by describing a couple of them.

HıLεX system

The HıLεX system [56, 55, 57] is an advanced tool for semantic information-extraction from unstructured or semi-structured documents. Here, an $\mathcal{O}ntoDLP$ ontology is used to represent concepts of the documents domain, while a set of “semantic” regular expressions (HiLEx expressions) represent ways of writing a concept in a document. The extraction is achieved by rewriting such expressions in $\mathcal{O}ntoDLP$ (by exploiting modules and collection classes) and computing the answer sets of the obtained $\mathcal{O}ntoDLP$ specification.

The HɩLεX system has been successfully applied for the extraction of clinical data (stored in flat text format in Italian language) from an Electronic Medical Record (EMR).

Ontology driven agent environment (RAP platform)

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 $\mathcal{O}ntoDLP$ program.

In particular, application domain has been represented through a domain ontology. The main events are captured by an agent network (supported by a physical RFID network) and mapped into semantic model (domain ontology). $\mathcal{O}ntoDLV$ supports the agent network giving the ability to reason on the basis of captured events. On the basis of results, the agents are able to choose the right action as the logic consequence of the captured event.

Semantic Extraction and Adaptive Delivery of Multimedia Contents for the Cultural Assets

A mesh between the $\mathcal{O}ntoDLP$ ontology-based modeling system and the ***DISAS*** [2, 27, 60] adaptive system has been presented in [15]. $\mathcal{O}ntoDLP$ supports ontology-based contents modeling and extraction, while ***DISAS*** allows to deliver contents extracted and produced by $\mathcal{O}ntoDLV$ in an adaptive way, taking into account user's profile, network characteristics and kind of user terminal.

We envisioned a scenario composed by a set of cultural assets (e.g., museums, archeological sites, as well as points of interests on a cultural path) each one producing its own contents by using the $\mathcal{O}ntoDLP$ system. Each $\mathcal{O}ntoDLP$ instance is coupled to an instance of a ***DISAS*** server to form a site of the Content Network. Content extracted by the $\mathcal{O}ntoDLP$ server is passed to the ***DISAS*** server that adapts it taking into account user profile. User profiles are built by ***DISAS*** servers that monitor user activity and network bandwidth. In a centralized implementation, each ***DISAS*** server maintains the profiles of its visiting users needed to make adaptation, so when a user moves to another site his/her profile has to be rebuilt. In the paper we sketch a Peer-To-Peer (P2P) approach for the sharing of profile information among ***DISAS*** servers. When a user enters a new site, the ***DISAS*** server recovers his/her profile, if available, using the P2P sharing mechanism.

Part IV

Appendices

A

Disjunctive Logic Programs with Aggregates

Syntax

We assume that the reader is familiar with standard DLP; we refer to atoms, literals, rules, and programs of DLP, as *standard atoms, standard literals, standard rules*, and *standard programs,* respectively. Two literals are said to be complementary if they are of the form p and $\mathrm{not}p$ for some atom p. Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$. For further background, see [7, 30].

Set Terms.

A (DLP) *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\mathit{Vars} : \mathit{Conj}\}$, where Vars is a list of variables and Conj is a conjunction of standard atoms.[1] A *ground set* is a set of pairs of the form $\langle\bar{t} : \mathit{Conj}\rangle$, where $\bar{t}$ is a list of constants and Conj is a ground (variable free) conjunction of standard atoms.

Aggregate Functions.

An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol.* Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

Example A.1. (In the examples, we adopt the syntax of DLV to denote aggregates.) *Aggregate functions currently supported by the* DLV *system are:* #`count` *(number of terms),* #`sum` *(sum of non-negative integers),* #`times` *(product of positive integers),* #`min` *(minimum term, undefined for empty set),* #`max` *(maximum term, undefined for empty set)*[2].

[1] Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X \mid \exists Y s.t.\ a(X,Y), p(Y)\ \mathit{is\ true}\}$.

[2] The first two aggregates correspond, respectively, to the cardinality and weight constraint literals of Smodels.

Aggregate Literals.

An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

Example A.2. The following aggregate atoms in DLV *notation, where the latter contains a ground set and could be a ground instance of the former:*

$$\#\mathtt{max}\{Z : r(Z), a(Z, V)\} > Y$$
$$\#\mathtt{max}\{\langle 2 : r(2), a(2, k)\rangle, \langle 2 : r(2), a(2, c)\rangle\} > 1$$

An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, L is an *aggregate literal*.

DLP *Programs.*

A *(*DLP*) rule* $\mathcal{R}$ is a construct

$$a_1 \,\mathtt{v}\, \cdots \,\mathtt{v}\, a_n \text{ :- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m.$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, $n \geq 0$, and $m \geq k \geq 0$. The disjunction $a_1 \,\mathtt{v}\, \cdots \,\mathtt{v}\, a_n$ is referred to as the *head* of $\mathcal{R}$ while the conjunction $b_1, ..., b_k,$ not $b_{k+1}, ...,$ not b_m is the *body* of $\mathcal{R}$. We denote the set $\{a_1, \ldots, a_n$ of the head atoms by $H(\mathcal{R})$, and the set $\{b_1, ..., b_k, \text{ not } b_{k+1}, ..., \text{not } b_m\}$ of the body literals by $B(\mathcal{R})$.

A *(*DLP*) program* is a set of DLP rules. A *global* variable of a rule r is a variable appearing in a standard atom of r; all other variables are *local* variables.

Safety.

A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r appearing in a symbolic set $\{\mathit{Vars} : \mathit{Conj}\}$ appears in an atom of Conj; (iii) each guard of an aggregate atom of r is a constant or a global variable. A program $\mathcal{P}$ is safe if all $R \in \mathcal{P}$ are safe. In the following we assume that DLP programs are safe.

Stable Model Semantics

Universe and Base.

Given a DLP program $\mathcal{P}$, let $U_{\mathcal{P}}$denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$be the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to I (the set of integers).

Instantiation.

A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for S*. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of S* is the following ground set of pairs $inst(S)$:
$\{\langle\gamma(Vars) : \gamma(Conj)\rangle \mid \gamma \textit{ is a local substitution for } S\}$.[3]
A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example A.3. Consider the following program $\mathcal{P}_1$:

$$q(1) \,\mathtt{v}\, p(2,2). \qquad\qquad q(2) \,\mathtt{v}\, p(2,1).$$
$$t(X) \text{ :- } q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1.$$

The instantiation $Ground(\mathcal{P}_1)$ is the following:

$$q(1) \,\mathtt{v}\, p(2,2).t(1) \text{ :- } q(1), \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1.$$
$$q(2) \,\mathtt{v}\, p(2,1).t(2) \text{ :- } q(2), \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1.$$

Interpretations.

An *interpretation* for a DLP program $\mathcal{P}$ is a consistent set of standard ground atoms, that is $I \subseteq B_{\mathcal{P}}$. A positive literal A is true w.r.t. I if $A \in I$, is false otherwise. A negative literal $\mathrm{not}A$ i true w.r.t. I, if $A \notin A$, is false otherwise.

An interpretation also provides a meaning for aggregate literals.

Let I be an interpretation. A standard ground conjunction is true (resp. false) w.r.t I if all its literals are true. The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I. More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, ..., t_n : Conj\rangle \in S \wedge \; Conj \textit{ is true w.r.t. } I\,]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$).[4]

An instantiated aggregate atom $A = f(S) \prec k$ is *true w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\mathrm{not}A = \mathrm{not}f(S) \prec k$ is *true w.r.t.* I if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

Minimal Models.

Given an interpretation I, a rule r is *satisfied w.r.t. I* if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. An interpretation M is a *model* of a DLP program $\mathcal{P}$ if all $R \in Ground(\mathcal{P})$ are satisfied w.r.t. M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N \subset M$.

[3] Given a substitution σ and a DLP object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

[4] In this paper, we assume that the value of an aggregate function can be computed in time polynomial in the size of the input multiset.

Stable Models.

We now recall the generalization of the Gelfond-Lifschitz transformation to programs with aggregates from [25].

Definition A.4 ([25]). Given a ground DLP 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. I is a stable model of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

Example A.5. Consider the following two programs:

$$P_1 : \{p(a) \text{:-} \#\texttt{count}\{X : p(X)\} > 0.\}$$
$$P_2 : \{p(a) \text{:-} \#\texttt{count}\{X : p(X)\} < 1.\}$$

$Ground(P_1) = \{p(a) \text{:-} \#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$ *and*
$Ground(P_2) = \{p(a) \text{:-} \#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$,
and interpretation $I_1 = \{p(a)\}$, $I_2 = \emptyset$. *Then,* $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$, *and* $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ *hold.*

I_2 is the only stable model of P_1 (because I_1 is not a minimal model of $Ground(P_1)^{I_1}$), while P_2 admits no stable model (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$).

Note that any stable model A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

Summary of publications

- L. Gallucci, F. Ricca, R. Schindlauer, T. Dell′Armi, and N. Leone, OntoDLV: a DLP-based system for Enterprise Ontologies in Journal of Logic and Computation (*Elsevier*). (2008 *invited*)
- M. Ruffolo, M. Manna, L. Gallucci, N. Leone, and D. Saccà: A logic-based tool for semantic information extraction, JELIA, LNCS, vol. 4160, Springer, 2006 [57]
- M. Ruffolo, R. Curia, and L. Gallucci, Process management in health care: A system for preventing risks and medical errors, BPM 05 Proceedings, LNCS 3649, Springer, 2005 [54]
- M. Cannataro, L. Gallucci, A. Pietramala, P. Rullo, and P. Veltri Semantic extraction and adaptive delivery of multimedia contents for the cultural assets (UPGRADE 07 Proceedings) (ACM Press 2007) [15]
- L. Gallucci, M. Cannataro, and P. Veltri: Models and Technologies for Adaptive Web Portals, in Encyclopaedia of Portal Technology and Applications (A. Tatnall ed.) Vol. II (IGI Global 2007) [60]
- R. Curia, M. Ettorre, L. Gallucci, S. Iiritano, and P. Rullo, Textual document preprocessing and feature extraction in OLEX, in Data Mining VI: Data Mining, Text Mining and their Business Applications, WIT Transactions on Information and Communication Technologies, vol. 35, WIT Press, 2005. [18]
- R. Curia, L. Gallucci, and M. Ruffolo, Knowledge management in health care: An architectural framework for clinical process management systems, DEXA Workshops, IEEE Computer Society, 2005 [19]
- L. Gallucci, G. Grasso, N. Leone, and F. Ricca: Interoperability Mechanisms for Ontology Management Systems (CILC 07 Proceedings) [28]
- L. Gallucci, M. Cannataro, L. Palopoli, and P. Veltri: DISAS: a DISelect-based Middleware for Building Adaptive Systems (A3H 06 Proceedings) [27]
- M. Ruffolo, L. Gallucci, N. Leone, M. Manna, and D. Saccà, Towards a semantic information extraction approach from unstructured documents (SEBD 06 Proceedings) [55]
- T. Dell′Armi, L. Gallucci, N. Leone, F. Ricca, R. Schindlauer, and, OntoDLV: an ASP-based system for Enterprise Ontologies (ASP 07 Proceedings) [20]
- L. Gallucci and F. Ricca, Visual querying and application programming interface for an ASP-based ontology language (SEA07) [29]

References

1. *A logic for objects*, Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington, DC (M. Maier, ed.), 1986, pp. 6–26.
2. G. Acati, Mario Cannataro, E. Giampà, Marco Mastratisi, Luigi Palopoli, and Pasquale Rullo, *A software platform for services delivery in the public administration.*, CAiSE Workshops (2) (Jaelson Castro and Ernest Teniente, eds.), FEUP Edições, Porto, 2005, pp. 69–81.
3. José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski, *HiLog: A foundation for higher-order logic programming*, Journal of Logic Programming **15** (1993), no. 3, 187–230.
4. Grigoris Antoniou, Carlos Viegas Damsio, Benjamin Grosof, Ian Horrocks, Michael Kifer, Jan Maluszynski, and Peter F. Patel-Schneider, *Combining Rules and Ontologies. A survey.*, 2005.
5. Krzysztof R. Apt, Howard A. Blair, and Adrian Walker, *Towards a Theory of Declarative Knowledge*, Foundations of Deductive Databases and Logic Programming (Jack Minker, ed.), Morgan Kaufmann Publishers, Inc., Washington DC, 1988, pp. 89–148.
6. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (eds.), *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press, 2003.
7. Chitta Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.
8. Chitta Baral and Michael Gelfond, *Logic Programming and Knowledge Representation*, Journal of Logic Programming **19/20** (1994), 73–148.
9. Kristof Van Belleghem, Marc Denecker, and Danny De Schreye, *A strong correspondence between description logics and open logic programming.*, ICLP, 1997, pp. 346–360.
10. Rachel Ben-Eliyahu and Rina Dechter, *Propositional Semantics for Disjunctive Logic Programs*, Annals of Mathematics and Artificial Intelligence **12** (1994), 53–87.
11. Rachel Ben-Eliyahu-Zohary and Luigi Palopoli, *Reasoning with Minimal Models: Efficient Algorithms and Applications*, Artificial Intelligence **96** (1997), 421–449.
12. Alexander Borgida, *On the relative expressiveness of description logics and predicate logics.*, Artificial Intelligence **82** (1996), no. 1–2, 353–367.
13. Francesco Buccafurri, Nicola Leone, and Pasquale Rullo, *Enhancing Disjunctive Datalog by Constraints*, IEEE Transactions on Knowledge and Data Engineering **12** (2000), no. 5, 845–860.

14. Francesco Calimeri, Stefania Galizia, Massimo Ruffolo, and Pasquale Rullo, *Ontodlp: a logic formalism for knowledge representation.*, Answer Set Programming (Marina De Vos and Alessandro Provetti, eds.), CEUR Workshop Proceedings, vol. 78, CEUR-WS.org, 2003.
15. Mario Cannataro, Lorenzo Gallucci, Adriana Pietramala, Pasquale Rullo, and Pierangelo Veltri, *Semantic extraction and adaptive delivery of multimedia contents for the cultural assets*, UPGRADE '07: Proceedings of the second workshop on Use of P2P, GRID and agents for the development of content networks (New York, NY, USA), ACM, 2007, pp. 49–56.
16. Weidong Chen, Michael Kifer, and David Scott Warren, *Hilog: A foundation for higher-order logic programming*, Journal of Logic Programming **15** (1993), 187–230.
17. Weidong Chen and David Scott Warren, *C-logic of complex objects*, Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, ACM Press, 1989, pp. 369–378.
18. Rosario Curia, Mario Ettorre, Lorenzo Gallucci, Salvatore Iiritano, and Pasquale Rullo, *Textual document pre-processing and feature extraction in olex*, Data Mining VI: Data Mining, Text Mining and their Business Applications (Alessandro Zanasi, ed.), WIT Transactions on Information and Communication Technologies, vol. 35, WIT Press, 2005.
19. Rosario Curia, Lorenzo Gallucci, and Massimo Ruffolo, *Knowledge management in health care: An architectural framework for clinical process management systems*, DEXA Workshops, IEEE Computer Society, 2005, pp. 393–397.
20. Tina DellÁrmi, Lorenzo Gallucci, Nicola Leone, Francesco Ricca, and Roman Schindlauer, *Ontodlv: an asp-based system for enterprise ontologies*, Answer Set Programming (Stefania Costantini and Richard Watson, eds.), CEUR Workshop Proceedings, CEUR-WS.org, 2007.
21. T. Eiter and G. Gottlob, *On the Computational Cost of Disjunctive Logic Programming: Propositional Case*, Annals of Mathematics and Artificial Intelligence **15** (1995), no. 3/4, 289–323.
22. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits, *Combining Answer Set Programming with Description Logics for the Semantic Web*, Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, 2004, Extended Report RR-1843-03-13, Institut für Informationssysteme, TU Wien, 2003., pp. 141–151.
23. Thomas Eiter, Georg Gottlob, and Heikki Mannila, *Disjunctive Datalog*, ACM Transactions on Database Systems **22** (1997), no. 3, 364–418.
24. Thomas Eiter, Nicola Leone, and Domenico Saccá, *Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases*, Theoretical Computer Science **206** (1998), no. 1–2, 181–218.
25. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer, *Recursive aggregates in disjunctive logic programs: Semantics and complexity*, Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004) (José Júlio Alferes and João Leite, eds.), Lecture Notes in AI (LNAI), vol. 3229, Springer Verlag, September 2004, pp. 200–212.
26. Steve Freeman and Nat Pryce, *Evolving an embedded domain-specific language in java*, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM, 2006, pp. 855–865.
27. Lorenzo Gallucci, Mario Cannataro, Luigi Palopoli, and Pierangelo Veltri, *Disas: a diselect-based middleware for building adaptive systems*, Proceedings of Workshops held at the Fourth International Conference on Adaptive Hypermedia and Adaptive Web-Based

Systems (AH2006) (Rosa M. Carro Alexandra Cristea and Franca Garzotto, eds.), 2006, pp. 377–386.

28. Lorenzo Gallucci, Giovanni Grasso, Nicola Leone, and Francesco Ricca, *Interoperability mechanisms for ontology management systems*, CILC-07 (Alessandro Provetti, ed.), CEUR Workshop Proceedings, CEUR-WS.org, 2007.
29. Lorenzo Gallucci and Francesco Ricca, *Visual querying and application programming interface for an asp-based ontology language*, Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07) (M. De Vos and T. Schaub, eds.), 2007, pp. 56–70.
30. M. Gelfond and V. Lifschitz, *Classical Negation in Logic Programs and Disjunctive Databases*, New Generation Computing **9** (1991), 365–385.
31. Georg Gottlob, *Complexity and Expressive Power of Disjunctive Logic Programming*, Proceedings of the International Logic Programming Symposium (ILPS '94) (Ithaca NY) (M. Bruynooghe, ed.), MIT Press, 1994, pp. 23–42.
32. Georg Gottlob, Nicola Leone, and Helmut Veith, *Succinctness as a Source of Expression Complexity*, Annals of Pure and Applied Logic **97** (1999), no. 1–3, 231–260.
33. Sergio Greco, Nicola Leone, and Pasquale Rullo, *COMPLEX: An Object-Oriented Logic Programming System*, IEEE Transactions on Knowledge and Data Engineering **4** (1992), no. 4.
34. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, *Description logic programs: Combining logic programs with description logics*, Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 2003, pp. 48–57.
35. Stijn Heymans and Dirk Vermeir, *Integrating semantic web reasoning and answer set programming.*, Answer Set Programming, 2003.
36. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean, *Swrl: A semantic web rule language combining owl and ruleml*, May 2004, W3C Member Submission. `http://www.w3.org/Submission/SWRL/`.
37. Ullrich Hustadt, Boris Motik, and Ulrike Sattler, *Reducing shiq-description logic to disjunctive datalog programs*, Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, 2004, pp. 152–162.
38. David S. Johnson, *A Catalog of Complexity Classes*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), vol. A, Elsevier Science Pub., 1990.
39. Michael Kifer, Georg Lausen, and James Wu, *Logical foundations of object-oriented and frame-based languages.*, Journal of the ACM **42** (1995), no. 4, 741–843.
40. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello, *The DLV System for Knowledge Representation and Reasoning*, ACM Transactions on Computational Logic **7** (2006), no. 3, 499–562.
41. Nicola Leone and Francesco Ricca, *Ontodlv: An object-oriented disjunctive logic programming system*, Convegno Italiano di Logica Computazionale (CILC 2006), 26-27 June 2006, Bari., 2006.
42. Nicola Leone, Pasquale Rullo, and Francesco Scarcello, *Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation*, Information and Computation **135** (1997), no. 2, 69–112.
43. Mengchi Liu, Gillian Dobbie, and Tok Wang Ling, *logical foundation for deductive object-oriented databases.*, ACM Trans. Database Syst. **27** (2002), 117–151.
44. Jorge Lobo, Jack Minker, and Arcot Rajasekar, *Foundations of Disjunctive Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1992.

45. W. Marek and V.S. Subrahmanian, *The Relationship between Logic Program Semantics and Non-Monotonic Reasoning*, Proceedings of the 6th International Conference on Logic Programming – ICLP'89, MIT Press, 1989, pp. 600–617.
46. Boris Motik, Ulrike Sattler, and Rudi Studer, *Query answering for owl-dl with rules.*, J. Web Sem. **3** (2005), no. 1, 41–60.
47. Natasha Noy and Alan Rector, *Defining N-ary Relations on the Semantic Web. W3C Working Group Note*, 2006, `http://www.w3.org/TR/swbp-n-aryRelations/`.
48. Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.
49. Teodor C. Przymusinski, *Stable Semantics for Disjunctive Programs*, New Generation Computing **9** (1991), 401–424.
50. Theodor C. Przymusinski, *On the Declarative Semantics of Deductive Databases and Logic Programs*, Foundations of Deductive Databases and Logic Programming (Jack Minker, ed.), Morgan Kaufmann Publishers, Inc., 1988, pp. 193–216.
51. Francesco Ricca and Nicola Leone, *Disjunctive Logic Programming with Types and Objects: The DLV+ System*, Journal of Applied Logics **5** (2007), 545–573.
52. Riccardo Rosati, *Dl+log: Tight integration of description logics and disjunctive datalog.*, KR, 2006, pp. 68–78.
53. ______, *Integrating ontologies and rules: Semantic and computational issues.*, Reasoning Web, 2006, pp. 128–151.
54. Massimo Ruffolo, Rosario Curia, and Lorenzo Gallucci, *Process management in health care: A system for preventing risks and medical errors*, Business Process Management (Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, eds.), vol. 3649, Springer, 2005, pp. 334–343.
55. Massimo Ruffolo, Lorenzo Gallucci, Nicola Leone, Marco Manna, and Domenico Saccà, *Towards a semantic information extraction approach from unstructured documents*, SEBD (Valeria De Antonellis, Claudia Diamantini, and Paolo Tiberio, eds.), 2006, pp. 167–174.
56. Massimo Ruffolo, Nicole Leone, Marco Manna, Domenico Sacca', and Antonio Zavatto, *Exploiting ASP for Semantic Information Extraction*, Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation (Bath, UK) (Marina de Vos and Alessandro Provetti, eds.), July 2005.
57. Massimo Ruffolo, Marco Manna, Lorenzo Gallucci, Nicola Leone, and Domenico Saccà, *A logic-based tool for semantic information extraction*, JELIA (Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, eds.), Lecture Notes in Computer Science, vol. 4160, Springer, 2006, pp. 506–510.
58. Michael K. Smith, Chris Welty, and Deborah L. McGuinness, *OWL web ontology language guide. W3C Candidate Recommendation*, 2003, `http://www.w3.org/TR/owl-guide/`.
59. Terrance Swift, *Deduction in ontologies via asp.*, LPNMR, 2004, pp. 275–288.
60. Arthur Tatnall, *Encyclopaedia of portal technology and applications*, vol. II, ch. Models and Technologies for Adaptive Web Portals, pp. 615–623, IGI Global, 2007, Lorenzo Gallucci, Mario Cannataro, and Pierangelo Veltri.
61. Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta, *Experimenting with recursive queries in database and logic programming systems*, Journal of the Theory and Practice of Logic Programming (2008), (*forthcoming*).
62. K. Permanente T.Paul, V. Biron and A. Malhotra, *XML Schema Part 2: Datatypes Second Edition.*, 2004, `http://www.w3.org/TR/xmlschema-2/`.
63. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf, *The Well-Founded Semantics for General Logic Programs*, Journal of the ACM **38** (1991), no. 3, 620–650.

64. W3C, *The resource description framework.*, 2006, `http://www.w3.org/RDF/./`.
65. Guizhen Yang, Michael Kifer, and Chang Zhao, *"flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web."*, CoopIS/DOA/ODBASE (Robert Meersman, Zahir Tari, and Douglas C. Schmidt, eds.), 2003, pp. 671–688.