OntoDLV: An Object-Oriented Disjunctive Logic Programming System

Nicola Leone, Francesco Ricca

Department of Mathematics, University of Calabria, Rende (CS) 87036, Italy

Abstract. The paper presents OntoDLV a system based on an extension of Disjunctive Logic Programming (DLP) which combines the expressive power of DLP with the modeling capabilities of the object-oriented languages. In particular, the OntoDLV language supports the most important object-oriented constructs including classes, objects, (multiple) inheritance, and types.
OntoDLV is built on top of DLV (a state-of-the art DLP system), and provides a graphical user interface that allows to specify, update, browse, query, and reason on knowledge bases. Two strong points of the system are the powerful type-checking mechanism, and the advanced interface for visual querying.

1 Introduction

Disjunctive Logic Programming under the Stable Model Semantics [1] (DLP) is an expressive logic programming language, which has been proposed in the area of nonmonotonic reasoning and logic programming.

The high expressive power of DLP [2] can be profitably exploited when one has to deal with problems of high complexity, and this makes DLP an advanced formalism for Knowledge Representation and commonsense Reasoning (KR&R)[1].

Moreover, the availability of a couple of efficient DLP systems, like DLV [3], GnT [4] and, more recently, the disjunctive version of Cmodels [5] make DLP a powerful tool for developing advanced knowledge-based applications [6, 7].

Despite its high expressiveness, there are several problems that DLP cannot encode in a natural way. For instance, it misses constructs for representing complex real-world entities[8], like classes, objects, compound objects, and taxonomies. Moreover, DLP systems are missing tools for supporting the programmers, like type-checkers and easy-to-use graphical environments, to manage the large and complex domains to be dealt with in real-world applications.

The recent applications of DLP in the emerging areas of Knowledge Management (KM) and Information Integration [9] have evidenced the practical need to enhance DLP languages and systems to overcome the above drawbacks.

This paper describes the OntoDLV system, a first step towards overcoming the above limitations. It is a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The OntoDLV system

allows for the development of complex applications and allows one to perform advanced reasoning tasks in a user friendly visual environment. The OntoDLV system seamlessly integrates the DLV system [3] exploiting the power of a stable and efficient DLP solver.

A strong point of the system is its powerful language, extending DLP by object-oriented features. In particular, the language includes, besides the concept of **relation**, the object-oriented notions of **class**, **object** (class instance), **object-identity**, **complex object**, **(multiple) inheritance**, and the concept of modular programming by means of **reasoning modules**.

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

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

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type)[1].

Importantly, OntoDLP supports two kind of classes and relations: *(base) classes and (base) relations*, corresponding to basic facts (that can be stored in a database); and *derived classes and derived relations* corresponding to facts that can be inferred by logic programs.

As in DLP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). In this way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, OntoDLP logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

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

Moreover, OntoDLV offers several important facilities driving the development of both the knowledge base and the reasoning modules. Using OntoDLV, developers and domain experts can create, edit, navigate and query object-oriented knowledge bases by an easy-to-use **visual environment**, enriched by a graphic **query interface** à la QBE.

In short, the contribution of the paper is twofold:

– We describe a new language, named OntoDLP, for Knowledge Representation and Reasoning, extending DLP with relevant constructs of the object-

[1] Note that, unlike objects, relation instances are not identified by means of oid's.

oriented paradigm, like Classes, Types, Objects and Inheritance. We illustrate syntax, semantics, and knowledge modeling features of OntoDLP by examples.
– We design and implement a system supporting OntoDLP, named OntoDLV. The system offers all features of OntoDLP, it provides a user friendly Graphical User Interface, and a powerful type checking mechanism, which supports the user in a fast development of error-free ontologies. OntoDLV is endowed also with a visual query interface, allowing to combine navigation and querying for powerful information extraction.

The system is already employed in practice in a couple of applications for text classification and information extraction (see Section 5).

2 The OntoDLP Language

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 taxonomy, relations, and axioms is commonly called an ontology.

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

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

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

It is worth noting that OntoDLP is actually an extension of Disjunctive Logic Programming (DLP)[2], which has been enriched by concepts from the object-oriented paradigm; from now on, we assume the reader to be familiar with DLP syntax and semantics. For a comprehensive introduction to DLP the reader can refer to [1, 11].

2.1 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.

[2] We actually use DLP with aggregates functions [10].

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 OntoDLP as follows:

class *person.* ***class*** *animal.* ***class*** *food.* ***class*** *place.*

The simplest way to declare a class is, hence, to specify the class name, preceded by the keyword **class**. 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 OntoDLP 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 which 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, OntoDLP features the built-in classes *string* and *integer*, respectively representing the class of all alphanumeric strings and the class of non-negative numbers.

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

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

Note that this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects. 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 OntoDLP.

[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).

2.2 Objects

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

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

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, 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 the *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. This instance can be declared as follows:

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

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

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

2.3 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 OntoDLP, 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 OntoDLP as follows:

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

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

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

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

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

al:student(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome, code:"100", school:"Cambridge", tutor:hanna).
jack:employee(name:"Jack", age:54, father:jim, mother:mary, birthplace:rome, salary:1000, skill:"Java programmer", company:"SUN", tutor:betty).

They are automatically considered also instances of person as follows:

al:person(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome).
jack:person(name:"Jack", age:54, father:jim, mother:mary, birthplace:rome).

Note that it is not necessary to assert the above two instances, both *al* and *jack* are automatically considered instances of *person*.

In OntoDLP 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:

class *stud-emp* ***isa*** {student, employee}(*workload:integer*).

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* [5].

[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 criterion adopted in OntoDLP for solving type conflicts due to multiple inheritance was introduced with the COMPLEX language, see [12]

We complete the description of inheritance recalling that there is also another built-in class in OntoDLP, which is the superclass of all the other classes and is called *object* (or $\top$).

2.4 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 OntoDLP by means of *Relations*.

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

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

relation *friend(pers1:person, pers2:person).*
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:"two years").

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 oid).

2.5 Derived Classes and Derived Relations

The notions of class and relation introduced above correspond, from a data-base point of view, to the the *extensional* part of the OntoDLP language. In fact, their instances and tuples are defined explicitly asserting some logic facts. However, there are many cases in which some property or some class of individuals can be "derived" (or inferred) from the information already stated in an ontology. In the database world, the *views* allows to specify this this kind of knowledge which is usually called "intensional". In OntoDLP there are two different "intensional" constructs: *derived classes* and *derived relations*.

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

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

Note that, in this case the instances of the class *youngAndShy* are "borrowed" from the class person, and are inferred by using a logic rule.

In general, the derived classes neither have proper instances nor proper oid's (they group already defined objects), and cannot be organized in taxonomies by using the *isa* relation.

In an analogous way we specify *derived relations*. As an example, consider the derived relations *ancestor*:

relation *ancestor(anc:person,disc:person).* {
$ancestor(anc : A, disc : X) :- X : person(father : A).$
$ancestor(anc : A, disc : X) :- X : person(mother : A).$
$ancestor(anc : A, disc : X) :- X : person(father : Y),$
$\qquad ancestor(anc : A, disc : Y).$
$ancestor(anc : A, disc : X) :- X : person(mother : Y),$
$\qquad ancestor(anc : A, disc : Y).$ }

The above definition states that A is an ancestor of X if: (i) A is either the father of X or the mother of X; or (ii) A is either an ancestor of the father of X or an ancestor of the mother of X.

Note that this definition is recursive (it is a kind of transitive closure).

In general, *derived classes* and *derived relations* are both more natural and more expressive than relational database views, in-fact they allow the use of the navigational style of object-oriented programming combined with a more powerful language that allows recursion and negation as failure[6].

2.6 Axioms and Consistency

The structural representation of a knowledge domain is obtained in OntoDLP by specifying classes and relations. In general, this information is not enough to obtain a correct description of the domain. Often, it is necessary to impose constraints asserting additional conditions which hold in the domain.

These assertions are modeled in OntoDLP 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) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we assert the following axioms:

[6] It is worth noting that the programs which define derived classes and derived relations must be normal and stratified (see e.g. [3]).

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

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

Note that OntoDLP axioms do not derive new knowledge, but they are only used to model sentences that must be always true, like integrity constraints[7].

Observe that axioms are syntactically distinguished by constraints because they are declared by using the symbol ::– instead of :– .

The usefulness of axioms is rather clear, as they allows one to enforce the consistency of the specified ontology.

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

2.7 Reasoning modules

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

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

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

We now show an example demonstrating that the reasoning power of OntoDLP can be exploited for solving complex real-world problems.

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

- the project team has to be constituted of a fixed number of employees;
- 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.

[7] The difference between axioms and constraints is that axioms are specifically conceived to work with the knowledge contained in an ontology, while constraints are conceived in order to enforce some property in a logic program.

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:

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

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

$\boldsymbol{module}(teamBuilding)\{$

$(r) \qquad inTeam(E,P) \vee outTeam(E,P) :- E : employee(), P : project().$

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

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

2.8 Querying

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

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

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

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

3 The OntoDLV System

OntoDLV is a complete tool that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies. We refrain describing the implementation details of OntoDLV in this paper[8]. Rather, we illustrate the overall OntoDLV architecture, and present the main features of the system by describing the main components of the graphical user interface of OntoDLV.

3.1 System Architecture

The system architecture of OntoDLV, depicted in Figure 1, is composed of eight modules, namely, GUI, Parser, Data Handler, Type Checker, Intelligent Rewriter, Output Handler and Message Handler, DLV, and two libraries: Lucene and, JGraph.

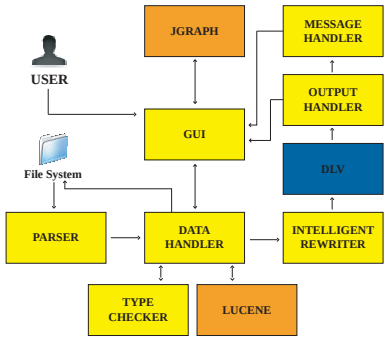

Fig. 1. The OntoDLV architecture

The user exploits the system through an easy-to-use visual environment called GUI (Graphical User Interface). The GUI combines a number of specialized visual tools for authoring, browsing and querying a OntoDLP ontology. In particular, the GUI features a graph-based ontology viewer and a graphical query environment, which are based on JGraph, an open-source library.

The Parser has the job to analyze and load the content of a OntoDLP text file in the data structures supplied by the Data Handler. The Data Handler provides all the methods needed to access and manipulate the ontology components. In particular, data indexing and full-text search are based on the open-source library Lucene. The admissibility of an ontology is ensured by the Type Checker module which implements a number of type checking routines. The Intelligent Rewriter module translates OntoDLP ontologies, reasoning modules and queries to an equivalent Disjunctive Logic Program which runs on the DLV system. The Intelligent Rewriter features a number of optimization and caching techniques in order to reduce the time used by interacting with DLV. Reasoning results

[8] For a better description of the implementation and for an in-depth specification of the rewriting procedure see [15].

and possible error messages are handled by the Output Handler and by the Message Handler modules respectively, and are displayed by the user interface accordingly.

3.2 Implementation and Usage

The OntoDLV system has been implemented in Java and is based on an efficient and optimized implementation of the rewriting module. Moreover, the OntoDLV system exploits the DLV system, a state-of-the-art DLP solver that has been shown to perform efficiently on both hard and "easy" (having polynomial complexity) problems[9].

The DLV system is a highly portable software written in ISO C++, available for various operating systems (UNIX, Mac OSX and Windows). Thus, the OntoDLV system runs under a variety of operating systems.

The OntoDLV system was designed to be simple for a novice to understand and use, and powerful enough to support experienced users.

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

The class/subclass hierarchy is displayed both in an indented text and a graph-based form.

The user can browse the ontology by double-clicking the items in the panels. The structure of each ontology entity (classes, relations, and instances) can be displayed in the middle of the screen by switching among several tabbed panels.

In the editing phase, the user enters the domain information by filling in the blanks of intuitive forms and selecting items from lists (exploiting an simple mechanism based on drag-and-drop). An up-to-date list of messages informs the user about the occurrence of errors (e.g. type checking messages, etc.) in the ontology under development. When the user clicks on an error message item the system promptly shows the entity involved in it.

Reasoning and querying can be performed by selecting the appropriate panel. The interface also allows the reasoning modality (both brave reasoning and cautious reasoning are supported) to be selected, and the reasoning modules needed to solve the specified reasoning task to be enabled/disabled. Importantly, queries can also be created by exploiting both a text-editing tool and a visual querying interface à la QBE which allows one to write queries without wondering about the syntax in a drag-and-drop-based environment. A sort of "reverse-engineering" procedure allows to smoothly switch between the text editing and the visual editing environment.

Finally, query results are presented to the user in an appealing way, while details about the interaction with DLV are hidden by the system.

The OntoDLV system, together with the system manual describing all the features available, can be downloaded at *http://www.mat.unical.it/ontodlv*.

[9] This feature is crucial for the implementation of the OntoDLV system, in fact ontologies are translated in an equivalent DLP program which is solved by DLV in polynomial time (under data complexity)

4 Related Work

A number of languages and systems somehow related to OntoDLP have been proposed in the literature. The most closely related system is COMPLEX [12], supporting the Complex-Datalog language, an extension of (non-disjunctive) Datalog with some concepts from the object-oriented paradigm. OntoDLV and COMPLEX share a similar object-oriented model, however the language of the latter is less expressive than OntoDLP. In fact, COMPLEX supports normal (non-disjunctive) stratified programs only (its expressive power is confined to P), which are strictly less expressive than OntoDLP language expressing even Σ_2^P-complete properties [2].

Another popular logic-based object-oriented language is F-Logic [13], implemented in the Flora-2 system [14], 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. Comparing OntoDLP with F-Logic, we note that the latter has a richer set of object oriented features (e.g. class methods, and multi-valued attributes), but it misses some important constructs of OntoDLP like disjunctive rules, which increase the knowledge modeling ability of the language. Concerning system-related aspects, an important advantage of OntoDLV (w.r.t. Flora-2) is the presence of a graphical development environment, which simplifies the interaction with OntoDLV for both the end user and the knowledge engineer.

A couple of other formalisms for specifying ontologies have been recently proposed by W3C, namely, RDF/RDFS and OWL. The Resource Description Framework (RDF) [16] 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). Basically, RDF(S) allows for expressing knowledge about the resources (identified via URI), and features a rich data-type library (richer than OntoDLP), but, unlike OntoDLP, 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)[17] 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)[17]. The largest decidable subset of OWL, called OWL-DL, coincides, basically, with $SHOIN(D)$, an expressive Description Logic (DL)[18]. OWL is based on classical logic (there is a direct mapping from $SHOIN$ to First Order Logic (FOL)) and, consequently, is quite different form OntoDLP, which is based on DLP. Compared to OntoDLP, OWL misses, for instance, *default negation*, *nonmonotonic disjunction*, and *inference rules*.

In sum, the strong point of OntoDLP, w.r.t. to other ontology representation languages, is the natural way in which it combines the most common ontology definition constructs with a powerful logic programming language, including rules, nonmonotonic disjunction, and default negation.

5 Conclusion

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

Importantly, we have provided also a concrete implementation of OntoDLP: the OntoDLV system. OntoDLV is built on top of DLV (the state-of-the art DLP system). It implements all features of OntoDLP, it also provides an advanced visual-interface, and a powerful type-checking mechanism, supporting the user for fast ontologies specification and errors detection.

The OntoDLV system is a valid support for the development of knowledge-based applications. Indeed, even if OntoDLP has been released very recently, it is already employed, playing a central role, in a couple of advanced applications for information extraction and text classification: HiLEx [19] and OLEX [21].

Ongoing work concerns the enhancement of OntoDLV by extending its language with new features such as optional and multi-valued attributes, a more powerful forms of both intentional classes, and reasoning modules.

Acknowledgements

This work was partially supported by M.I.U.R. under projects "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione", and "ONTO-DLV: Un ambiente basato sulla Programmazione Logica Disgiuntiva per il trattamento di Ontologie" n.2521.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2005) To appear.
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923

5. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
7. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge, Massachusetts (1992)
8. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
9. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
10. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003, Acapulco, Mexico,(2003) 847–852
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
12. Greco, S., Leone, N., Rullo, P.: COMPLEX: An Object-Oriented Logic Programming System. IEEE TKDE **4** (1992)
13. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. JACM **42** (1995) 741–843
14. Yang, G., Kifer, M., Zhao, C.: "flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web.". In: CoopIS/DOA/ODBASE. (2003) 671–688
15. Ricca, F., Leone, N.: Disjunctive Logic Programming with Types and Objects: The DLV$^+$ System. KBS Research Reports INFSYS RR-1843-05-10 Institut für Informationssysteme Technische Universität Wien - Austria
16. W3C: The resource description framework. (2006) http://www.w3.org/RDF/./.
17. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) http://www.w3.org/TR/owl-guide/.
18. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. CUP (2003)
19. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
20. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387
21. Curia, R., Ettorre, M., Iiritano, S., Rullo, P.: Textual Document Per-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)
22. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424

A Disjunctive Logic Programming

In this section, we provide a brief introduction to the syntax and semantics of Disjunctive Logic Programming; for further background see [11, 1].

Syntax. A *disjunctive rule* R is a formula:

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atomsand $n \geq 0$, $m \geq k \geq 0$. A literal is either an atom a or its default negation $\texttt{not} \ a$. Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\texttt{not} \ b_{k+1}, ..., \texttt{not} \ b_m\}$ the set of positive and negative body literals, resp., and $B(r) = B^+(r) \cup B^-(r)$. the set of body literals.

A rule r with $B^-(r) = \emptyset$ is called *positive*; a rule with $H(r) = \emptyset$ is referred to as *integrity constraint*. If the body is empty we usually omit the $:-$ sign.

A *disjunctive logic program* $\mathcal{P}$ is a finite set of rules; $\mathcal{P}$ is a *positive* program if all rules in $\mathcal{P}$ are positive (i.e., `not` -free). An object (atom, rule, etc.) containing no variables is called *ground* or *propositional*.

Semantics. The semantics of a disjunctive logic program is given by its stable models [22], 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 which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_\mathcal{P}$.

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

For every program $\mathcal{P}$, we define its stable models using its ground instantiation $\mathcal{P}$ 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 $\texttt{not} \ \ell$ 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. Interpretation I is *total* if, for each atom A in $B_\mathcal{P}$, either A or $\texttt{not} \ .A$ 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 $R \in \mathcal{P}$, 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 $\mathcal{P}$.

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 $R \in \mathcal{P}$ 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 $\mathcal{P}^X$.