OntoDLP: a Logic Formalism for Knowledge Representation

F. Calimeri[2], S. Galizia[2], M. Ruffolo[1,3], P. Rullo[1,2]

[1]*Exeura s.r.l. Via P. Bucci 87030 Rende (CS), Italy*
{ruffolo,rullo}@exeura.it

[2]Università della Calabria – Dip. Di Matematica, Via P. Bucci, 87030 Rende (CS), Italy
{calimeri,galizia,rullo}@unical.it

[3]*ICAR-CNR, Via P. Bucci, 87030 Rende (CS), Italy*
ruffolo@icar.cnr.it

Abstract. This paper provides an introduction to knowledge representation using OntoDLP, a formalism which combines the full computational power of Disjunctive Logic Programming (DLP) with suitable abstraction mechanisms for the representation of complex objects and default reasoning. The paper does not provide a formal definition of the language, rather it is intended as an informal presentation of its main features.

1 Introduction

Disjunctive Logic Programming (DLP) is an extension of Datalog where disjunction is allowed in the rules' heads. It is nowadays widely recognized as a valuable tool for knowledge representation and reasoning [6, 7]. An important merit of DLP over normal (i.e., disjunction-free) logic programming is its capability to model incomplete knowledge [4, 15]. Much research work has been done on the semantics of DLP and several alternative semantics have been proposed (see [7, 9, 8] for comprehensive surveys). The most widely accepted semantics is the extension of the stable model semantics of Gelfond-Lifschitz [9] to disjunctive deductive databases [9, 8]. Stable model semantics for DLP has a very high expressive power as they allow to capture the complexity class $\sum_2^P$ (i.e., every property which is decidable in non-deterministic polynomial time with an oracle in NP is expressible). An interesting extension of DLP by strong and weak constraints has been proposed in [BUC97, 7], and others are currently object of research.

A number of large scale systems, capable of handling thousands of rules, have been implemented in the last few years: DLV developed by Leone's group [7, 15], GnT/SModels developed by Niemelä and Simons [17], XSB developed by Warren et al. [19], and DeReS developed by Marek and Truszczynski[4]. While XSB and

DeReS support only fragments (i.e., sublanguages) of DLP, DLV and GnT/SModels support full DLP (in particular, DLP supports also strong and weak constraints [2]). The high expressiveness of DLP suggests that these systems have high potential for exploitation in the area of knowledge management. Unfortunately, DLP as such lacks of suitable abstraction mechanisms needed for the representation and the manipulation of complex application domains.

Complex-Datalog [11] is an extension of Datalog which includes a number of object-oriented constructs, namely, object identity, complex schemes, multiple inheritance. The most salient feature of the language is the way it combines the computational power of Datalog with the structural complexity of nested relation and object data models.

Ordered Logic [14, 16] is a non-monotonic reasoning formalism providing support for default reasoning. The basic mechanism of Ordered Logic (OL for short) comes from the utilization of true (or classical) negation (i.e., the possibility of explicitly stating the falsity of atoms) in the context of inheritance hierarchies (which assign different levels of reliability to rules). An OL program is a set of components organized into a hierarchical structure. Each component consists of a set of rules which may have negative heads. Like in the object-oriented approach, properties (rules) defined for the ``higher'' components in the hierarchy flow down to the ``lower'' ones. Thus, contradicting conclusions may be drawn. A stable model semantics for OL programs has been proposed in [16].

In this paper we present OntoDLP, a logic based formalism for ontology specification which extends DLP with the abstraction mechanisms of Complex-Datalog, the default reasoning capabilities of OL and other suitable abstraction mechanisms, such as set attributes, to represent complex knowledge. Roughly speaking, OntoDLP supports the notions of concept, relation and axiom. Concepts are either classes or instances of the world being modelled. Each concept can be characterized by a number of attributes and properties (the latter asserting general facts about concepts). Attribute values, possibly of set type, can be derived through logical rules. Concept instances can be explicitly stated or inferred by their definitions.

Concepts can be partially ordered by four types of built-in relations: *Isa*, *InstanceOf*, *PartOf* and *MoreReliableThan*, the latter used to express the comparative "credibility" of concepts. These relations can be exploited to generate taxonomic structures among concepts. When a property is defined for a concept, it holds for all its sub-concepts, unless overwritten by more specific properties. Multiple inheritance is supported.

Other kinds of (non-taxonomic) relations can be specified as well to relate concepts to each other within an ontology.

Axioms are used to express the semantics of both concepts and relations. An axiom is either a (possibly disjunctive) logic rule, or a strong constraint (used to state inviolable conditions) or a weak constraint (used to express desiderata). Rules may

have negative heads (true negation); negation by failure (in rule bodies) is also allowed. Axioms can be used for several different purposes, such as

- defining the properties of concepts
- defining derived concepts and relations
- specifying the cardinality constraints of relations
- stating the algebraic properties of relations (like symmetry, transitivity, etc.), etc.

The semantics of OntoDLP relies on the notion of stable model and uses a three-valued logic where each fact can be either true, false or undefined (*maybe* fact); as we will se, the proposed semantics is very intuitive for the purpose of modeling inheritance-based reasoning tasks.

OntoDLP is currently being implemented as a front-end of the disjunctive logic programming system DLV.

2 Concepts, relations and axioms

In this section we build a simple ontology that represents people, places they leave in and hobbies they enjoy.

2.1 Concepts

We start by defining the concepts we are modelling:

- **place**
- **person** has attributes
  name: string
  age: integer
  father: person
  spouse: person
  lives: place
  end;
- **hobby** has attributes
  name: string
  end;
- **city** ISA {place}, PartOf {country}, has attributes
  name: string
  people: integer
  end;
- **country** ISA {place},has attributes
  name: string
  capital: city
  end;

As we can see, all concepts above, except for **place**, are characterized by a number of attributes of different types; for instance, *person* owns several attributes, some of which of simple type, namely, *name* and *age,* and other of class type, namely, *father* and *spouse* of type *person*, and *lives* of type *place.* The concept **city** is defined as being a **place** (**city** is a *subclass* of **place**) and a part of a **country**. Likewise, **country** is a **place**.

We point out that, besides explicit attributes, each concept owns a special hidden attribute, that we call *self*, used to assign a unique, immutable identity to each instance. As an example, a possible instance (or individual) of the concept *person* is the following:

Person(p1, Mario, 34, p3, p5, c2)

where *p1* is the unique object identifier of the instance (it is the value of the attribute *self*), *p3* and *p5* are the object identifiers of the father and the spouse of *p1*, respectively, and *c2* is the object identifier of the city where *p1* lives (note that the values of the attribute *lives*, of type *place*, are either cities or countries).

2.2 Relations

Concepts can be related through either taxonomic or non-taxonomic relations; for example, the concept **Country** above is related to **Place** through the ISA relation and **City** to **Country** through the PART-OF relation.

Next we define two non-taxonomic relations, namely, *lives* and *enjoys*:

- lives (psn: person, plc: place)
- enjoys (psn: person, hby: hobby)

So we can say that a person lives in a place (a city or a country) and that people enjoy hobbies.

2.3 Axioms

In general, axioms are used to either specify constraints, or define properties, or provide an intensional, declarative definition of classes, relations and attributes.

Constraints

To start, let us assume that *place* is a concept for which instances are not allowed; we express this condition by the constraint

← Place(self:X)

expressing that, for each X, *Place(self:X)* must be false.

Another condition that we want to enforce in our ontology is the SYMMETRY of the marriage relation (expressed through the attribute *spouse* of *person*) -- i.e., it cannot happen that a person X has spouse Y, Y has spouse Z and X and Z are different individuals:

← person(self:X, spouse:Y), person(self:Y, spouse:Z), X <> Z

Finally, the following axiom is used to state the CARDINALITY constraint on the relation *lives* according to which each person lives in *exactly* one place:

← lives(person:X, place:Y), lives(person:X, place:Z), Z <> Y

Besides the above explicit constraints, there are some that are implicit, such as:

- the ISA relation is a PARTIAL ORDER (as any other built-in relation):
  - Path(X,Y) ← ISA(X,Y)
  - Path(X,Y) ← ISA(X,Z), Path(Z,X)
  - ← Path(X,X)

- a concept cannot be both a class and an instance

  ← class(X), instance(X)

- only classes can appear in the ISA relation

  ← ISA(X,Y), Instance(X)
  ← ISA(X,Y), Instance(Y)

Properties

Properties are used to assert general facts about concepts; for an instance, we may want to express that persons are mortal:

Person has properties
{ mortal()← }

As we will see in Section 3, properties hold for all instances, subclasses and PART-OF related classes, unless explicitly negated.

Derived Classes and Relations

A DERIVED CLASS or RELATION is one that is intensionally defined by a suitable set of logic rules. Next we report some examples.

```
Man() ISA person
{
  Man (self: X) ← person(self: X, sex:Y), Y=”male”
 }

Woman() ISA person
{
  Woman (self: X) ← person(self: X, sex:Y), Y=”female”
 }
```

Here, each of the above classes is defined by a logic rule that provides a formal definition of its meaning and is used to automatically infer its instances. We further express that **Person** is a GENERALIZATION of both **Man** and **Woman;** this generalization must be TOTAL (there is no person who is neither a man nor a woman) and DISJOINT (a person is not both a man and a woman):

```
← Person(self:X), not Man(self:X), not Woman(self:X).

← Man(X), Female(X)
```

As another example of derived class, consider the following definition of “cheerful person” as a person who enjoys at least three hobbies:

```
Cheerful-Person (numOfHobbies: integer) ISA {person}

 {
  Busy-Person (self: X, numOfHobbies: Z) ← person(self: X),
                                          %count(Y:enjoys(X,Y)}  =  Z,
Z>=3
 }
```

Note that the body of the above rule contains a special predicate (AGGREGATE predicate) used to count the number of hobbies enjoyed by person X.

As an example of derived relation, consider the following specification:

```
father_in_law (son: person, father:person)

    {
        father_in_law(X,Y) ←  person(self:X, spouse: person(father:Y))
    }
```

Here the definition consists of a unique rule stating that if Y is the father of the spouse of X then Y is the father in law of X. We point out as the structure *person(father:Y)* (called “class term” in Complex-Datalog) allows us to declaratively navigate the instances of person.

It is easy to see how we can define (derived) classes (or relations) as UNION, DIFFERENCE, INTERSECTION of other classes (or relations); to see an example, consider the classes

- **Student** ISA Person has attributes
  registration#: string
  enrolled: faculty
  end;

- **Worker** ISA Person has attributes
  salary: integer
  company: string
  end;

from which we derive the class Student-Worker as their INTERSECTION:

Student-Worker ISA { Student, Worker}

{ Student-Worker(self:X) ← Student(self:X), Worker(self:X) }.

Notice that, although no specific attribute is explicitly stated, **Student-Worker** inherits all the attributes of both **Student** and **Worker** (that, in turn, inherit those of **Person**).

Finally, we use DISJUNCTION to provide the definition of the derived class **MinMarried** whose set S of instances is the minimal set of persons such that, for each two persons who are married, at least one in S:

MinMarried() ISA person

{ MinMarried(X) V not_MinMarried(X) ← person(X)

← person(self:X, spouse:Y), not MinMarried(X), not MinMarried(Y)

⇐ MinMarried(X)
}

Here we have a DISJUNCTIVE logic program defining the concept **MinMarried**; the disjunctive rule

MinMarried(X) V not_MinMarried(X) ← person(X)

partitions the set of persons into two subsets, **MinMarried** and **not_MinMarried**, i.e., it guesses a possible solution; the strong constraint

← person(self:X, spouse:Y), not MinMarried(X), not MinMarried(Y)

checks the guess, that is, verifies the statement "X and Y are married and none of them belong to **MinMarried**" to be false; finally, the WEAK CONSTRAINT

⇐ MinMarried(X)

minimized the number of persons that belong to **MinMarried**. It holds that each stable model of this program is a possible set of instances of **MinMarried**.

Derived Attributes

Axioms can be used to derive attribute values as well; to see this point, let us extend the class **Person** by the attribute *has_ancestors* defined as having SETS of persons as possible values:

```
person  has attributes
          name: string
          age: integer
          father: person
          mother: person
          spouse: person
          lives: place
          has_ancestors: setOf(person)
end;
```

The values for *has_ancestors* are derived using the following axioms:

```
has_ancestor(X,Y) ← person(self:X, father:Y)
has_ancestor(X,Y) ← person(self:X, mother:Y)
has_ancestor(X,Y) ← has_ancestor(X,Z), has_ancestor(Z,Y)
has_ancestors(X,<Y>) ← has_ancestor(X,Y).
```

The latter rule above associates with each person X the set <Y> of his ancestors -- *grouping set* ([3], [16]).

As another example, consider the class *faculty*

```
faculty has attributes
          name: string;
          has_enrolled: setOf(student)
          last_enrolled: student;
end;
```

where both **has_enrolled** and **last_enrolled** are derived attributes . In particular, **has_enrolled** associates with each faculty the respective set of enrolled students:

has_enrolled(X,<Y>) ← student(self:Y, enrolled:X).

Likewise, the attribute **last_enrolled** (i.e., the student with the greatest registration number) is derived as follows:

Last_enrolled(X,Y) ← student(self:X, enrolled:Y, registration#:Z), not exists_bigger_reg#(Z,Y)

Bigger_reg#(X,Y) ← student(self:Z, registration#: W, enrolled:Y), W>X.

3 A simple hierarchy

The following example shows an ontology consisting of a simple hierarchy of living beings.

- LIVING_BEING
  { mortal()← }
  - ANIMAL isa LIVING_BEING
    - MAMMAL isa ANIMAL
      - CARNIVORE isa MAMMAL
        { eats(animal) ← }
        - LION isa CARNIVORE
      - HERBIVORE isa MAMMAL
        { eats(plant) ←
        - GIRAFFE instanceof HERBIVORE
          { eats(leaf) ← }
        - COW instanceof HERBIVORE
    - BIRD isa ANIMAL
      { flies()← }
      - PENGUIN instanceof BIRD
        { ¬flies() }
      - EAGLE instanceof BIRD
  - PLANT isa LIVING_BEING
    - LEAF partof PLANT
    - GRASS isa PLANT

Here we have a number of concepts related by the built-in relations *Isa, PartOf* and *InstanceOf*; we note that these concepts have no (explicit) attributes and are defined through some properties, such as *mortal()*, associated with **Living_Being**, *eats(animal),* specified for Carnivore, and others.

We enrich the specification of our ontology by adding the following DISJOINTNESS axioms:

← ISA*(X, animal), ISA*(Y, plant), X=Y
← InstanceOf(X, animal), InstanceOf(Y, plant), X=Y

expressing that **Animal** and **Plant** are disjoint classes (ISA* is used to denote the transitive closure of *Isa*). Likewise, we can state that **Carnivore**, **Herbivore** and **Bird** are disjoint too.

We next provide an intuitive semantics of hierarchies by using our running example. The keywords for this semantics are INHERITANCE and DEFAULT REASONING. Indeed, properties defined for a concept are inherited by its sub-concepts, unless explicitly negated (to this end we use TRUE NEGATION). For an instance, the property *eats(herbivore, plant)* is inherited by both *giraffe* and *cow*, so that both *eats(giraffe, plant)* and *eats(cow, plant)* hold. However, we apply the principle that specific pieces of information are more reliable than generic ones. So, if we ask "what does giraffe eat?" the answer will be "leaf" (*eats(giraffe, leaf)* is more specific than *eats(giraffe, plant)*), while the question "what does cow eats?" will be answered by "plant" (indeed, no specific information is available for the concept *cow*). Inherited information can be exploited in several ways. For an instance, if we ask whether giraffe eats plants, the answer will be YES. More interestingly, if we ask if cows eat grass, the answer will be MAYBE. This is because the fact *eats(cow, grass)* is not in contradiction with the general knowledge *eats(cow, plant)*, but we have no evidence about its truth. So, we are in presence of a fact that is neither true nor false -- it is undefined (*maybe* fact). Likewise, *eats(cow, leaf)* will have answer MAYBE.

Inheritance of properties can be blocked by using true (classical) negation. In our ontology, the general property *fly(bird)* is overwritten by the negative fact ¬*flies(penguin)* (here, ¬ is the classical negation symbol); so, the query *flies(penguin)* will be answered NO, while *flies(eagle)* is true.

QUERY	EXPECTED ANSWER	COMMENTS
?- mortal(X)	X = living being	
?- mortal(animal)	YES	
?_eats(herbivore,X)	X = plant	
?_eats(giraffe,X)	X = leaf	A more specific information prevails on a general one
?_eats(cow,X)	X = plant	*Eats(plant)* is inherited by *animal* and no more specific information holds for *cow*
?_eats(cow,grass)	MAYBE	*Grass Isa Plant* and no more specific

		information is available; so, it might be true
?_eats(cow,leaf)	MAYBE	*Leaf partOf Plant* and no more specific information is available; so, it might be true
?_eats(X,grass)	X = herbivore-MAYBE	
?_eats(giraffe,plant)	YES	
?_eats(carnivore,bird)	MAYBE	
?_flies(X)	X = bird	
?_flies(penguin)	NO	True negation is used to override general information
?_flies(eagle)	TRUE	

Table 1. Intuitive semantics of inheritance

4 Multiple inheritance

Here we show an example of multiple inheritance where contradicting pieces of knowledge are inherited; this is a typical case of non-monotonic reasoning, where adding new knowledge may invalidate the previous one.

The example models a financial consultancy where two experts, Expert1 and Expert2, are asked to advise an investor. Briefly:

1. Expert1 suggests to buy shares when the bull rules over the stock market. More precisely, he suggests to buy "dynamic" stocks, using the money gained selling "defensive" ones. In case of bear ruling, he advises to buy Treasury bonds and defensive titles.
2. Expert2 suggests, when the bear rules, not to buy any kind of shares, but taking only Treasury bonds.
3. Both Expert1 and Expert2 think that, if the potential investor is not available to risk, it is preferable not to buy shares.
4. In the end, the investor: if the budget is below a certain threshold, say c1, he does not buy anything; he can risk only if
    - The budget is above the threshold c1
    - He is already covered enough by Treasury bonds.

The assumption is that Expert1 and Expert2 are equally trustable; however, we consider the investor's opinion to be decisive for the final decision.

We note that there exists a conflict among different opinions: when the bear rules, Expert1 suggests defensive shares, while Expert2 advises against buying any kind of shares.

Next we show the OntoDLP encoding. For the sake of simplicity, we split stocks into two categories, dynamic and defensive:

```
Stock(dynamic)
Stock(defensive).

EXPERT1
{
        Buy(X) ← bull, stock(X), X = dynamic.
        Sell(X) ← bull, stock(X), X = defensive.
        Buy(bond) ← bear.
        Buy(X) ← bear, stock(X), X = defensive.
        ⇐ not open_to_risk, titolo(X), buy(X).
}

EXPERT2
{

        ¬ buy(X) ←bear, stock(X).
        buy(bond) ← bear
        ⇐ not open_to_risk, titolo(X), buy(X).
}

INVESTOR
{
        ¬ buy(X) ← stock(X), budget(Y), Y < c1.
        Open_to_risk ← budget(X), X >= c1, investment_bond(Y), Y > c2.
}
```

These concepts are partial-ordered by *MoreReliableThan* (<) as follows: Investor < Expert1, Investor < Expert2.

The rules associated with each concept above state the respective knowledge about the application domain. In particular, the weak constraint

```
⇐ not open_to_risk, titolo(X), buy(X),
```

which is common to both experts, encodes point 3 above according to which it is *preferable* not to buy shares if the investor is not open to risk; that is, it states a condition that should *preferably* be satisfied, but *not necessarily*.

SOLUTIONS: let's ponder about the following cases

1) Hypothesis: **budget < c1, bull**: the budget is below the threshold, so the investor idea (do not buy anything) wins against all the rest; so, OntoDLP offers a single

solutions, in which the following facts are true: *not open_to_risk*, *not buy(defensive)*, *not buy(dynamic), sell(defensive)*.

2) Hypothesis: **budget >= c1, investment in bond <= c2, bull:** the investor has enough money to buy; Expert1 suggests, since the bull rules, to buy dynamic stocks and sell defensive ones; Expert2 has no opinion. So, again, a single solution: { *not open_to_risk*, *buy(dynamic)*, *sell(defensive)* }.
3) Hypothesis: **budget >= c1, investment in bond <= c2, bear:** the investor has enough money to buy, but cannot risk; since bear rules, both Expert1 and Expert2 suggest to buy bonds. In addition, Expert1 suggests defensive stocks while Expert2 suggests not to buy any. Anyway, since the investor is not open to risk, both experts think it is preferable not to buy stocks (weak constraint); so the Expert2's opinion (do not buy) overcomes Expert1's one; here, we point out how the weak constraint supports one of two potential contradicting solutions. There is, again, another single solution, in which the true facts are: {*not open_to_risk, buy(bond), not buy (defensive), not buy(dynamic)* }..
4) Hypothesis: **budget >= c1, investment in bond > c2, bear:** the investor can buy, and he's open to risk. Both Expert1 and Expert2 suggest bonds. Expert1 suggests also defensive stocks, while Expert2 suggests not to buy stocks at all. There is a clash, not solvable (they are trustable, both), so that the only true fact is *buy(bond)* (note that *buy(defensive)* is undefined).

5 Synonyms

An ontology can be regarded as a "controlled vocabulary" where a concept or a relation may have several different names, one of which is the preferred one. Thus, we can define the following equivalence relation

Synonyms (concept_name: concept, synonym: concept)

associating to each concept its synonyms within the ontology. The following are possible instances:

Synonyms (place, spot)
Synonyms (spot, site).

6 Conclusions

In this paper we have given an informal presentation of a logic-based formalism for ontology specification. This formalism, called OntoDLP (where DLP stands for Disjunctive Logic Programming), supports abstraction mechanisms for the specification of concepts, relations and axioms. Concepts have a schema whose attributes may be of type concept. Relations are either taxonomic (*Isa, InstanceOf, PartOf* and *MoreReliableThan* that are built-in) or non-taxonomic (user-defined). Axioms are logic rules that are used to specify the semantics of concepts and relations

(properties of concepts, cardinality constraints and algebraic properties of relations, etc.).

OntoDLP supports very complex reasoning tasks based on the following basic features:

- Concepts, relations and axioms
- Disjunction in the head of rules: as shown in [8, 15], disjunction provides the language with a very high expressive power
- Constraints of two types: strong and weak [BUC97, 7]; the former are used to state conditions that must be satisfied, while the latter supports the specification of desiderata
- Classical negation, used to explicitly state negative information; as shown in [8], classical negation can be used within hierarchies to override inherited information
- Default reasoning, i.e., the (multiple) inheritance of properties from concepts to sub-concepts based on mechanisms whereby more specific (reliable) information prevails on general one.

Such features make OntoDLV a powerful language for ontology specification.

The semantics of OntoDLP is based on stable models (to handle disjunction) where facts can be either true, false or undefined (*maybe*); as we have seen, a three-value logic is very intuitive for the purpose of modeling inheritance-based reasoning.

OntoDLP is currently being implemented as an extension of the disjunctive logic programming system DLV [7] to take into account extra-logic constructs (hierarchies, inheritance, etc.)

References

1. J. Bouaud, B. Bachimont, J. Charlet and P. Zweigembaum, “Methodological principles for structuring an ontology” *In ACM Press*, editor, Proc. of IJCAI95 Workshop: Basic Ontological Issues in Knowledge Sharing, 1995.
2. F. Buccafurri, N. Leone and P. Rullo, "Strong and Weak Constraints in Disjunctive Datalog", *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach and A. Nerode editors, 1265 pp. 2-17, Springer Verlag, (Jul) 1997.

[3] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur and C. Zaniolo, “The LDL System Prototype”, IEEE TKDE, vol. 2, nr. 1, 1990

[4] P. Cholewinsky, A. Marek, V. W. Mikitiuk and M. Truszczynski, “Computing with default logic”, *Journal of Artificial Intelligence*, 112(1-2) pp. 105-146, 1999.

[5] DAML+OIL. Specification. In www.daml.org/2001/03/daml+oil-

index.htm, 2001.

[6] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, "A Deductive System for Non-Monotonic Reasoning", *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach and A. Nerode, editors, 1265, pp.364-375, Springer, LNAI 1997.

[7] W. Faber and G. Pfeifer, "DLV homepage", (www.dlvsystem.com), since 1996.

[8] M. Gelfond and V. Lifschitz, "Classical Negation in Logic Programs and Disjunctive Databases", *New Generation Computing*, 9 pp. 365-385, 1991.

[9] M. Gelfond, and V. Lifschitz, "The Stable Model Semantics for Logic Programming", *In 5fth Logic Programming Symposium*, pp. 1070-1080, Cambridge Mass, MIT Press, 1988.

[10] G. Gottlob, "Complexity and Expressive Power of Disjunctive Logic Programming", *Proceedings ILPS '94*, pp. 23-42, MIT Press, 1994.

[11] S. Greco, N. Leone, P. Rullo, "COMPLEX: An Object-Oriented Logic Programming System", *IEEE Transaction on Knowledge and Data Engineering*, 4(4), (Aug) 1992.

[12] N. Guarino, "Formal ontology, conceptual analysis and knowledge representation", *International Journal of Human-Computer Studies*, 43(5/6) pp. 625-640, Special issue on The Role of Formal Ontology in the Information Technology, 1995.

[13] N. Guarino and C. Welty, "A formal ontology of properties", In *Knowledge Engineering and Knowledge Management: Methods, Models and Tools. Int. Conf. EKAW'2000*, R. Dieng and O. Corby, editors pp. 97-112. Springer-Verlag, 2000.

[14] E. Laenens and D. Vermeir, "A Fixpoint Semantics for Ordered Logic", *Journal of Logic and Computation*, 1(2) pp. 159-185, (Dec) 1990.

[15] N. Leone, P. Rullo and F. Scarcello, "Disjunctive stable models: Unfounded sets, fixpoint semantics and computation*", Information and Computation*, 135(2) pp. 69-112, (June) 1997.

[16] N. Leone and P. Rullo, "Ordered Logic Programming with Sets", *Journal of Logic and Computation*, 3(6) pp. 621-642, (Dec) 1993.

[17] I. Niemelä and P. Simons, "Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LogicPrograms", *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, 1997.

[18] N. Noy, "Tutorial on Ontology Engineering", *Int. Semantic Web Working Symp. (SWWS'2001)*, www. Semanticweb.org/SWWS/program/tutorials/tutorial1/2001.

[19] P. Rao, K. Sagonas, T. Swift, D.S. Warren and J. Friere, "XSB: A system for efficiently computing well-founded semantics", *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach, and A. Nerode, editors, 1265, pp. 430-440, Springer, LNAI, 1997.

[20] W. Swartout, R. Patil, K. Knight and T. Russ, "Toward distributed use of large-scale ontologies", *Spring Symposium Series on Ontological Engineering*, pp. 33-40, Stanford, AAAI Press, 1997.
[21] F. van Harmelen and I. Horrocks, "FAQs on OIL: the Ontology Inference Layer", *IEEE Intelligent Systems* 15(6) pp. , 69-72, 2000.