

Declarative Methods for the Evaluation of Ontologies

Dietmar Seipel and Joachim Baumeister

The ontology web language Owl has been established as a standardized representation for knowledge, especially in the context of the semantic web. An important facet of the management of such knowledge bases consists in its evaluation. Besides standard evaluation methods described in the literature particular applications can require to consider further measures. In this paper, we use a declarative, logic-based XML query and transformation language called FnQuery, which is suitable for flexibly defining queries for evaluating Owl-based knowledge. The queries are evaluated using logic programming and nonmonotonic reasoning systems. The presented approach could be extended to handle complex refactorings as well.

1 Introduction

In the last years, the *ontology web language* OWL has been established as a standardized and widely accepted representation of various types of knowledge, especially in the context of building semantic web applications.

The *acquisition and management* of OWL-based knowledge also is concerned with its syntactic and semantic evaluation. In the literature some design criteria for evaluating ontologies were defined, cf. Gruber [9]: clarity, coherence, extendibility, minimal encoding bias, and minimal ontological commitment. However, the actual evaluation task heavily depends on the requirements of the particular application domain. Therefore, tools are required that cannot only handle standard validation queries, but that can be flexibly adapted to application driven quality measures. A declarative approach for rapidly formulating queries is a promising method for coping with the problem of varying evaluation tasks.

In this paper, we use a declarative, logic-based XML query and transformation language called FNQuery, cf. [17], which we have implemented in SWI-PROLOG.

- FNQuery is used for implementing certain *syntactic tests* for the verification of OWL-based ontologies, such as tests for circularity, inconsistency, or redundancy. Some of the tests could be based on the semantic web library Triple 20 of SWI-PROLOG [20], which we have incorporated into our system.
- More refined evaluations including *semantic tests* can be performed in our approach, too. We use FNQuery for transforming an OWL knowledge base into a disjunctive logic program (under certain closed-world assumptions), which we then can evaluate using standard tools for *nonmonotonic reasoning* (NMR) and *answer set programming* (ASP) such as dlv [13].

In some situations the logic programming (LP) approach to reasoning about OWL knowledge bases seems to be more intuitive than the description logic (DL) approach. We embed DL into a logic programming framework for increasing the expressibility and for improving the ease of representation

of knowledge. The *closed-world assumption* (CWA) for basic concepts (with a finite extension) can be expressed naturally in DL; the CWA for roles, however, cannot be expressed in DL. Mixing ideas from both DL and disjunctive logic programming by considering CWAs in DL refines the standard semantics of DL. Obviously, the translated disjunctive logic programs can be combined with arbitrary other disjunctive logic rules, which results in an expressive, *hybrid* knowledge representation approach similar to the one of Eiter et al. [5]. We assume that the reader has some basic knowledge about OWL and DL; detailed introductions can be found in [1] and [2], respectively.

The rest of this paper is organized as follows: In Section 2 we will use logic programming for extracting knowledge from OWL, based on which we can handle basic taxonomy evaluation in Section 3. Section 4 shows that more refined evaluations of ontologies can be done by embedding DL into ASP. Some related approaches that can be found in literature are briefly described in Section 5.

2 Using Logic Programming for Owl Knowledge Bases

We use logic programming and nonmonotonic reasoning techniques for extracting knowledge from and for reasoning about OWL-based ontologies.

2.1 Knowledge Extraction by FnQuery

In this section, we will provide some introduction to FNQuery and motivating examples applying the novel technique to OWL-based ontologies. A detailed description of FNQuery can be found in [17]. We demonstrate FNQuery using a modified version of a printer ontology that was introduced in Antoniou and van Harmelen [1], cf. Figure 1, and we will particularly consider the following OWL fragment:

```
<owl:Class rdf:ID="personalPrinter">  
<rdfs:subClassOf
```

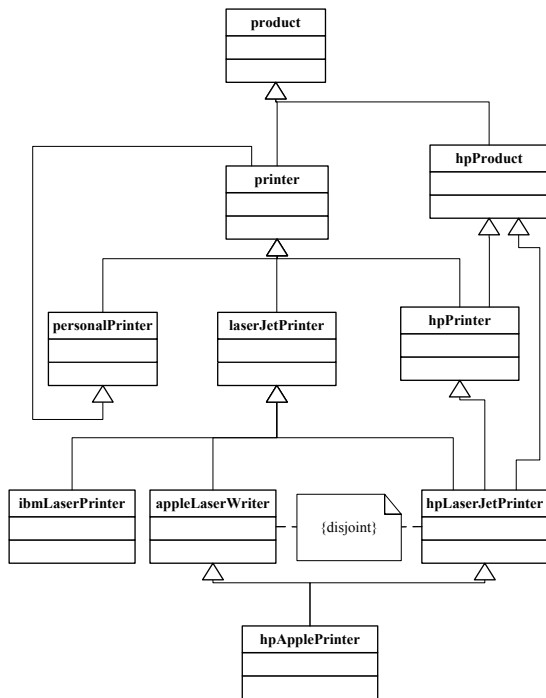


Figure 1: A Printer Ontology in UML

```

    rdf:resource="#printer"/>
</owl:Class>

<owl:Class rdf:ID="hpProduct">
  <rdfs:comment>
    HP products are exactly those products
    that are manufactured by Hewlett Packard
  </rdfs:comment>
  <owl:intersectionOf>
    <owl:Class rdf:about="#product"/>
    <owl:Restriction>
      <owl:onProperty
        rdf:resource="#manufactured_by"/>
      <owl:hasValue>
        <xsd:string
          rdf:value="Hewlett Packard"/>
        </owl:hasValue>
      </owl:Restriction>
    </owl:intersectionOf>
  </owl:Class>

```

We load an OWL file into PROLOG using the call `file_to_owl(File, Owl)`, which returns a corresponding PROLOG term structure `Owl`. Alternatively, it is also possible to represent an OWL file as a collection of PROLOG facts. We can extract the information that we need for further reasoning from `Owl`. In the following, we will give some example predicates for deriving the relationships `subClassOf`, `intersectionOf`, and `disjointWith` using complex path expressions in FNQuery.

subClassOf The following rule uses a complex path expression for selecting an `rdfs:subClassOf`-element c_2 within

an `owl:Class`-element c_1 . In that case the predicate `@'rdf:ID'=C1` determines the attribute `rdf:ID` of c_1 as $C1$, and the attribute `rdf:resource` of c_2 gives the reference $R2$ to a subclass $C2$:

```

owl_to_rule(isa, Owl, [isa(C1, C2)]) :-
  R2 := Owl^'owl:Class'::[@'rdf:ID'=C1]
  ^'rdfs:subClassOf'@'rdf:resource',
  owl_reference_to_id(R2, C2).

```

```

owl_reference_to_id(Reference, Id) :-
  concat('#', Id, Reference).
owl_reference_to_id(Id, Id).

```

Note that we have to dereference attributes such as `rdf:resource` and `rdf:about` to make them compatible with `rdf:ID`-attributes. After loading the OWL knowledge base `printer.owl` we can derive that `personalPrinter` is a subclass of `printer`:

```

?- file_to_owl('printer.owl', Owl),
   owl_to_rule(isa, Owl, Rule).

Owl = ...,
Rule = [isa(personalPrinter, printer)]
Yes

```

intersectionOf The first of the following rules selects an `owl:Class`-element having a subelement `owl:intersectionOf` containing an `owl:Restriction`, and then the second rule parses this restriction.

```

owl_to_rule(intersectionOf, Owl, Rule) :-
  Restriction := Owl
  ^'owl:Class'::[@'rdf:ID'=C1]
  ^'owl:intersectionOf'::[
    ^'owl:Class'@'rdf:about'=R2]
  ^'owl:Restriction',
  owl_reference_to_id(R2, C2),
  owl_restriction_parse(
    has_value, Restriction, P=V),
  Rule = [X:C1]-[X:C2, (X,V):P].

owl_restriction_parse(has_value,
  Restriction, P=V) :-
  R := Restriction
  ^'owl:onProperty'@'rdf:resource',
  owl_reference_to_id(R, P),
  ( Type = 'xsd:string'
  ; Type = 'xsd:integer' ),
  V := Restriction^'owl:hasValue'
  ^Type@'rdf:value'.

```

From the OWL knowledge base `printer.owl` we can derive a rule saying that every product that is manufactured by Hewlett Packard is an `hpProduct`; the rule is represented in the syntax of the disjunctive logic programming system DISLOG [16]:

```

?- file_to_owl('printer.owl', Owl),
   owl_to_rule(intersectionOf, Owl, Rule).

Owl = ...,
Rule = [X:hpProduct]-[X:product,
  (X,'Hewlett Packard'):manufactured_by]
Yes

```

disjointWith The following rule extracts disjointWith-constraints:

```
owl_to_rule(disjointWith, Owl,
  [disjointWith(C1, C2)]) :-
  R2 := Owl^owl:Class'::[@'rdf:about'=R1]
  ^'owl:disjointWith'@'rdf:resource',
  owl_reference_to_id(R1, C1),
  owl_reference_to_id(R2, C2).
```

General Axioms Finally, we need some general axioms for propagating the relationships `instanceOf` and `isa` transitively:

```
X:C2 :-
  X:C1, subClassOf(C1, C2).

subClassOf(C1, C2) :-
  isa(C1, C2).
subClassOf(C1, C2) :-
  isa(C1, C), subClassOf(C, C2).
```

We have chosen an encoding which *reifies* concepts and roles: $X:C$ is preferred over $C(X)$, since it allows for more reasoning about C ; e.g., we can derive all classes C , such that a given X is an instance of. Moreover, $isa(C1, C2)$ is preferred over $C2(X) :- C1(X)$, since it allows for reasoning about subclasses rather than just reasoning about their instances.

2.2 Reasoning Using LP/NMR Tools

For reasoning about OWL knowledge bases two approaches are possible: a two-layer approach using FNQuery and non-monotonic reasoning tools, and an integrated approach embedding FNQuery into PROLOG-based (disjunctive) logic programming.

Firstly, we can load an OWL ontology into PROLOG and create a knowledge base consisting of the facts and rules that we have shown in Section 2.1 using FNQuery. Based on this knowledge base we can do reasoning using DATALOG or NMR tools such as dlv or Smodels. It is not possible to directly embed FNQuery into these systems, since they cannot handle function symbols. But the presented queries can be compiled to equivalent DATALOG queries which can be applied to the PROLOG fact representation of the OWL knowledge base, such that all of the reasoning could be done using the NMR tools.

Secondly, we can also integrate knowledge extraction from OWL using FNQuery and reasoning in PROLOG. In that case we do not have to create the complete PROLOG knowledge base before we can start reasoning. For non-cyclic class hierarchies the reasoning can be done top-down using PROLOG. In general, we have to use efficient and terminating bottom-up approaches (including magic sets techniques). Such methods have for instance been implemented in the PROLOG-based disjunctive logic programming system DISLOG, which can handle function symbols and allows for directly embedded calls to FNQuery.

3 Taxonomy Evaluation

According to Gómez-Pérez [6, 7] the evaluation of an ontology includes the inspection of its taxonomy, which should be checked for *inconsistency*, *incompleteness*, and *redundancy*, cf. Figure 2.

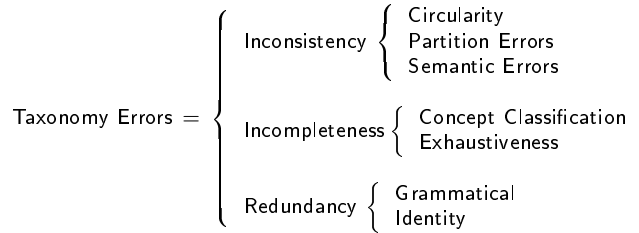


Figure 2: Taxonomy Errors in an Ontology

In the following, we will show that our approach can be easily applied for declaratively implementing most of these standard tests. Furthermore, the examples will motivate that additional, application-driven measures can be implemented analogously. Measurements of run times using dlv and Triple 20 will be given in Section 4.

3.1 Inconsistency

Inconsistency tests check, if a contradictory definition of an individual can be found or if contradictory knowledge can be derived from other definitions and axioms given in the ontology. We distinguish between *circularity errors*, *partition errors*, and *semantic errors*.

Circularity A circularity is identified if a class defined in an ontology is a specialization or generalization of itself. For example, in Figure 1 there exists a cycle of length 2 between the concepts `printer` and `personalPrinter`. We can easily detect circularities with the following query:

```
?- isa(C1, C2),
   subClassOf(C2, C1).

C1 = personalPrinter, C2 = printer
Yes
```

Observe, that for keeping the report on circularities compact it is sufficient to report only given `isa`-relationships that are involved in a cycle; moreover, this yields a much more efficient evaluation.

Partition Errors In a taxonomy subconcepts of a class can be defined as a disjoint partition of the generalizing class. E.g., in Figure 1 the concept `laserJetPrinter` has two disjoint subconcepts `appleLaserWriter` and `hpLaserJetPrinter`. A class partition error occurs, if a class is defined as a common subclass of several classes of a disjoint partition. Analogously, a common instance of two

disjoint classes resembles an instance partition error. In our example the concept `hpApplePrinter` causes a class partition error since it is inherited from the disjoint concepts `appleLaserWriter` and `hpLaserJetPrinter`. Class partition errors can be found using the following query:

```
?- disjointWith(C1, C2),
   subClassOf(C, C1),
   subClassOf(C, C2).

C = hpApplePrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter
Yes
```

Semantic Inconsistency Semantic inconsistencies occur if the developer of the ontology asserts incorrect semantic classifications; e.g., if a class is falsely defined as a subconcept of another class, such that there is no semantic relation between the concepts.

Semantic inconsistencies often are difficult to find using automated methods, but they may be detected by a manual inspection of the ontology. However, assistance can be offered using `FNQuery`, e.g., by transforming (parts of) the taxonomy into a human-readable format, cf. [18].

3.2 Incompleteness

Ontological knowledge can be incomplete if `isa` relationships between concepts are imprecisely defined or possible information about decompositions is missing. Typically, incompleteness occurs if (probably) important concepts are omitted during the definition of the taxonomy. Furthermore, partitions are incompletely defined if knowledge about the disjointness or exhaustiveness of a partition is omitted.

Unfortunately, it is very difficult to provide automated methods for detecting such errors. However, it is possible to support the identification of incomplete ontologies. E.g., partially defined `disjointWith` relationships between siblings may indicate an incomplete definition of a disjoint partition. A partially defined disjointness can be formulated as follows:

```
?- isa(C1, C), isa(C2, C), isa(C3, C),
   disjointWith(C1, C2),
   not(disjointWith(C2, C3)).

C = laserJetPrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter,
C3 = ibmLaserPrinter
Yes
```

Furthermore, the manual inspection of an ontology for finding incomplete knowledge can be supported by preparing human-readable reports using `FNQuery`. An interesting measure, which requires *aggregation*, would be the number of classes C , such that most, but not all of their subclasses have been modelled as disjoint, since this indicates potentially missing disjointness conditions.

3.3 Redundancy

For a given ontology we can detect redundant class/instance definitions or redundant `subClassOf` relations in the taxonomy.

Redundant subClassOf/instanceOf Relations An `isa` definition is redundant, if it can be derived from other `isa` relationships. These redundancies are harder to detect if transitive subclasses are involved:

```
redundant_isa(C1->C2->C3) :-
  isa(C1, C3), subClassOf(C3, C2),
  isa(C1, C2).
```

In the printer example in Figure 1 the `subClass` relation between `hpLaserJetPrinter` and `hpProduct` is redundant, since an indirect `subClass` relation already exists due to the concept `hpPrinter`:

```
?- redundant_isa(X).

X = hpLaserJetPrinter ->
  hpPrinter -> hpProduct
Yes
```

Analogously, redundant `instanceOf` relations occur if more than one `instanceOf` relation is defined; we also distinguish direct and indirect (transitive) redundancy.

Identical Definition of Classes/Instances In larger ontologies or ontologies that are developed in a distributed environment, more often identical definitions of classes or instances can be found. Two classes are identical if they have an equivalent formal definition (given by the data type properties etc.), but are named differently; similarly, instances can be identical.

```
<owl:DatatypeProperty
  rdf:ID="manufactured_by">
  <rdfs:domain rdf:resource="#product"/>
  <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>
```

We can easily extract the corresponding facts `dtProperty(Domain, (ID, Range))` from the OWL knowledge base using `FNQuery` and then compute, for a given class c , the DL concept

$$\forall dtProperty. (dtProperty^- . \{c\}),$$

which consists of all classes d that are subsumed by c ; for each method of such a class d there exists a method of c with the same ID and Range (R^- denotes the inverse of a role R). Using ASP it would be possible to compute all such pairs (c, d) by letting c vary over all classes.

Another interesting problem is to find all classes having methods that are also defined for some of their subclasses, i.e., to compute the DL role

$$dtProperty \sqcap (subClassOf \circ dtProperty),$$

where \circ denotes the composition of roles.

4 Evaluation of Ontologies Using Answer Set Programming

We translate DL knowledge bases into disjunctive logic programs, which we evaluate using the system dlv. We extend the approach taken by Grosf et al. [8], who have represented *Description Horn Logic (DHL) ontologies* in first-order logic. In the following we will briefly explain some differences. Firstly, we add the contrapositives for implications resulting from concept inclusions. Secondly, for axioms and concepts with quantifiers we depart from Grosf’s approach, if certain *closed-world assumptions* (CWA) can be made. Finally, since we translate to disjunctive logic programs, we can handle *negation*, arbitrary *unions* and *intersections*, and the *composition of roles* in DL axioms.

DHL Ontologies Grosf et al. [8] have defined *Description Horn Logic* as an expressive DL language containing only axioms of the form

$$\begin{aligned} C \sqsubseteq D, A \equiv B, \\ C \sqsubseteq \forall P.D, \exists P.C \sqsubseteq D, \\ P \sqsubseteq Q, P \equiv Q, P \equiv Q^-, P^+ \sqsubseteq P, \\ a : D, (a, b) : P, \end{aligned}$$

where C is an \mathcal{L}_b -class, D is an \mathcal{L}_h -class, A, B are \mathcal{L} -classes, P and Q are properties, and a, b are individuals. Here, \mathcal{L}_b and \mathcal{L}_h are the classes of concepts from which can be mapped into the head or body of definite rules, respectively, and $\mathcal{L} = \mathcal{L}_b \cap \mathcal{L}_h$. It has been shown that DHL programs can be translated to logically equivalent definite logic programs, i.e., programs having the same set of models and entailed conclusions.

Concept Inclusions A concept inclusion of the form $C \sqsubseteq D$ over basic concepts can be represented by two rules

$$X:D \leftarrow X:C, \quad X:\neg C \leftarrow X:\neg D.$$

In our approach we need the second rule, which represents the contrapositive of the first rule, if we do not have the CWA for the concepts C and D .

Negation Negated concepts are simply encoded as facts $X:\neg C$, and for arbitrary concepts C and instances X , the generic denial rule $r_0 = \leftarrow X:C \wedge X:\neg C$ (which resembles a disjunctive rule with zero head atoms) forbids the derivation of contradictory `instanceOf` facts.

Unions and Intersections For concept inclusions of the form $C \sqsubseteq C_1 \sqcup C_2$ over basic concepts, which are not allowed in Grosf’s approach, we can use disjunctive logic rules:

$$\begin{aligned} X:C_1 \vee X:C_2 \leftarrow X:C, \\ X:\neg C \leftarrow X:\neg C_1 \wedge X:\neg C_2. \end{aligned}$$

A practical example of a concept inclusion of the form above would be $product \sqsubseteq printer \sqcup hpProduct$, i.e., every product must be either a printer or an hpProduct. Similarly, the contrapositives of concept inclusions $C_1 \sqcap C_2 \sqsubseteq C$ with intersections on the left hand side lead to disjunctive rules, as we will see in the example below.

Quantifiers For concepts with quantifiers certain CWAs can be made. In particular, we can translate concepts $\forall R.C$ and $\exists R.C$ into normal logic rules, i.e., rules with default negations, if we have the CWA for the role R and the concept C , respectively. This is useful in many situations. E.g., assume that we want to express the fact that a printer is an HP-printer if all its components are HP-products. Intuitively, this should be possible using the DL formula

$$printer \sqcap \forall component.hpProduct \equiv hpPrinter.$$

However, DL works with the *open-world assumption*, cf. [2], which in particular means that we do not know about all the facts for *component*. Thus, no HP-printers can be derived from the formula above, since for each candidate printer there could be a component that is not manufactured by HP. If we have the CWA for the role *component*, then we can translate the concept $\forall component.hpProduct$ into normal logic rules. Also the evaluation of the quantified DL concept $\forall dtProperty.(dtProperty^- \{c\})$ in Section 3.3 requires the CWA for the role *dtProperty*.

In general, we translate a concept $\forall R.C$ into a set of rules defining a new concept, which we call $[\forall R.C]$, and a helper concept $(\forall R.C)$, if we have the CWA for the role R :

$$\begin{aligned} r_1 &= X:[\forall R.C] \leftarrow X:\top \wedge \text{not } X:(\forall R.C) \\ r_2 &= X:\neg[\forall R.C] \leftarrow X:(\forall R.C) \\ r_3 &= X:(\forall R.C) \leftarrow (X, Y):R \wedge \text{not } Y:C \end{aligned}$$

\top denotes the top concept of the taxonomy. We replace all occurrences of the concept $\forall R.C$ in axioms by the new concept $[\forall R.C]$. Note that we distinguish between the negation $\neg[\forall R.C]$ of the new concept and the helper concept $(\forall R.C)$, but that – as a consequence of the rules r_1 and r_2 and the denial rule r_0 – these concepts are correlated.

In the *hpPrinter* example above, where $R = component$ and $C = hpProduct$, we first define the new concept $D = [\forall component.hpProduct]$, and then we can translate the modified, quantifier-free DL equivalence $printer \sqcap D \equiv hpPrinter$ to a disjunctive logic program. Notice that one of the rules resulting from a contrapositive is disjunctive, due to the intersection on the left hand side of the DL equivalence:

$$X:\neg printer \vee X:\neg D \leftarrow X:\neg hpPrinter.$$

Similarly, a concept $\exists R.C$ can be translated into a rule defining two new concepts $[\exists R.C]$ and $(\exists R.C)$, if we have the CWA for the role R and the concept C :

$$\begin{aligned} r_5 &= X:[\exists R.C] \leftarrow X:(\exists R.C) \\ r_6 &= X:\neg[\exists R.C] \leftarrow \text{not } X:(\exists R.C) \\ r_7 &= X:(\exists R.C) \leftarrow (X, Y):R \wedge Y:C. \end{aligned}$$

We replace all occurrences of the concept $\exists R.C$ in axioms by the new concept $[\exists R.C]$, and we distinguish between the new concept $[\exists R.C]$ and the helper concept $(\exists R.C)$.

According to Grosf et al. we can translate arbitrary concept inclusions of the form $D \sqsubseteq \forall R.C$ and $\exists R.C \sqsubseteq D$ to logically equivalent rules $Y:C \leftarrow X:D \wedge (X,Y):R$ and $X:D \leftarrow (X,Y):R \wedge Y:C$, respectively. The contrapositives of these rules are $X:\neg D \leftarrow (X,Y):R \wedge Y:\neg C$ and $Y:\neg C \leftarrow (X,Y):R \wedge X:\neg D$, respectively. All of these rules can be evaluated using systems such as dlV. But, for evaluating inclusion axioms of the form $\forall R.C \sqsubseteq D$ and $D \sqsubseteq \exists R.C$, respectively, over non-closed roles and concepts we would need systems such as the tableau prover KRHyper [4], since the logically equivalent rules require Skolem functions: $\forall R.C \sqsubseteq D$ is equivalent to $(X, f(X)):R \wedge f(X):\neg C \leftarrow X:\neg D$, and $D \sqsubseteq \exists R.C$ is equivalent to $(X, f(X)):R \wedge f(X):C \leftarrow X:D$.

Composition of Roles This can be represented easily; e.g., the role describing redundant data type properties at the end of Section 3 gets translated to

$$(Z, Y) : dtPropertyRed \leftarrow (Z, Y) : dtProperty \wedge (X, Y) : dtProperty \wedge subclassOf(X, Z).$$

Semantics Since we translate a DL program P to a disjunctive logic program (with default negation) $\tau(P)$, we can additionally allow for unions on the right-hand side of concept inclusions, and we can translate quantification over closed roles and concepts. The disjunctive logic program $\tau(P)$ is evaluated under the stable model semantics, which ignores models that are non-minimal w.r.t. the closed roles and concepts. This strengthens the entailment relation: if P entails $a:C$ for a concept C and an instance a , then $a:C$ holds in all stable models of $\tau(P)$.

Practical Applications The small *printer ontology* which we have used for this paper consists of 154 concepts; no roles were considered. The transformed disjunctive logic program that we have created consisted of 156 `isa` relationships, from which 1 030 `subclassOf` relationships were derived. Using dlV we could find all inconsistency and incompleteness errors that we had implanted in about 2.5 seconds. Here, DL tools such as Racer [11] cannot assign the intuitive user-intended semantics to concepts such as `hpPrinter`.

Furthermore, we have applied our approach to a larger, real-world example taken from the medical domain: The taxonomy of the *MediConsult knowledge system* contains 1 595 diagnostic concepts organized in 2 003 `isa` relations. The depth of the taxonomy is 12, and the maximal number of direct subclasses (`isa`) of a class is 31. With dlV we have derived the 8 994 transitive `subclassOf` relations in 3 seconds, and dlV needed 14 seconds when we used quadratically recursive rules for computing `subclassOf` from `isa`. Interestingly, the tool Triple 20 from SWI-PROLOG was also able to derive them quite fast in 5 seconds; about 20 000 triples were created from the OWL loader. We have implanted

22 classes C as subclasses of disjoint classes $C1$ and $C2$ into the 13 577 `disjointWith` relationships of the taxonomy. Their detection took only 2.5 seconds in SWI-PROLOG, but it took about 3.5 minutes in dlV, since dlV does not implement indexing; Triple 20 does not consider `disjointWith`.

5 Related Work

Other approaches for DL reasoning can be found in literature. There are several papers showing how DL formulas can be represented in first-order logic, i.e., under the all-models semantics, cf. e.g., Grosf, Horrocks, Volz, and Decker [8]. Motik, Volz, and Maedche [15] optimize query answering in DL using disjunctive deductive databases. Hustadt, Motik, and Sattler [12] reduce the DL $SHIQ^-$ to disjunctive DATALOG programs. Eiter, Lukasiewicz, Schindlauer, and Tompits [5] combine ASP with DL by building rules on top of ontological concepts or axioms and to a limited extend by building ontologies on top of rules. So far, none of these approaches considers CWAs. Further extensions to expressive DLs have been described by Celanese and De Giacomo as well as Baader, Küsters and Wolter in the DL Handbook [2].

F-Logic, XPath-Logic and the tool XPathLog of May [14] allow for accessing XML documents in a logic programming style, but no nonmonotonic reasoning can be done on top of this. The knowledge extraction features seem to be comparable to FNQuery, but FNQuery offers additional *transformation* and *update* features, cf. [17, 18], which we can use for refactoring ontologies.

Ontology development tools and tool suites such as Protégé and OntoEdit [19] have been applied successfully in various domains including eCommerce, medicine, configuration, and software engineering. Our approach could be used as a plugin for extending the evaluation facilities of these general development tools. For example, OntoEdit offers support for the OntoClean methodology [10], which provides semantic analysis of ontologies based on predefined meta-properties. It should be possible to apply appropriate reasoning techniques using such meta-properties within our approach.

6 Discussion and Outlook

We have presented an approach for flexibly stating declarative queries for evaluating OWL knowledge bases, which is based on the XML query and transformation language FNQuery in combination with techniques from LP and ASP. In addition to standard DL queries we can process queries involving complex computations such as aggregations, and we can handle knowledge bases with CWAs.

So far we only motivated the implementation of standard evaluation tasks, e.g., testing for inconsistency, incompleteness, or redundancy. The presented approach can be also applied for specifying further measures inspecting the design and validity of an ontology. For example, detecting *lazy instances* based on the frequency of the instantiation

of concepts. Moreover, FNQuery can be used for finding over-detailed taxonomies based on mean values of hierarchy depth/breadth or for implementing a standardization check, which tests given ontologies against a given standard ontology in order to investigate its reusability and exchangeability.

In the future, we would like to integrate certain forms of reasoning – such as the inference and handling of subclass relationships – into FNQuery by allowing for parameterized queries. The described evaluation measures are a starting point for the application of *refactoring methods*, that provide a structured approach for improving the validity and the design of ontologies. Refactoring methods for knowledge bases were introduced in [3]. Since FNQuery can be used not only for stating queries on OWL but also for implementing declarative transformations and updates of XML documents, the presented approach can be extended to handle complex refactorings as well.

References

- [1] G. Antoniou, F. van Harmelen: A Semantic Web Primer, MIT Press, 2004.
- [2] F. Baader, D. McGuinness, D. Nardi, P. Patel-Schneider (Eds.): The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.
- [3] J. Baumeister, F. Puppe, D. Seipel: Refactoring Methods for Knowledge Bases, Engineering Knowledge in the Age of the Semantic Web: 14th Intl. Conference EKAW 2004, Springer LNAI 3257
- [4] P. Baumgartner, U. Furbach, M. Groß-Hardt, T. Klemmann: Model Based Deduction for Database Schema Reasoning, Research Report 5-2004, Univ. Koblenz.
- [5] T. Eiter, T. Lukasiewicz, R. Schindlauer, H. Tompits: Combining Answer Set Programming with Description Logics for the Semantic Web, Proc. Intl. Conf. on Principles of Knowledge Representation and Reasoning, KR 2004.
- [6] A. Gómez-Pérez, M. Fernández-López, O. Corcho: Ontological Engineering, Springer 2004.
- [7] A. Gómez-Pérez Evaluation of Taxonomic Knowledge on Ontologies and Knowledge-Based Systems, Proc. North American Workshop on Knowledge Acquisition, Modeling, and Management, KAW 1999.
- [8] B. N. Grosz, I. Horrocks, R. Volz, S. Decker: Description Logic Programs: Combining Logic Programs with Description Logics, Proc. Intl. WWW Conf. 2003.
- [9] T. Gruber: A Translation Approach to Portable Ontology Specifications. Journal of Knowledge Acquisition, vol. 5(2), 1993.
- [10] N. Guarino, C.A. Welty: Evaluating Ontological Decisions with OntoClean, Communications of the ACM, vol. 45 (2), 2002.
- [11] V. Haarslev, R. Möller: RACER System Description, Proc. Intl. Joint Conference on Automated Reasoning, IJCAR 2001, Springer LNAI 2083.
- [12] U. Hustadt, B. Motik, U. Sattler: Reducing SHIQ⁻ Description Logics to Disjunctive Datalog Programs, Proc. Intl. Conf. on Principles of Knowledge Representation and Reasoning, KR 2004.
- [13] N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, A. Polleres: The DLV System, Proc. European Conf. on Logics in Artificial Intelligence, JELIA 2002, Springer LNCS 2424.
- [14] W. May: XPath-Logic and XPathLog: A Logic-Programming-Style XML-Data Manipulation Language, Journal on Theory and Practice of Logic Programming (TPLP), vol. 4 (3), 2004.
- [15] B. Motik, R. Volz, A. Maedche: Optimizing Query Answering in Description Logics using Disjunctive Deductive Databases, Proc. Intl. Workshop on Knowledge Representation meets Databases, KRDB 2003.
- [16] D. Seipel: DISLOG – A Disjunctive Deductive Database Prototype, Proc. Workshop on Logic Programming, WLP 1997.
- [17] D. Seipel: Processing XML Documents in PROLOG, Proc. Workshop on Logic Programming, WLP 2002.
- [18] D. Seipel, J. Baumeister, M. Hopfner: Declaratively Querying and Visualizing Knowledge Bases in XML, Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management, INAP 2004, Springer LNAI.
- [19] Y. Sure, J. Angele, S. Staab: OntoEdit: Guiding Ontology Development by Methodology and Inferencing, Proc. Intl. Conf. on Ontologies, Databases and Applications of Semantics for Large Scale Information Systems, ODBASE 2002, Springer LNCS 2519.
- [20] J. Wielemaker: Triple 20 – An RDF/RDFS/OWL Visualisation and Editing Tool, www.swi-prolog.org/packages/Triple20.

Contact

Dietmar Seipel and Joachim Baumeister
 University of Würzburg, Institute for Computer Science
 Am Hubland, D – 97074 Würzburg, Germany
 {seipel, baumeister}@informatik.uni-wuerzburg.de



Dietmar Seipel is an associate professor at the University of Würzburg since 1995. His research areas are databases, knowledge bases, deductive databases, and logic programming. Currently, he is also investigating XML-based knowledge representations in different fields of applications.



Joachim Baumeister is research assistant at the University of Würzburg since 1999. His research focuses on practical knowledge engineering techniques including validation, restructuring, and semi-automatic learning of diagnostic knowledge. In 2004 he finished his PhD thesis on agile knowledge engineering.