Smelly Owls – Design Anomalies in Ontologies

Joachim Baumeister and Dietmar Seipel

Department of Computer Science, University of Wuerzburg, Germany
email: {baumeister, seipel}@informatik.uni-wuerzburg.de

Abstract

In the last years, ontologies have played a major role for building large, distributed, and heterogeneous intelligent systems. E.g., ontologies are one key technique of the semantic web layer. The development process of an ontology heavily depends on its evaluation. In this paper, we introduce several measures for the evaluation of ontological knowledge. Besides standard methods like taxonomic errors we also present novel metrics focusing on *design anomalies*. For the implementation of these measures we propose a declarative approach using the logic-based language FNQuery.

Introduction

Since the late 90's the impact of ontologies has increased significantly in the context of building intelligent systems. Ontologies describe a formal, conceptual, and executable model of an application domain. With the vision of the semantic web (Berners-Lee, Hendler, & Lassila 2001) the significance of ontologies has additionally being emphasized, and many applications have been deployed in industrial environments. The acquisition and management of ontologies also is concerned with its syntactic and semantic evaluation. According to Gómez-Pérez (Staab & Studer 2004, Ch. 13) the evaluation comprises the verification, the validation, and the assessment of ontologies. More formal criteria for evaluating an ontology are consistency, completeness, conciseness, expandability, and sensitiveness. A detailed and formal introduction of these terms can be found in (Gómez-Pérez 1996). In the last years, the ontology web language OWL has been established as a standardized and widely accepted representation of knowledge. OWL is build on RDF Schema and it can be expressed in XML syntax for a natural exchange and share between intelligent systems.

In this paper, we present methods for declaratively defining evaluation queries on OWL-based ontologies. Although there exists a collection of standardized measures for evaluating ontologies, e.g. taxonomic errors, in each particular application these measures need to be adapted w.r.t. the domain. Sometimes, even new types of appropriate measures are required for a particular domain. For example, the detection of design anomalies is of special interest, since it is suitable for improving the usability and maintainability of ontologies. Thus, such measures can indicate areas in the ontology that may be simplified, or they can avoid the reusability of particular concepts. Then, refactoring methods are executed subsequently to eliminate anomalies. Therefore, we present a declarative approach for flexibly defining evaluation queries and transformations.

The paper is organized as follows: In the next two sections we briefly introduce the ontology web language OWL and the query language FNQuery. Building on PROLOG technology the language FNQuery allows for the declarative analysis and modification of XML documents and is appropriate for the evaluation of OWL-based ontologies. After these introductory parts we describe standard measures for the evaluation of ontologies, i.e., the detection of taxonomic errors, and we show how FNQuery can be applied for these methods. The subsequent section presents design anomalies as more ambiguous measures for the evaluation of ontologies; also we show how FNQuery can be used for detecting such anomalies. Finally, we conclude the paper with a discussion of the presented work.

OWL in a Nutshell

The ontology web language OWL represents a standardized approach for machine-readable semantics of knowledge. It builds on the XML-based syntax of RDF(S), which is the standard format, although some equivalent alternative verbalizations for OWL are in use that are more human-readable.

An ontology contains definitions of hierarchies of *classes* (concepts), hierarchies of *properties* stating relations between classes or properties, and *instances* of classes and properties. OWL allows for the declaration of specialized properties like restrictions on properties, transitivity, disjointness, boolean combinations, and enumerations. Thus, OWL provides very expressive primitives for describing the semantics of knowledge, which in turn makes the evaluation process less efficient or even causes undecidability of statements. For this reason, three sub-languages were defined for OWL:

- OWL Full is the entire language including all modeling primitives for stating semantics on knowledge; e.g., the semantics of language primitives can be modified by applying arbitrary primitives to other primitives.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

- OWL DL is a reasonable subset of OWL Full which allows for efficient reasoning support. For short, the expressiveness of description logics (DL) is mapped in OWL DL.
- OWL Lite is further restricted in its expressiveness when compared to OWL DL; however, due to its limited expressiveness it is easy to learn and to implement, e.g., for users and for vendors providing OWL tools for reasoning and construction.

For a careful introduction into OWL and the related concepts we refer to (Antoniou & van Harmelen 2004).

Figure 1 and the subsequent text show an excerpt of a printer ontology given in UML and XML-based OWL syntax, respectively.

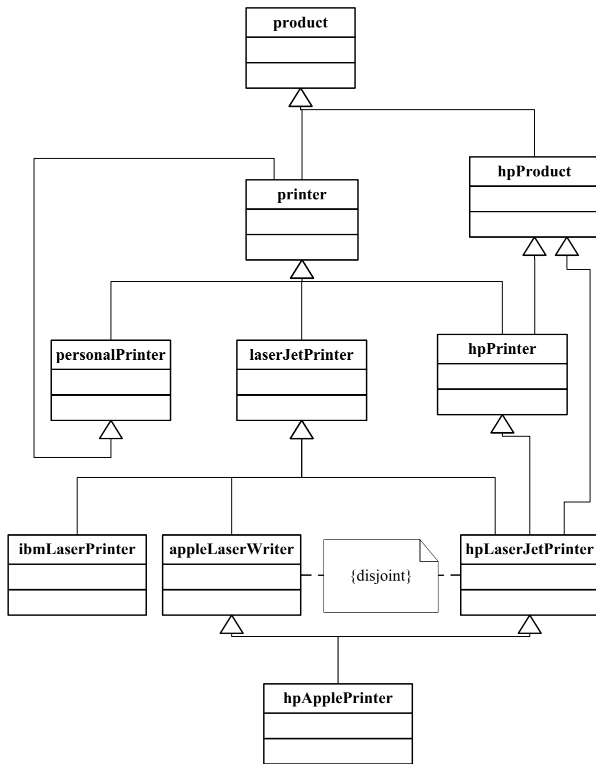

Figure 1: A Printer Ontology in UML

The concepts `laserJetPrinter`, `appleLaserWriter`, and `hpLaserJetPrinter` are defined in the XML-based syntax of OWL as follows:

```
<owl:Class rdf:ID="appleLaserWriter">
   <rdfs:comment>Apple laser writers are laser printers
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="#laserJetPrinter"/>
   <owl:disjointWith rdf:resource="#hpLaserJetPrinter"/>
</owl:Class>

<owl:Class rdf:ID="hpApplePrinter">
   <rdfs:comment>Printers from a joint
    venture of HP and Apple: contradicts the
    disjoint restriction</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#hpLaserJetPrinter"/>
   <rdfs:subClassOf rdf:resource="#apperLaserWriter"/>
</owl:Class>
<owl:Class rdf:ID="laserJetPrinter">
   <rdfs:subClassOf rdf:resource="#printer"/>
</owl:Class>
<owl:Class rdf:ID="hpLaserJetPrinter">
   <rdfs:subClassOf rdf:resource="#laserJetPrinter"/>
   <rdfs:subClassOf rdf:resource="#hpPrinter"/>
   <rdfs:subClassOf rdf:resource="#hpProduct"/>
  <owl:disjointWith rdf:resource="#appleLaserWriter"/>
</owl:Class>
```

Examples of instances for the given printer ontology are as follows:

```
<appleLaserWrite rdf:ID="1000"/>
<appleLaserWrite rdf:ID="1001"/>
<appleLaserWrite rdf:ID="1002"/>
<hpLaserJetPrinter rdf:ID="1003"/>
<hpLaserJetPrinter rdf:ID="1004"/>
<hpLaserJetPrinter rdf:ID="1005"/>
<hpLaserJetPrinter rdf:ID="1006"/>
```

The measures presented in this paper mainly focus on hierarchical relationships between classes; also characteristics of (primitive) properties of classes are investigated. Here, the expressiveness of OWL Lite is sufficient. Some further measures use disjointness relations of concepts that can be formulated with OWL DL.

Declarative Queries in FNQuery

In the following, we consider the syntactic and semantic analysis of OWL-based ontologies using a declarative approach.

Field Notation and FNQuery There exist several XML parsers and term representations for XML documents in PROLOG. We are using SWI-PROLOG and a special XML representation, which we call *field notation*. E.g., we represent the first `owl:Class`-element as follows:

```
'owl:Class':['rdf:ID':'appleLaserWriter']:[
   'rdfs:comment':['Apple laser ...'],
   'rdfs:subClassOf':[
     'rdf:resource':'#laserJetPrinter']:[],
   'owl:disjointWith':[
     'rdf:resource':'#hpLaserJetPrinter']:[] ]
```

In general, an XML element is represented as a term structure `T:As:C`, which we call an FN-triple; `T` is the tag of the element, `As` is a list representing the attribute/value pairs `A:V` of the element, and `C` is a list of FN-triples representing the subelements. We often need to quote tags, attributes or values, e.g., if they start with a capital letter or if they are structures containing ':'.

XML documents in field notation can be queried and modified using the query language FNQuery (Seipel 2002), which consists of the retrieval language FNSELECT, the update language FNUPDATE, and the transformation language FNTRANSFORM. FNQuery has been developed and implemented as a PROLOG library by using suitable, intuitive PROLOG constructs including path expressions similar to

XPath. The main predicate is the binary infix predicate `:=/2` for evaluating path expressions to FN-triples.

FNSelect Given an OWL knowledge base `Owl`, there exists an `isa`-relationship between two classes `C1` and `C2`, if a `subclassOf`-relationship has explicitly been specified between the two classes, or if `C1` has been defined as the intersection of `C2` and some other classes:

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

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

The two cases can be covered using the two rules defining the predicate `isa/3`; further cases could be added nicely. In the first rule the path expression selects a subelement `'owl:Class'`, whose attribute `'rdf:ID'` is `C1`, and then it selects the subelement `'rdfs:subClassOf'`, and assigns the attribute `'rdf:resource'` to `R2`. Note that we have to dereference attributes such as `rdf:resource` and `rdf:about` to make them compatible with `rdf:ID`–attributes.

Similarly, the following rule determines disjointness conditions between classes in an OWL document:

```
disjointWith(Owl, 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).
```

FNUpdate The following update rule can be used for deleting a redundant `subClassOf`-relationship from an OWL knowledge base:

```
delete(isa(C1, C2), Owl1, Owl2) :-
   owl_reference_to_id(R2, C2),
   Owl2 := Owl1 <-> [
      /'owl:Class'::[@'rdf:ID'=C1]
      /'rdfs:subClassOf'::[
         @'rdf:resource'=R2] ].
```

The following rule inserts a comment indicating that there exists a redundant `subClassOf`-relationship in an OWL knowledge base:

```
indicate(isa(C1, C2), Owl1, Owl2) :-
   Owl2 := Owl1 <+> [
      /'owl:Class'::[@'rdf:ID'=C1]
      /'rdfs:comment':[
         'redundant subClassOf: ', C2] ]
```

FNTransform There is also a language similar to XSLT for transforming XML documents in Prolog.

Prolog Rules The following rules derive transitive subclasses of a class based on the `isa`-relation:

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

If the ontology `Owl` is fixed, then we define `isa(C1, C2)` as `isa(Owl, C1, C2)`, and omit the ontology argument.

Taxonomic Errors in Ontologies

According to Gómez–Pérez (Gómez-Pérez 1999) the evaluation of an ontology includes the inspection of its taxonomy, which should be checked for *inconsistency*, *incompleteness*, and *redundancy*, cf. Figure 2.

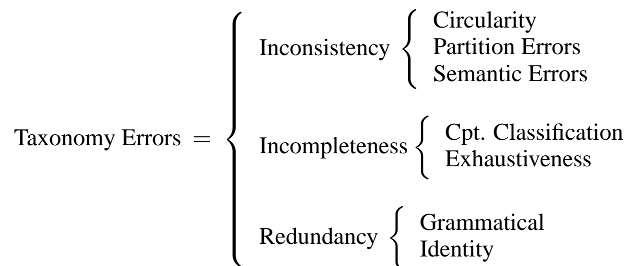

Figure 2: Taxonomy Errors in an Ontology

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.

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 circle of the 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. (Seipel, Baumeister, & Hopfner 2004).

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 modeled as disjoint, since this indicates potentially missing disjointness conditions.

Redundancy

For a given ontology we can detect redundant class/instance definitions or redundant subclass relations (`isa`) in the taxonomy. 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.

Design Anomalies

In addition to the taxonomic errors presented in the previous section there may exist even more subtle anomalies in ontologies. Such anomalies do not directly imply errors in reasoning but point to problematic areas in the ontology that may require modification. Originally, anomalies had been identified and investigated for relational databases. In the last years, software engineering research has coined the term *bad smells* for parts of the source code that do not produce false behavior but are badly designed and should be improved for better maintainability, cf. (Fowler 1999).

The concept of design anomalies and corresponding refactoring methods was transferred to rule-based and case-based knowledge in (Baumeister, Seipel, & Puppe 2004). In this section, we define typical design anomalies in ontologies.

We extend the predicate `subClassOf/2` to `subClassOf/3`, where the third argument contains the list of classes in a chain of `isa`-relationships:

```
subClassOf(C1, C2, [C1, C2]) :-
   isa(C1, C2).
subClassOf(C1, C2, [C1|Cs]) :-
   isa(C1, C), subClassOf(C, C2, Cs).
```

The second rule constructs a path `[C1|Cs]` from `C1` to `C2` recursively by appending `C1` to a path `Cs` - which is a list of classes that has to be constructed before - from `C` to `C2`, if `C1` is a subclass of `C`.

Lazy Concepts Class and property definitions are the basis for instances created for the particular application domain. If a leaf class or property is never used in an application, then we call this a *lazy concept*. Lazy concepts can appear in large manually defined ontologies, e.g., if the lazy concept has been replaced with specialized or generalized concepts during the development phase. A lazy concept is not always a certain candidate for deletion, e.g., the concept should remain in the ontology for preserving the reusability or the standardization of the ontology. Then, the developer has to deal with the trade-off between the reusability of the original and verbose ontology and the simplicity of the ontology design. In FNQuery we define a lazy concept as follows:

```
lazy_concept(C) :-
   not(isa(_, C)), not(_:C).
```

Given a concept C, we encode by X:C that X is an instance of C. The concept is lazy, if it is a leaf in the concept hierarchy (i.e., there exists no subclass of C), and if C does not have any instances. We have used anonymous variables "_" here, since we are not interested in the particular subclasses or instances of C.

Chains of Inheritance Classes can be inherited by other classes using subClassOf relations. Due to the application of restructuring in even larger ontologies, an originally well-designed subClassOf tree may be degenerated such that some parts of the tree represent long paths in which each particular class contains only on inherited child. We call such a path a *chain of inheritance*, and in some cases it may be reasonable to cut the path down to an appropriate number of subClassOf relations. Such chains are detected in FNQuery with the following statements:

```
chain_of_inheritance(Cs, Threshold) :-
   subClassOf(C1, C2, Cs),
   length(Cs, N), N > Threshold,
   not(contains_branch(Cs)).

contains_branch(Cs) :-
   subsequence(Cs, [X, Y]),
   isa(Z, Y), Z \= Y.
```

A path Cs = [C1, ..., X, Y, ..., C2] of isa-relationships from a class C1 to another class C2 contains a branch, if there exists a subsequence [X, Y] of two consecutive classes on the path, such that Y has another subclass Z which is different from X.

Lonely Disjoints In OWL DL (and OWL Full) disjointness of classes can be explicitly defined using the disjointWith restriction. Often siblings of the same parent class are defined to be disjoint. During the development of an ontology, often one sibling class is moved to another point in the hierarchy. However, when using visual ontology editors, e.g., Protégé (Grosso *et al.* 1999) and OntoEdit (Sure, Angele, & Staab 2002), developers may forget to delete the disjointness relation from the moved class (which is often reasonable). Such *lonely disjoints* can yield inconsistency errors and errors in reasoning with the ontology, for which the lonely disjointness relation cannot be identified at first sight. Therefore, it is helpful to report a class as a possible source of anomaly, which is disjoint with other classes that are all located at another level of the hierarchy. If these classes are all at the same point of the hierarchy, then a lonely disjointness is reported. In the case of a correct detection the anomaly can be removed by simply deleting the disjointWith restriction. Lonely disjoints are described in FNQuery as follows:

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

disjoint_with_some_sibling(C) :-
   siblings(C, S),
   disjointWith(C, S).

lonely_disjoint(C, C1-C2) :-
   disjointWith(C1, C2),
   siblings(C1, C2),
   disjointWith(C, C1),
   not(siblings(C, C1)),
   disjointWith(C, C2),
   not(siblings(C, C2)),
   not(disjoint_with_some_sibling(C)).
```

Two classes C1 and C2 are siblings, if they have a common parent class C. A class is a lonely disjoint, if there exist two disjoint siblings C1 and C2, such that C is disjoint with both of them, but C is not a sibling of them, and moreover, C is not disjoint with one of its own siblings. If we don't have multiple inheritance, then the condition not(siblings(C, C2)) is redundant, since it could be derived from siblings(C1, C2) and not(siblings(C, C1)).

Property Clumps For classes additional properties can be defined that either attach primitive information (e.g., strings or integers) to the classes (DataType Properties), or relate the class to other classes (Object Properties). If a collection of such properties is repeatedly included in several class definitions, then we call this a *property clump*. The design of the ontology may be improved by defining an abstract concept aggregating this clump and replacing the clump by the new concept. E.g., from an OWL DataType Property

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

the property information can be extracted in FNQuery:

```
property(Owl, Property, Domain, Range) :-
   P := Owl/'owl:DatatypeProperty'::[
      @'rdf:ID'=Property],
   D := P/'rdfs:domain'@'rdf:resource',
   Range := P/'rdfs:range'@'rdf:resource',
   owl_reference_to_id(D, Domain).
```

From property/3 we can derive all properties for a given class Domain. If the set Properties of common

properties of a given set Ds of classes is larger than a given threshold T, then we could extract the properties and form a new class, from which the classes in Ds inherit:

```
properties(Owl, Domain, Properties) :-
   findall( Prop:Range,
      property(Owl, Prop, Domain, Range),
      Properties ).

property_clump(Owl, Ds, T, Properties) :-
   maplist( properties(Owl),
      Ds, Ps ),
   intersection(Ps, Properties),
   length(Properties, N), N > T.
```

Discussion

In recent years, ontologies have been a major subject of research for building intelligent systems, especially in the context of the semantic web. In this paper, we have presented declarative methods for evaluating ontologies; we have motivated that our declarative approach using FNQuery can be applied for implementing standard queries, such as the evaluation of taxonomic errors, but also for novel evaluation measures considering design anomalies in ontologies. Since OWL is commonly represented in XML the question arises, whether OWL documents can be processed using standard XML tools like XQUERY and XSLT. However, due to their generality such systems can only handle tasks based on the syntax of the XML document, but not on its semantics. Thus, it is very difficult to define queries and transformations that take the defined data model of OWL into account. For example, a query for all instances of `printer` should not only consider the actual instances of the class `printer`, but also all instances of its subclasses, e.g., `laserJetPrinter`, `hpPrinter`. In FNQuery the particular semantics of the OWL data model can be easily defined in order to allow for deductive queries.

Ontology development tools and tool suites such as Protégé and OntoEdit have been successfully applied 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 (Sure, Angele, & Staab 2002) offers the plugin OntoAnalyser for the analysis of ontologies w.r.t. the conformity of properties and the consistency of the ontology. The features of OntoAnalyser can be configured by specialized rules steering the analysis process. Analogously, the web-based ontology development workbench WebODE includes the plugin ODEClean supporting the OntoClean methodology. OntoClean (Guarino & Welty 2002) allows for the 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 as well.

The described evaluation measures are a starting point for the application of refactoring methods, that provide a structured approach for improving the design of ontologies. The application of refactoring methods is driven by the detection of design anomalies; each identified design anomaly corresponds to a sequence of refactorings aiming at removing the anomaly. Since FNQuery can be used not only for formulating 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. The identification and formal description of appropriate refactorings together with their declarative implementation using FNQuery is an issue for future work.

References

[Antoniou & van Harmelen 2004] Antoniou, G., and van Harmelen, F. 2004. *A Semantic Web Primer*. MIT Press.

[Baumeister, Seipel, & Puppe 2004] Baumeister, J.; Seipel, D.; and Puppe, F. 2004. Refactoring Methods for Knowledge Bases. In *Engineering Knowledge in the Age of the Semantic Web: 14th International Conference, EKAW 2004, LNAI 3257*, 157–171. Springer.

[Berners-Lee, Hendler, & Lassila 2001] Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The Semantic Web. *Scientific American*.

[Fowler 1999] Fowler, M. 1999. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.

[Gómez-Pérez 1996] Gómez-Pérez, A. 1996. Towards a Framework to Verify Knowledge Sharing Technology. *Expert Systems with Applications* 11(4).

[Gómez-Pérez 1999] Gómez-Pérez, A. 1999. Evaluation of Taxonomic Knowledge on Ontologies and Knowledge-Based Systems. In *Proceedings of the 12th International Workshop on Knowledge Acquisition, Modeling and Management (KAW 1999)*.

[Grosso *et al.* 1999] Grosso, W.; Eriksson, H.; Fergerson, R. W.; Gennari, J. H.; Tu, S. W.; and Musen, M. 1999. Knowledge Modeling at the Millennium – The Design and Evolution of Protégé-2000. In *Proceedings of the 12th International Workshop on Knowledge Acquisition, Modeling and Management (KAW 1999)*.

[Guarino & Welty 2002] Guarino, N., and Welty, C. 2002. Evaluating Ontological Decisions with OntoClean. *Communications of the ACM* 45(2).

[Seipel, Baumeister, & Hopfner 2004] Seipel, D.; Baumeister, J.; and Hopfner, M. 2004. Declaratively Querying and Visualizing Knowledge Bases in XML. In *Proceedings of the 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004), LNAI 3392*, 18–34. Springer.

[Seipel 2002] Seipel, D. 2002. Processing XML Documents in Prolog. In *Proceedings of Workshop on Logic Programming (WLP 2002)*.

[Staab & Studer 2004] Staab, S., and Studer, R., eds. 2004. *Handbook on Ontologies*. International Handbooks on Information Systems. Berlin: Springer.

[Sure, Angele, & Staab 2002] Sure, Y.; Angele, J.; and Staab, S. 2002. OntoEdit: Guiding Ontology Development by Methodology and Inferencing. In *Proceedings Intl. Conf. on Ontologies, Databases and Applications of Semantics for Large Scale Information Systems, ODBASE 2002, LNCS 2519*. Springer.