Owlifier: Creating OWL-DL Ontologies from Simple Spreadsheet-Based Knowledge Descriptions☆

Shawn Bowers[a,*], Joshua S. Madin[b], Mark P. Schildhauer[c]

[a] *UC Davis Genome Center*
[b] *Dept. of Biological Sciences, Macquarie University, Australia*
[c] *National Center for Ecological Analysis and Synthesis, UC Santa Barbara*

Abstract

Discovery and integration of data is important in many ecological studies, especially those that concern broad-scale ecological questions. Data discovery and integration is often a difficult and time-consuming task for researchers, which is due in part to the use of informal, ambiguous, and sometimes inconsistent terms for describing data content. Ontologies offer a solution to this problem by providing consistent definitions of ecological concepts that in turn can be used to annotate, relate, and search for data sets. However, unlike in molecular biology or biomedicine, few ontology development efforts exist within ecology. Ontology development often requires considerable expertise in ontology languages and development tools, which is often a barrier for ontology creation in ecology. In this paper we describe an approach for ontology creation that allows ecologists to use common spreadsheet tools to describe different aspects of an ontology. We present conventions for creating, relating, and constraining concepts through spreadsheets, and provide software tools for converting these ontologies into equivalent OWL-DL representations. We also consider inverse translations, i.e., to convert ontologies represented using OWL-DL into our spreadsheet format. Our approach allows large lists of terms to be easily related and organized into concept hierarchies, and generally provides a more intuitive and natural interface for ontology development by ecologists.

1. Introduction

Within the fields of molecular biology and biomedicine considerable effort has gone into developing ontologies for improving data discovery and integration [1, 2]. While similar benefits can be obtained for ecological data, far fewer efforts exist to develop broad and consistent terminologies within ecology [9, 12]. The use of formal ontologies can significantly enhance metadata descriptions of ecological data. For instance, annotating data with ontology terms can both help users interpret data as well as enable advanced capabilities for data discovery

*Corresponding author

and integration, e.g., by exploiting subsumption and part-of hierarchies as well as more formal constraints such as cardinality restrictions on properties and term equivalence [9].

Efforts to engage scientists in the development of ontologies typically leverage the W3C Web Ontology Language (OWL) [17] as a standard XML syntax for representing and sharing ontologies. A key advantage of OWL is that it is supported by a wide range of generic tools, including editors [8, 7], reasoning systems [16, 18], query languages [13, 11], and storage technologies [5, 4]. Most of these tools, however, are primarily targeted at experts in knowledge engineering and software development familiar with the underlying description logic semantics of OWL-DL [6]. This is especially true with ontology editors (such as Protege, SWOOP, etc.), which allow for very detailed ontology specifications, but at the same time require considerable understanding of the underlying ontology formalisms and syntax. We see the lack of suitable ontology editing tools for scientists without expertise in knowledge representation as one of the major barriers for more wide-scale adoption of ontologies in ecology.

This paper presents a novel approach for ontology creation that aims at being more intuitive for ecologists and that can be used to rapidly construct large ontologies for describing scientific data. Our approach is to allow scientists to use common spreadsheet-based tools to describe, in an intuitive way, different aspects of an ontology, and then to take these descriptions and convert them into full-fledged OWL ontologies using a software application called owlifier. An owlifier spreadsheet consists of a set of *blocks* that have a predefined template structure for users to fill in. Each non-empty row in an owlifier table constitutes a block. Each block defines different aspects of an ontology including ontology classes, subclasses, synonyms, and properties. We also provide blocks for plain-text descriptions of classes and properties, and for referencing one or more existing ontologies (e.g., to extend an existing ontology or to define ontology articulations). Blocks can be sparse (inheriting from previous blocks), which can further simplify the creation of large ontologies.

While not as expressive as OWL-DL, our approach can produce ontology structures that are essential for improved data discovery and integration [10]. Just as important, because spreadsheet tools are frequently used by ecologists to store and analyze data, owlifier can provide ecologists with a familiar and accessible user interface for ontology creation. This approach also leverages the ability of spreadsheet tools to organize and manipulate tabular data, e.g., allowing users to rapidly construct class hierarchies from long lists of keywords. In this way, an ecologist can easily construct (or import) a set of terms, and then incrementally organize these into class hierarchies, properties, and constraints. In initial experiments with ecologists and evolutionary biologists studying trait data, we found that owlifier enabled them to quickly and easily comprehend and construct useful ontologies.

The rest of this paper is organized as follows. In Section 2 we describe the basic syntax and semantics of owlifier. We define blocks that support a large subset of OWL-DL and that also generally follow the ontology creation guidelines defined in [15]. We also simplify certain aspects of ontology creation

using OWL-DL, e.g., by assuming classes are disjoint by default (unless specified otherwise) and supporting sufficient restrictions [15]. In Section 3 we describe additional characteristics of owlifier and discuss issues with respect to classification and reasoning. In Section 4 we briefly describe the owlifier implementation, and conlcude in Section 5 with related and future work. In general, the goal of owlifier is not to support all constructs in OWL-DL, but instead to provide a higher-level ontology syntax (via spreadsheet blocks) that is easy for ecologists to use and understand while also providing the necessary constructs for developing typical ecological ontologies. By compiling owlifier to OWL-DL, we also allow for experts to refine and extend the ontology using more advanced ontology editing tools if necessary.

2. The Syntax and Semantics of Owlifier

As described above, an owlifier table defines an OWL-DL [17] ontology through a set of *blocks* representing one or more ontology definitions. Each non-empty row in an owlifier table corresponds to a block. The type of the block is given in the first column of the row. We assume that if any properties or classes used in a block are not imported from another ontology, then they are to be added to the ontology being specified by the owlifier table (i.e., the "current" ontology). In general, we name blocks according to the terms used in [3, 10] as opposed to the names used for corresponding constructs in OWL-DL. This choice of block names helps to simplify terminology (e.g., we use "relation" below instead of "object property"), allows owlifier to easilty generate ontologies that extend the observational model of [3, 10], and avoids confusion with established terms commonly used within ecology (e.g., "class").

Import Blocks. Import blocks assign namespace labels to external ontologies. Each external ontology is imported into the current ontology. We refer to the ontologies of import blocks as *imported ontologies*. Using import blocks, classes and properties of imported ontologies can be used within other blocks of an owlifier table. Rows containing import blocks take the form

import	n	u

where n is a namespace label and u is an OWL ontology URI. Classes and properties from imported ontologies are referenced by prefixing the namespace label n to the corresponding class or property name in the normal way. As an example, the following block imports the SWEET "Earth Realm" ontology [14]

```
import sweet http://sweet.jpl.nasa.gov/ontology/earthrealm.owl
```

With this import block the class denoting Marine Ecosystems (a class defined in the SWEET ontology) can be referred to from within an owlifier table using the expression sweet:MarineEcosystem. Because this class refers to a class in another ontology, we refer to it as an *imported class*.

Entity Blocks. Entity blocks are the primary blocks used to define ontologies. An entity block introduces new OWL classes and specifies subclass relationships. Imported classes may also be used within entity blocks by prefixing class names with namespace labels (as described above). Rows containing entity blocks take the form

entity	c_1	c_2	...	c_n	$(n \geq 1)$

where each class c_i is asserted in the current ontology to subsume c_{i+1}, for $1 \leq i < n$. That is, each c_i in an entity block induces the description logic axiom $c_{i+1} \sqsubseteq c_i$. If both c_i and c_{i+1} are imported classes, we say that the block defines an "articulation" (i.e., mapping) between the two classes. The following entity block defines a simple subclass hierarchy.

entity PhysicalFeature AquaticPhysicalFeature River

This block states that Physical Feature, Aquatic Physical Feature, and River are classes; River is a subclass of Aquatic Physical Feature; and Aquatic Physical Feature is a subclass of Physical Feature. The following entity block introduces a new class via an imported class.

entity sweet:MarineEcosystem IntertidalEcosystem

This block states that Intertidal Ecosystem is a subclass of the Marine Ecosystem class imported from the SWEET ontology. Similarly, assuming "marine" denotes an existing ontology of marine ecosystem concepts, the following block defines a simple class articulation.

entity sweet:MarineEcosystem marine:DeapSeaEcosystem

This block states that the Deap Sea Ecosystem class of the marine ontology is a subclass of the Marine Ecosystem class of the SWEET ontology (thus defining a mapping between these two ontologies).

Synonym Blocks. Synonym blocks define equivalence relationships between ontology classes. Rows containing synonym blocks take the form

synonym	c_1	c_2	...	c_n	$(n \geq 2)$

where each class c_i is equivalent to class c_{i+1} in the current ontology, for $1 \leq i < n$. That is, each c_i in a synonym block induces a description logic axiom of the form $c_i \equiv c_{i+1}$. The following synonym block creates a simple equivalence relationship.

synonym Maize Corn

This block states that the Maize and Corn classes are synonyms (equivalent classes). Similar to entity blocks, synonym blocks often contain imported classes for extending existing ontologies or defining ontology mappings.

Overlap Blocks. Except in certain situations (described further in Section 3), classes are generally assumed to be disjoint in owlifier. Overlap blocks explicitly relax this assumption by stating that a given set of classes may have overlapping instances. Rows containing overlap blocks take the form

overlap	c_1	c_2	...	c_n	$(n \geq 2)$

where each class c_i is allowed to share instances with each class c_j, for $1 \leq i, j \leq n$. That is, a given c_i and c_j in an overlap block are not defined to be disjoint classes in the current ontology. As an example, consider the following entity blocks that define the classes Estuary, Lagoon, and Marsh as subclasses of Ecological Habitats.

```
entity EcologicalHabitat Estuary
entity EcologicalHabitat Lagoon
entity EcologicalHabitat Marsh
```

Given only these blocks, owlifier treats Estuary, Lagoon, and Marsh as disjoint classes. To relax this assumption and allow, e.g., types of Lagoons to also be types of Estuaries, we explicitly add the following overlap block

```
overlap Estuary Lagoon
```

In general, overlap blocks are rarely used but provide a mechanism to override the default behavior of owlifier in asserting disjoint classes.

Relationship Blocks. Relationship blocks define *required* class object properties. An object property within OWL is a property defined between two class instances. Rows containing relationship blocks take the form

relationship	p	c_1	c_2	...	c_n	$(n \geq 2)$

where p is an object property and each c is a class. For every class c_i, the relationship block induces the description logic axiom $c_i \sqsubseteq \exists p.c_{i+1}$ stating that each instance of c_i is p-related to some instance of c_{i+1}, for $1 \leq i < n$. For example, the following block states that instances of the class California Voles live in Grassy Areas.

```
relationship livesIn CaliforniaVole GrassyArea
```

In some cases, a particular property can apply to a sequence of classes, and for convenience, each such class can be specified in owlifier using a single block. For example, consider the following block.

```
relationship directlyBelow Hypolimnion Thermocline Epilimnion
```

This block states that, e.g., within a thermally stratified lake, the Hypolimnion layer is directly below the Thermocline layer, and the Thermocline layer is directly below the Epilimnion layer.

Transitive Blocks. Transitive blocks are special cases of relationship blocks where the object property is asserted to be transitive. If a property p is declared to be transitive, whenever p relates an individual x to an individual y, and an individual y to an individual z, then p is also assumed to relate x to z. Rows containing transitive blocks take the form

transitive	p	c_1	c_2	...	c_n	$(n \geq 2)$

where p is an object property and each c is a class. The following block is a simple example of a transitive relationship.

```
transitive hasPart Body Head Eye Retina
```

This block states that every instance of the class Body has a Head as a part, every instance of the class Head has an Eye as a part, and every instance of the class Eye has a Retina as a part. Moreover, because the hasPart property above is defined to be transitive, it is possible to infer that every instance of Body also has an Eye and a Retina as a part through the inherited relationship restrictions Body $\sqsubseteq \exists$hasPart.Head, Head $\sqsubseteq \exists$hasPart.Eye, and Eye $\sqsubseteq \exists$hasPart.Retina.

Cardinality Blocks. Cardinality blocks are also similar to relationship blocks. We consider three types of cardinality blocks for defining minimum, maximum, and exact cardinality restrictions. A minimum block states the smallest number of properties p to distinct individuals that an individual of a class may have. Rows containing minimum blocks take the form

min	p	m	c_1	c_2	...	c_n	$(n \geq 2)$

where m is the minimum number of properties p that instances of class c_i must have to instances of concept c_{i+1}, for $1 \leq i < n$. For each class c_i, a minimum cardinality block induces the description logic axiom $c_i \sqsubseteq (\leq m)\ p.c_{i+1}$ stating that each instance of c_i must be p-related to at least m unique instances of c_{i+1}. The following two blocks demonstrate simple minimum cardinality constraints.

```
min hasPart 1 Body Head Nose
min hasPart 2 Head Eye
```

The first block states that instances of the class Body have at least one Head as a part, which in turn have at least one Nose as a part.[1] The second block states that instances of the class Head have at least two Eyes as parts.

A maximum block states the largest number of properties p to distinct individuals that an individual of a class may have. Rows containing maximum blocks take the form

max	p	m	c_1	c_2	...	c_n	$(n \geq 2)$

[1] Cardinality restrictions ensuring participation to at least one property are typically not given through minimum cardinality blocks since they are also implied by relationship blocks.

where m is the maximum number of properties p that instances of concept c_i may have to instances of concept c_{i+1}, for $1 \leq i < n$. For each class c_i, a maximum cardinality block induces the description logic axiom $c_i \sqsubseteq (\geq m)\ p.c_{i+1}$ stating that each instance of c_i may be p-related to at most m unique instances of c_{i+1}. The following two blocks demonstrate simple maximum cardinality constraints.

```
max hasPart 1 Body Head Nose
max hasPart 2 Head Eye
```

The first block states that instances of the class Body have at most one Head as a part, which in turn has at most one Nose as a part. The second block states that instances of the class Head have at most two Eyes as parts.

An exact block ensures both a minimum and maximum number m of properties p to distinct individuals that an individual of a class must have. Rows containing exact blocks take the form

exact	p	m	c_1	c_2	...	c_n	$(n \geq 2)$

where m is the number of properties p that instances of concept c_i must have to instances of concept c_{i+1}, for $1 \leq i < n$. For each class c_i, an exact block induces the description logic axiom $c_i \sqsubseteq (= m)\ p.c_{i+1}$ stating that each instance of c_i must be p-related to m unique instances of c_{i+1}.

Inverse Blocks. Inverse blocks state that two object properties are inverses of each other. If p_1 and p_2 are defined to be inverse properties, whenever p_1 relates an individual x to an individual y then p_2 (as the inverse of p_1) is assumed to relate y to x. Rows containing inverse blocks take the form

inverse	p_1	p_2

where p_1 and p_2 are object properties. A common example of inverse properties are hasPart and partOf, i.e., if an individual x has an individual y as a part, then y is by definition a part of x.

Sufficient Blocks. Sufficient blocks are similar to synonym blocks in that they state equivalences between classes. A sufficient block serves as a modifier to entity blocks, relationship blocks (including transitive blocks), and cardinality blocks, modifying the associated description logic axioms by using equivalence ($\equiv$) in place of sublass ($\sqsubseteq$). For example the block

```
sufficient relationship hasPart Mammal Hair
```

induces the description logic axiom Mammal $\equiv \exists$hasPart.Hair stating that *any* individual that has Hair as a part is a Mammal. Additionally, we allow relationship blocks within a sufficient block to be extended with not to state that absense of the property is a defining characteristic of the class. For example the following blocks

```
sufficient entity Mammal Eutheria
sufficient relationship not hasPart Eutheria EpipubicBone
```

induces the axiom Eutheria $\equiv$ Mammal $\sqcap \neg\exists$hasPart.EpipubicBone stating that any Mammal that does not have an Epipubic Bone as a part is a Eutheria. Note also that multiple sufficient blocks for a particular class result in a single axiom in which each constraint is combined via conjunction ($\sqcap$).

Comment Blocks. Comment blocks provide a mechanisms to add plain-text comments to owlifier tables. A description block attaches a plain-text comment to classes and properties. Rows containing description blocks take the form

description	c or p	s

where the string s is associated as a comment to the class c or property p. A note block allows attaches comments to the current ontology. Rows containing note blocks take the form

note	s

where s is a comment string.

3. Additional Features of Owlifier

In this section, we briefly describe some of the additional features of owlifier, specifically focusing on the use of disjoint classes, owlifier reasoning, and additional block syntax.

3.1. Disjoint Class Inference

OWL is based on the open world assumption, which can lead to a number of ontology development "pitfalls" for those new to the language [17, 15]. One example is in the creation of disjoint classes. In particular, unless explicitly stated, distinct classes within an OWL-DL ontology are never assumed to be disjoint. However, in many ontologies a large number of classes are typically defined as being disjoint (e.g., sibling classes), and stating these disjoint constraints is often time-consuming since each pair of classes must be given an explicit disjoint assertion. Editors such as Protege [8] provide shortcuts via the user interface to create specific sets of disjoint assertions, e.g., by allowing a user to define all children of a particular class as disjoint. In general, however, many users expect such classes to be disjoint by default [15] and this expectation often leads to modeling errors.

Alternatively, the default assumption in owlifier is that distinct classes are disjoint. Specifically, as an owlifier table is converted to an OWL-DL ontology, the system analyzes the class hierarchy structure and identifies pairs of classes that are: (1) not related via subclass relations (either direct or indirect subclasses); (2) not defined as synonyms; and (3) not explicitly defined to overlap via an overlap block. Each such pair of classes is then asserted by owlifier in the resulting ontology as being disjoint. As described in [15], undeclared disjoint classes are a common problem in ontology development using OWL-DL and often limit the utility of reasoning systems (by limiting the inferences that can be

obtained). The approach employed in owlifier for handling disjoint classes makes the common expectations of users the default case, which in general should lead to a more intuitive ontology editing environment and an overall fewer number of modeling mistakes.

3.2. Reasoning in Owlifier

Blocks in owlifier are unambiguous, i.e., for every owlifier block (or set of blocks in the case of sufficient blocks) there is a well-defined set of corresponding OWL-DL axioms. This property is important because it implies that reasoning can be performed over ontologies defined in owlifier using standard OWL-DL reasoners. We use this capability in our current owlifier implementation (described further in Section 4) to verify ontologies defined using owlifier and report possible errors to users.

In general, new axioms are inferred from an owlifier ontology primarily from the use of synonym blocks, sufficient blocks, and transitive blocks (whose inferences are described in the previous section). For instance, let A, B, and C be classes and P be an object property. From an axiom $A \equiv B$ generated from a synonym block, and an axiom $B \sqsubseteq \exists P.C$ generated from a relationship block, the axiom $A \sqsubseteq \exists P.B$ would be inferred. Thus, the axioms of a particular class are "inherited" by all of the synonyms of the class. Similarly, from an axiom $A \equiv \exists P.C$ generated from a sufficent block, and an axiom $B \sqsubseteq \exists P.C$ generated from a relationship block, the axiom $B \sqsubseteq A$ would be inferred. For both the case of synonym and sufficient blocks, the use of equivalence permits a number of additional inferences to be made via an OWL-DL reasoner.

As described in [15] additional reasoning can occur within OWL-DL ontologies when domain and range axioms are providedq (as well as, e.g., property closure axioms). We explicitly do not consider these constraints in the current version of owlifier because they also often result in modeling errors for inexperienced OWL-DL users [15]. Instead, we adopt the more traditional description-logic approaches that do not have explicit domain and range axioms. Although not currently supported in owlifier, domain and range constraints could be inferred from given relationship blocks as well as property closure axioms.

3.3. Sparse Blocks

To help simplify the creation of class hierarchies and property sequences (including transitive, cardinality, and sufficient blocks), we allow for "sparse" blocks that inherit missing information from their closest proceeding block. For instance the following entity blocks

```
entity EcologicalHabitat Estuary Bay
entity EcologicalHabitat Estuary Fjord
entity EcologicalHabitat Marsh   TidalMarsh
entity EcologicalHabitat Marsh   SaltMarsh
```

can be equivalently represented in owlifier using the following sparse blocks.

entity	EcologicalHabitat	Estuary
		Bay
		Fjord
	Marsh	TidalMarsh
		SaltMarsh

In general, the use of sparse blocks simplifies ontology creation by not requiring users to enter every (redundant, leading) field to be explicitly given, which in turn can simplify the overall layout of the ontology within a spreadsheet. Additionaly, owlifier does not place constraints on the order of blocks within a spreadsheet. Classes also do not need to explicitly be defined within an entity block, e.g., classes without corresponding entity blocks can be introduced simply through synonym and relationship blocks. This typically occurs when a particular class does not participate as a subclass or superclass of another class in the current ontology. Overall, one of our goals is for owlifier to allow minimal information to be entered by limitting redundancy as well as by supporting a range of inferences, while at the same time reducing the number of errors commonly made in ontology development by non-experts.

4. The OwlifierApplication

Figure 1 shows the general architecture of the owlifier application. A scientist or researcher first creates a spreadsheet containing a set of ontology definitions using owlifier conventions. The spreadsheet is then exported as a plain-text file containing the owlifier table, e.g., using a CSV or tab-delimited format. The owlifier table is then sent to the owlifier application, which performs a number of steps that include: (i) parsing the file; (ii) generating the corresponding OWL-DL representation (e.g., in the current implementation, the Jena API is used [5]); (iii) validating the ontology and ensuring it is consistent via an OWL-DL reasoner (e.g., the current implementation uses Pellet [16]); and (iv) outputing the corresponding OWL-DL file. The resulting OWL file can then be refined and extended via existing OWL tools (e.g., Protege or SWOOP). It is possible that the refined ontology is converted back to a corresponding owlifier representation (shown using dashed lines in Figure 1). While simple to perform, this last conversion is not information preserving since not all OWL-DL constructs are represented through owlifier blocks.

The current implementation of the owlifier application is written in Java and supports a subset of the blocks defined in Section 2. In particular, we are currently extending owlifier to support cardinality and sufficient blocks as well as to perform the above conversion from OWL-DL files to corresponding owlifier blocks. As noted above, the current implementation, although it does not support all of the blocks defined here, is being used within a project focused on supporting trait data and many of the extensions outlined here are based on these experiences. In addition, owlifier has been used to construct term hierarchies from large sets of keywords (harvested from existing metadata documents) as well as to define articulations into the observation ontology defined in [10].

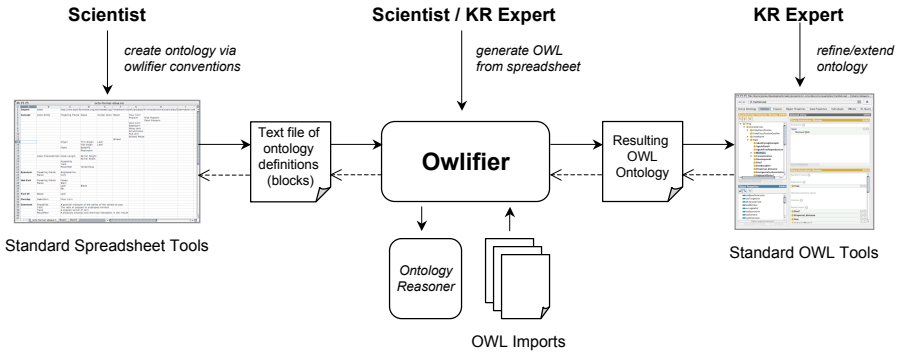

Figure 1: The basic owlifier application and relation to other technologies.

5. Conclusion

related work: similar to dave's work on "global taxonomic constraints", which assume certain structures of hierarchies, e.g., disjoint siblings. protege approach/plugin

future work: the ongoing extensions described above, support seamless conversion into and out of owlifier, e.g., if someone extends owlifier ontology, reapply extensions after new owlifier changes. Also, support translation to oboe. Additional testing and evaulation.

References

[1] M. Ashburner, *et al*. Gene ontology: tool for the unification of biology. *Nat. Genet.*, 25:25–29, 2000.

[2] J. Bard and S. Rhee. Ontologies in biology: design, applications, and future challenges. *Nat. Rev. Genet.*, 5:213–221, 2004.

[3] S. Bowers, J. S. Madin, and M. P. Schildhauer. A conceptual modeling framework for expressing observational data semantics. In *International Conference on Conceptual Modeling (ER)*, volume 5231 of *Lecture Notes in Computer Science*, pages 41–54, 2008.

[4] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference (ISWC)*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.

[5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *International Conference on the World Wide Web (WWW)*, pages 74–83. ACM, 2004.

[6] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2008. in press.

[7] A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca-Grau, and J. Hendler. Swoop: A 'web' ontology editing browser. *Journal of Web Semantics*, 4(2), 2005.

[8] H. Knublauch, M. A. Musen, and A. L. Rector. Editing description logic ontologies with the Protégé OWL Plugin. In *International Workshop on Description Logics (DL)*, volume 104 of *CEUR Workshop Proceedings*, 2004.

[9] J. Madin, S. Bowers, M. Schildhauer, and M. Jones. Advancing ecological research with ontologies. *Trends in Eco. and Evol.*, 23(3):159–168, 2008.

[10] J. Madin, S. Bowers, M. Schildhauer, S. Krivov, D. Pennington, and F. Villa. An ontology for describing and synthesizing ecological observation data. *Ecological Informatics*, 2(3):279–296, 2007.

[11] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *J. Web Sem.*, 3(1):41–60, 2005.

[12] C. Parr and M. Cummings. Data sharing in ecology and evolution. *Trends Ecol. Evol.*, 20:362–363, 2004.

[13] E. Prudhommeaux and A. Seaborne, editors. *SPARQL Query Language for RDF*. W3C. World Wide Web Consortium (W3C), January 2008.

[14] R. Raskin. Semantic web for earth and environmental terminology (sweet). http://sweet.jpl.nasa.gov/.

[15] A. L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. In *EKAW*, pages 63–81, 2004.

[16] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[17] M. K. Smith, C. Welty, and D. L. McGuinness, editors. *OWL Web Ontology Language Guide*. W3C. World Wide Web Consortium (W3C), February 2004.

[18] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297, 2006.