UML Style Graphical Notation and Editor for OWL 2

Jānis Bārzdiņš, Guntis Bārzdiņš, Kārlis Čerāns,
Renārs Liepiņš, Artūrs Sproģis

Institute of Mathematics and Computer Science, University of Latvia,
Raina blvd. 29, LV-1459, Riga, Latvia
Janis.Barzdins@lumii.lv, Guntis.Barzdins@lumii.lv, Karlis.Cerans@lumii.lv
Renars.Liepins@lumii.lv, Arturs.Sprogis@lumii.lv

Abstract. OWL is becoming the most widely used knowledge representation language. It has several textual notations but no standard graphical notation apart from verbose ODM UML. We propose an extension to UML class diagrams (heavyweight extension) that allows a compact OWL visualization. The compactness is achieved through the native power of UML class diagrams extended with optional Manchester encoding for class expressions thus largely eliminating the need for explicit anonymous class visualization. To use UML class diagram notation we had to modify its semantics to support Open World Assumption that is central to OWL. We have implemented the proposed compact visualization for OWL 2 in a UML style graphical editor. The editor contains a rich set of graphical layout algorithms for automatic ontology visualization, search facilities, zooming, graphical refactoring and interoperability with Protégé 4.

Keywords: OWL, UML class diagram, visualization.

1 Introduction

OWL [1] is gradually becoming the most widely used knowledge representation language and has been successfully deployed in a number of applications. Due to formal semantics and availability of reasoners for OWL (see e.g. Pellet [2] or Fact++ [3]), it is gaining popularity also in the software engineering community. However, the readability (the ability to perceive ontologies by humans) is crucial for wider application of OWL.

We propose a novel OWL visualization based on UML [4, 5] class diagram notation. In our opinion the most important feature for achieving readable graphical OWL notation is its maximum compactness. We achieve it by exploiting the native power of UML and using its notation as far as possible. Furthermore, many software engineers are already familiar with UML notation and use it to model data; it may be

expected that this familiarity would enable them to easily adopt the new formalism we propose.

Although UML and OWL have similar concepts (see e.g. [6] Chapter 16 for details), the UML notation cannot be used as is, since some OWL constructs have no equivalent in UML. Therefore an extension of UML with additional symbols and textual expressions is necessary. In addition, both languages have different semantics and it has to be reflected in the usage of the UML notation as well (for instance, UML is based on a "closed world" assumption, while OWL assumes the "open world"). In this paper we explain the extended UML notation by means of metamodel and clarify the interpretation of semantics for the proposed notation.

A number of other solutions have been proposed for graphical UML-style representation of OWL ontologies [6, 7, 8]; the most notable is ODM (see [6] Chapter 14) that defines a UML profile for OWL. The main advantage of ODM approach is the possibility to use existing UML tools for ontology modeling. Meanwhile the price for this compatibility is more verbose notation that does not facilitate comprehensibility.

To make our notation usable in practice we have implemented it in a graphical editor (OWLGrEd) complete with a number of features to ease ontology creation and exploration. In addition we provide interoperability with Protégé [9] to exchange ontologies between both editors. The latest version of the editor can be downloaded from http://OWLGrEd.lumii.lv.

2 Using UML to visually represent OWL ontologies

Despite the semantic differences between the UML and OWL modeling approaches, UML class diagrams can be used to present the core features of OWL ontologies – the OWL classes (presented as UML classes), OWL object properties (typically presented as associations in the UML diagram) and OWL datatype properties (typically presented as attributes in the UML class diagrams). Thus we will organize our explanation in two steps: first step will be the core part of our notation that is a true subset of UML 2.1 class diagram notation [4, 5] (this chapter); and the second step will cover the extension part that contains OWL 2.0 [10] specific features (chapter 3). The explanation is based on the formal metamodel (package diagram shown in Figure 1) that reflects the same structure: the full metamodel includes package UMLOWLCore corresponding to the metamodel part that is a true subset of UML 2.1 class diagrams (expanded view in Fig. 2) and package UMLOWLCoreExtension containing OWL specific features (expanded view in Fig. 4).

UMLOWLCore includes only those UML class diagram features, which have direct one-to-one equivalents in OWL. For example, UML n-ary associations are not included in UMLOWLCore, because their reduction into OWL requires introduction of multiple intermediate classes and properties [6 Chapter 16].

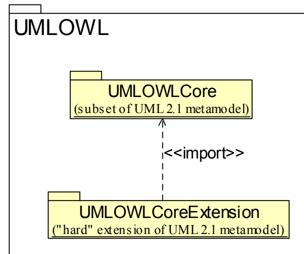


Fig. 1 UMLOWL packages structure

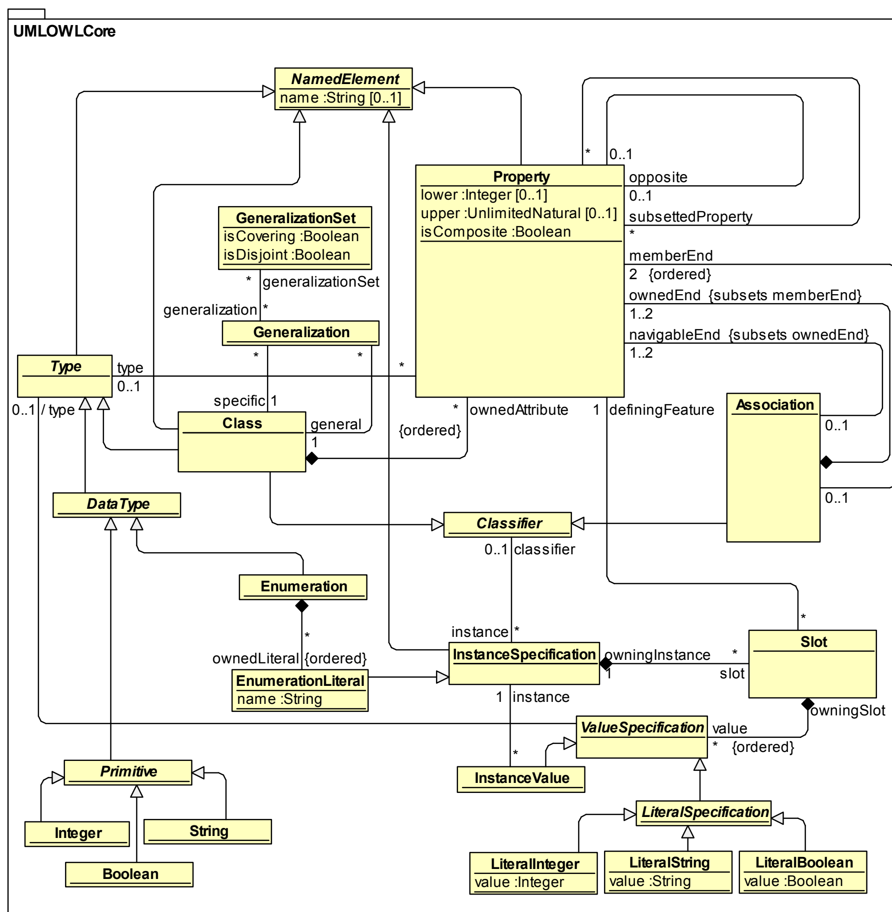


Fig. 2 UMLOWLCore metamodel

Figure 3 shows an example of mini-university ontology in our proposed UML notation, as well as in a textual form in OWL Functional syntax [11] notation. This example uses only UMLOWLCore.

Here we define the details of mappings between core OWL structures and UML class diagrams. UML classes denote OWL classes, while UML properties denote OWL object properties and OWL datatype properties (typically, UML properties that are included in associations denote OWL object properties and the UML properties that are depicted in attribute notation denote OWL datatype properties; however, other combinations of UML properties used for denoting OWL properties are allowed as well). The domain for an OWL property is defined to be the class where UML *ownedAttribute* property is connected to. The *type* of a UML property describes the OWL property's range.

The UML generalizations are used to denote *subclassOf* relation in OWL. We note that it is possible to use *complete* and *disjoint* tags with UML generalizations, and these have a well defined semantics in OWL: for a UML generalization set comprising *subClassOf(B,A)*, *subClassOf(C,A)* and *subClassOf(D,A)* relations, a *disjoint* tag would add an OWL axiom *disjointClasses(B,C,D)* and a *complete* tag would add an OWL axiom *subClassOf(A,unionOf(B,C,D))*.

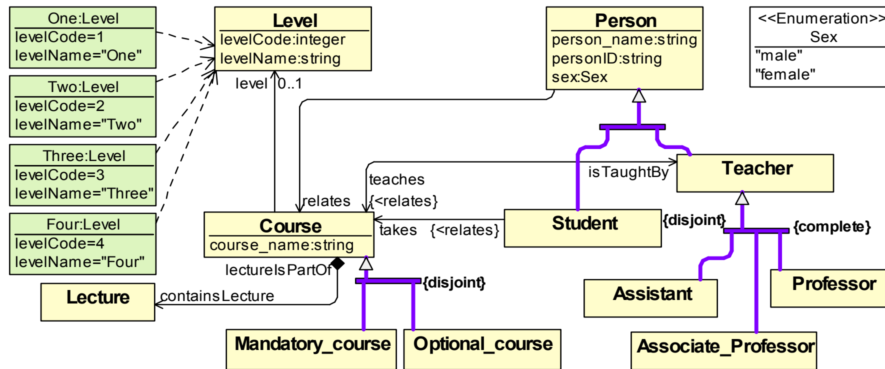
We allow use of aggregation in the OWL ontology diagram representation (e.g. *containsLecture and lectureIsPartOf* properties in Figure 3). Currently the aggregation symbol is treated as regular OWL property and is included in the UMLOWLCore metamodel and is supported in the editor for the sake of structuring and readability of the graphical model only; however, it would be prefferable to assign to the aggregation symbol the formal OWL semantics in the future.

The UML *subsettedProperty* link is used to denote a *subPropertyOf* relation (its visual symbol in the *editor* is '<', e.g. the property *takes* is a subproperty of *relates* in Figure 2). The UML cardinalities (lower and upper attributes in the *Property* class) are used to model the respective OWL cardinality restrictions.

Also OWL individuals are included in the UML class diagram specification – the OWL individuals are denoted by UML instance specifications; their concrete property values are denoted by the UML slots and their corresponding value specifications.

The UML enumerations are used to model simple OWL data ranges.

A number of semantic changes to the interpretation of the UML notation are unavoidable because OWL relies on the open world assumption whereas UML relies on the closed world assumption. To satisfy OWL needs using UML notation, we have changed default values of some UML constructions. First, in UML the default cardinality for class attributes is 1 and the default cardinality for association roles is "*". We have changed them to "*" in both cases. Second, the scope of a class attribute in UML is the corresponding class but in our notation the scope is changed to the whole ontology. Thus, if in multiple classes there are attributes with equal names, they are interpreted as the same OWL property with its domain being an intersection of classes where they are used and range being an intersection of the data types.




```
Namespace(=<http://lumii.lv/ontologies/MiniUniversity_UML.owl#>)
Namespace(rdfs=<http://www.w3.org/2000/01/rdf-schema#>)
Namespace(owl2xml=<http://www.w3.org/2006/12/owl2-xml#>)
Namespace(MiniUniversity_UML=
<http://lumii.lv/ontologies/MiniUniversity_UML.owl#>)
Namespace(owl=<http://www.w3.org/2002/07/owl#>)
Namespace(xsd=<http://www.w3.org/2001/XMLSchema#>)
Namespace(rdf=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Ontology(<http://lumii.lv/ontologies/MiniUniversity_UML.owl>
Declaration(Class(Optional_course))
SubClassOf(Optional_course Course)
DisjointClasses(Optional_course Mandatory_course)
Declaration(Class(Person))
Declaration(Class(Course))
Declaration(Class(Mandatory_course))
SubClassOf(Mandatory_course Course)
Declaration(Class(Professor))
DisjointClasses(Assistant Associate_Professor Professor)
Declaration(Class(Student))
SubClassOf(Student Person)
Declaration(Class(Assistant))
Declaration(Class(Associate_Professor))
Declaration(Class(Lecture))
Declaration(Class(Level))
Declaration(Class(Teacher))
EquivalentClasses(Teacher
ObjectUnionOf(Assistant Associate_Professor Professor))
SubClassOf(Teacher Person)
Declaration(ObjectProperty(relates))
ObjectPropertyDomain(relates Person)
ObjectPropertyRange(relates Course)
Declaration(ObjectProperty(lectureIsPartOf))
ObjectPropertyDomain(lectureIsPartOfLecture)
ObjectPropertyRange(lectureIsPartOf Course)
Declaration(ObjectProperty(containsLecture))
InverseObjectProperties(containsLecture lectureIsPartOf)
Declaration(ObjectProperty(teaches))
SubObjectPropertyOf(teaches relates)
InverseObjectProperties(teaches isTaughtBy)
ObjectPropertyDomain(teaches Teacher)
ObjectPropertyRange(teaches Course)
Declaration(ObjectProperty(level))
FunctionalObjectProperty(level)
ObjectPropertyDomain(level Course)
ObjectPropertyRange(level Level)
Declaration(ObjectProperty(takes))
SubObjectPropertyOf(takes relates)
ObjectPropertyDomain(takes Student)
ObjectPropertyRange(takes Course)
ObjectPropertyDomain(isTaughtBy Course)
ObjectPropertyRange(isTaughtBy Teacher)
Declaration(DataProperty(personID))
DataPropertyDomain(personID Person)
DataPropertyRange(personID xsd:string)
Declaration(DataProperty(course_name))
DataPropertyDomain(course_name Course)
DataPropertyRange(course_name xsd:string)
Declaration(DataProperty(levelCode))
DataPropertyDomain(levelCode Level)
DataPropertyRange(levelCode xsd:integer)
Declaration(DataProperty(levelName))
DataPropertyDomain(levelName Level)
DataPropertyRange(levelName xsd:string)
Declaration(DataProperty(sex))
DataPropertyDomain(sex Person)
DataPropertyRange(sex
DataOneOf("male" "female"))
Declaration(DataProperty(person_name))
DataPropertyDomain(person_name Person)
DataPropertyRange(person_name xsd:string)
ClassAssertion(Three Level)
DataPropertyAssertion(levelCode Three "3")
DataPropertyAssertion(levelName Three "Three")
ClassAssertion(One Level)
DataPropertyAssertion(levelName One "One")
DataPropertyAssertion(levelCode One "1")
ClassAssertion(Two Level)
DataPropertyAssertion(levelName Two "Two")
DataPropertyAssertion(levelCode Two "2")
ClassAssertion(Four Level)
DataPropertyAssertion(levelName Four "Four")
DataPropertyAssertion(levelCode Four "4"))
```

Fig. 3 A mini-university ontology (UML notation and OWL Functional Syntax)

3 Extension to UMLOWLCore metamodel

The UMLOWLCore metamodel is sufficient only for denoting a part of OWL constructs. Figure 4 shows UMLOWLCoreExtended metamodel that is a hard extension of UML metamodel and that is used as a basis for our OWL notation. The UMLOWLCoreExtended metamodel extends the UMLOWLCore metamodel (Figure 1) with constructs that enable convenient combination of graphical and textual rendering facilities for almost all OWL 2.0 constructs.

Figure 5 shows an illustration of the use of the extended UML notation for OWL on the mini-university ontology example.

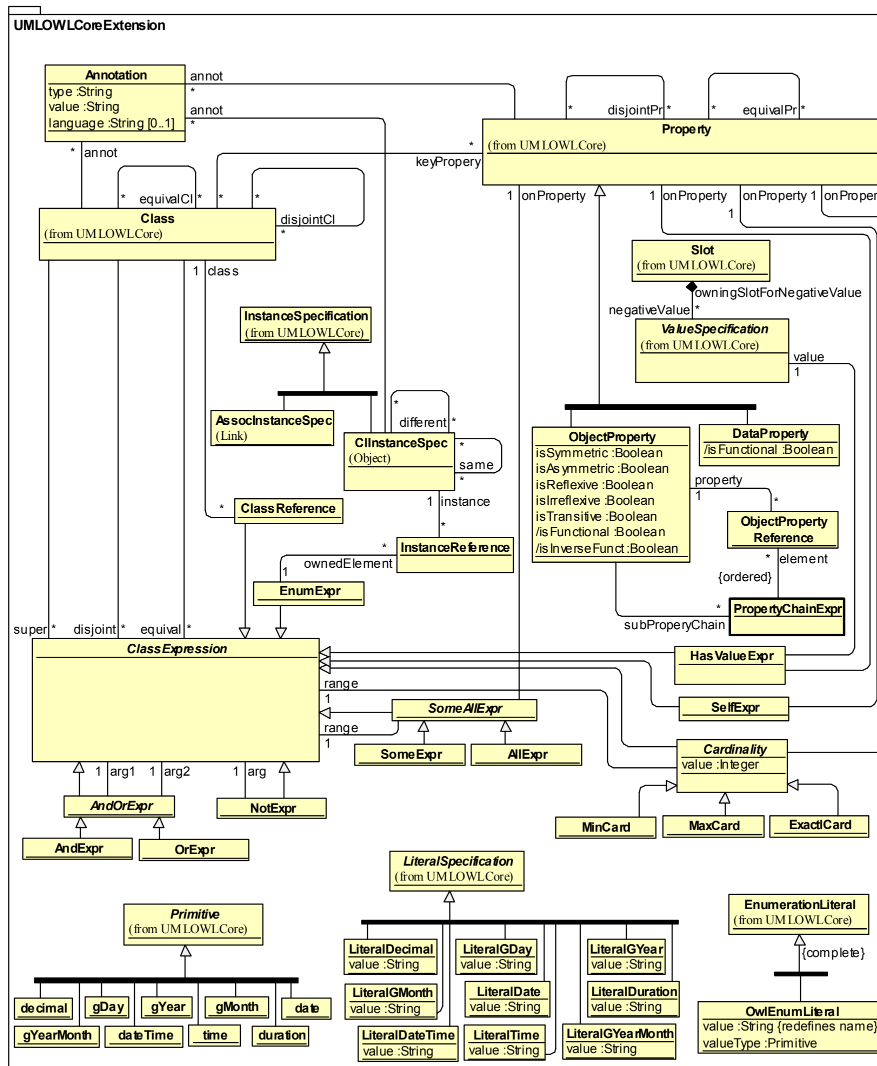


Fig. 4 UMLOWLCoreExtension metamodel

In what follows, we describe the details of our proposed UML notation for rendering and editing of OWL ontologies. We note that, as it is common already for UML class diagrams, also here in a number of cases we allow alternative graphical and/or textual notations for the same OWL construct. This allows the user to tune the look of a diagram to his/her taste, as well as to use the rendering option that is most suitable to the size and the structure of the particular ontology.

3.1. Equivalent and disjoint classes and properties

The simplest extensions to the UML metamodel are equivalent and disjoint classes and properties which are introduced in the extended UML by *eqClass* and *disjClass* relations from UML *Class* to UML *Class* and *eqProperty* and *disjProperty* relations from UML *Property* to UML *Property*.

The OWL class equivalence is modeled by the eqClass relation, and it can be visually represented in the diagram in two ways – either as a connector with <<equivalent>> stereotype linking two classes, or as a note symbol with <<equivalent>> stereotype connected to all equivalent classes. The OWL class disjointness is modeled by the disjClass relation, and it can be visually represented in the diagram either as a connector with <<disjoint>> stereotype linking two classes, or as a note symbol with <<disjoint>> stereotype connected to all disjoint classes (see Figure 5 where the disjointness of Person, Level and Course classes is asserted). We note that the class disjointness can be asserted also by means of attaching a disjoint tag to the GeneralizationSet already present in the original UMLOWLCore metamodel. There are other options available for denoting the class equivalence and disjointness using class expression notion that is explained later.

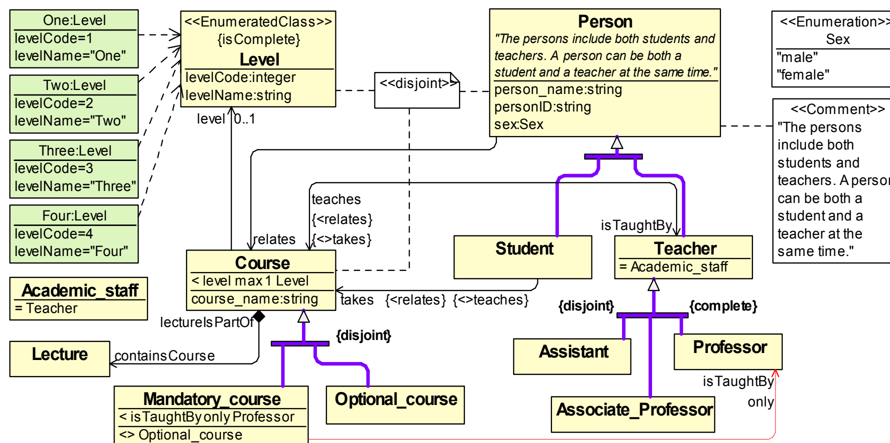


Fig. 5 A mini-university ontology (UMLOWLCoreExtended notation)

The OWL property equivalence is modeled by the *eqProperty* relation, and it is

represented by an equivalent property compartment in the property visualization; to assert that a property *p1* is equivalent to a property *p2*, we add a *{= p2}* compartment to the *p1* visualization (we may add a *{= p1}* compartment to the *p2* visualization, as well). A similar notation, using a <> symbol instead of = represents OWL property disjointness being modeled by *disjProperty* relation. For instance, in Figure 5 properties *teaches* and *takes* linking *Teacher* and *Course* classes are disjoint.

3.2. Class expressions

A principal source of OWL expressive power is its ability to form class expressions by means of Boolean expressions (and, or, not) out of the declared classes (referenced by the class name) and property-based constraints (e.g. an individual may satisfy a certain class expression when some (resp. all) links from this individual belong to a certain class (or a class expression); the cardinality restrictions also form a similar kind of class expressions). We have extended the basic UML metamodel by a *ClassExpression* notion and introduced a *ClassReference* class whose instances allow considering UML classes as class expressions (more precisely, a UML class is referenced via the *ClassReference* instance from the *ClassExpression* instance).

We generally depict the class expressions using OWL Manchester encoding [12], while we retain the traditional UML presentation for named classes. We include in the metamodel direct means of stating that a class is a subclass of, is equivalent to, or is disjoint with a class expression (the *super*, *equival* and *disjoint* relations from *Class* to *ClassExpression*), and we provide designated textual compartments in the class symbols in the diagrams where to show the class expressions that are related to the class via these relations. For instance, in Figure 5 the class 'Mandatory course' via the '<' - prefixed compartment is said to be a subclass of the expression '*isTaughtBy only Professor'*. In this case there is a *ClassReference* instance pointing to the class *Professor*, and it is *range* to an *AllExpr* object that has *isTaughtBy* as its *onProperty* value. The compartments for *equival* (an equivalent class expression) and *disjoint* (a disjoint class expression) are prefixed by '=' and '<>' respectively.

There is also an alternative form of denoting the fact that a class *c* is a subclass of a 'some values from' or 'all values from' expression (in this case we use a terminology that *c* satisfies a 'some values from' or 'all values from' constraint) namely a constraint line (depicted in the diagram in red color) leading from *c* to the class that corresponds to the *classReference* that is *rangeCl* of the expression; the constraint line is labeled by the name of the expression's *onProperty* property and the word 'only' or 'some' ('only' corresponds to an 'all values from' and 'some' to 'some values from' constraint). In Figure 5 the constraint line (denoted in red in a color diagram) from 'Mandatory course' to 'Professor' shows an alternative form of denoting the *subClassOf(Mandatory course, isTaughtBy only Professor)* constraint.

The OWL cardinality constraints *subClassOf(c, p card n cc)*, where *c* is a class, *cc* is a class expression, *card* $\in$ *{min, max, exactly}*, *n* is a nonnegative integer and *p* is a property) can be denoted either using the Manchester encoding (i.e. by adding a '< *p card n cc*' compartment to the class *c* symbol); or by the UML style cardinality

notation either at the line corresponding to *p*, if *p* domain is *c* and *p* range is class that is referenced by *cc*, or at a red constraint line that originates at *c* and leads into *cc*.

3.3. Anonymous classes

There may be a need to assert superclasses, equivalent classes or disjoint classes relation not only between two named classes, or between a class and a class expression, but also between two class expressions. It may be necessary in OWL ontology to define a domain or a range of a property not being a type, but a class expression, instead. These situations are modeled in the proposed metamodel by introducing anonymous (non-named) classes that are defined to be equivalent (via the *equival* relation) to the respective class expressions whose usage is required in a context in a diagram, where otherwise only classes (and not class expressions) were possible. Visually in the diagram such anonymous classes are depicted by symbols like any other classes, with all possibilities to be connected by lines and to have compartments, except that these classes have no given names.

We note that a similar construction to our anonymous classes is present in UML/OWL profile [6, 13], however, the essential source for our diagram compactness is our ability to introduce the anonymous classes (the restriction classes) as graphical symbols only in exceptional cases.

3.4. Enumerated classes

In the extended UML metamodel there are enumeration expressions, each owning a number of *InstanceReference* instances. The enumeration expressions correspond to OWL *ObjectOneOf* construction. In the graphical notation OWL Manchester encoding can be used to present the enumeration expressions (e.g., a class can have a compartment =*{A,B,C}*, where *A*, *B* and *C* are instance names).

The enumerated class construct that is similar to the one in [13] provides an alternative notation to the OWL Manchester encoding of the enumeration expressions. The enumerated class in our extended UML metamodel is a (named) class *c* that is equivalent (via *equival* link) to some enumeration expression *e* (for instance, *e* may be the expression *{A,B,C}*). Notationally we add a stereotype <<*EnumeratedClass*>> together with a tagged value *isComplete* to *c*. In this case the enumeration expression *e* needs not to appear explicitly in the OWL ontology presentation, however, the class *c* is assumed to be equivalent to the enumeration of all its instances being present in the diagram (and that are denoted either by instance-of links to *c*, or by explicit specification of *c* as instance's type in the instance denoting box). The corresponding expression *e* in this case is implicitly presented in the concrete diagram, and it will be restored explicitly when exporting the diagram into the OWL notation. In Figure 5 the class *Level* is defined to be an enumeration class and it is defined to be equivalent to the enumeration expression *{One, Two, Three, Four}*.

3.5. Further metamodel extensions

The UMLOWLCoreExtended metamodel provides means for introducing symmetric, asymmetric, reflexive, irreflexive and transitive characterizations for object properties in OWL diagram; in notation these characterizations are added as textual compartments to lines representing the respective property (these characterizations are available only for properties that are depicted as lines connecting their domain and range class boxes). The functional characterization for datatypes and object properties and inverse functional characterization for object properties are available in the metamodel as derived attributes to the data property and object property classes; the core specification of these characterizations in the extended UML metamodel is by means of cardinalities.

We note also the possibilities to add *same* and *different* specifications to OWL individuals that are denoted as instance specifications in the UMLOWLCore metamodel. On the graphical notation level these possibilities are available both in binary specification form by offering lines with stereotypes <<*sameAs*>> and <<*different*>>, and in n-ary specification form by offering note boxes with the same stereotypes and connecting them to the corresponding instances.

As for instance specifications, we introduce an *owningSlotForNegativeValue* link into the UMLOWLCoreExtended metamodel allowing specifying OWL negative data property assertions. Notationally, these specifications are similar to ordinary ("positive") data property assertions, with = replaced by <> (e.g. *x* <> *5* instead of *x* = *5*).

In order to model the data in OWL ontologies, we extend the spectrum of available primitive data types, as well as we classify the literal specifications by their corresponding primitive data types.

We provide in the UMLOWLCoreExtended metamodel also means for introducing annotations and attaching those to classes, properties and class instance specifications. The annotations to the classes can be depicted visually either in specifically designated textual compartments, or by respective stereotyped note symbols that are connected to the class symbols that are being annotated. The annotations typically are visually containing both the annotation property and annotation value, except for labels and comments which have special notation. It is also possible to restrict the set of annotations that are visible in the diagram.

UMLOWLCoreExtended covers also the advanced OWL 2 features such as *keyProperty, PropertyChainExpr*, and *SelfExpr* – some of them are still to be implemented in the editor.

4 The Graphical Editor

To make the notation usable in practice we have built a graphical OWL editor (OWLGrEd) which contains a number of additional services to ease ontology development and exploration, e.g. different layout algorithms for automatic ontology visualization, search facilities, zooming, graphical refactoring and interoperability

with Protégé. The editor is built using TDA [14, 15] technology. Figure 6 shows an African Wildlife ontology [16] in our editor.

Graphical refactoring is one of the most important services that allows modifying graphical notation without changing semantics as long as the same concept can be expressed through different constructs. This feature allows the user to choose the most compact graphical format depending on the context and taste. One of the typical situations illustrating the need for graphical refactoring is generalization and fork: if there is a single super class with multiple incoming generalization lines, a fork can be added to reduce multiple lines into a single line, and vice versa.

Automatic layout and search facilities are crucial when ontologies become large and their management becomes more difficult. A good automatic layout is significant for understanding large ontologies, whereas searching for the specific element in large ontologies may become irritating without an appropriate service. Therefore several alternative automatic layout modes and searching mechanism allowing finding the necessary element by the value for one of its text fields, e.g. searching class by its name is supported in our editor.

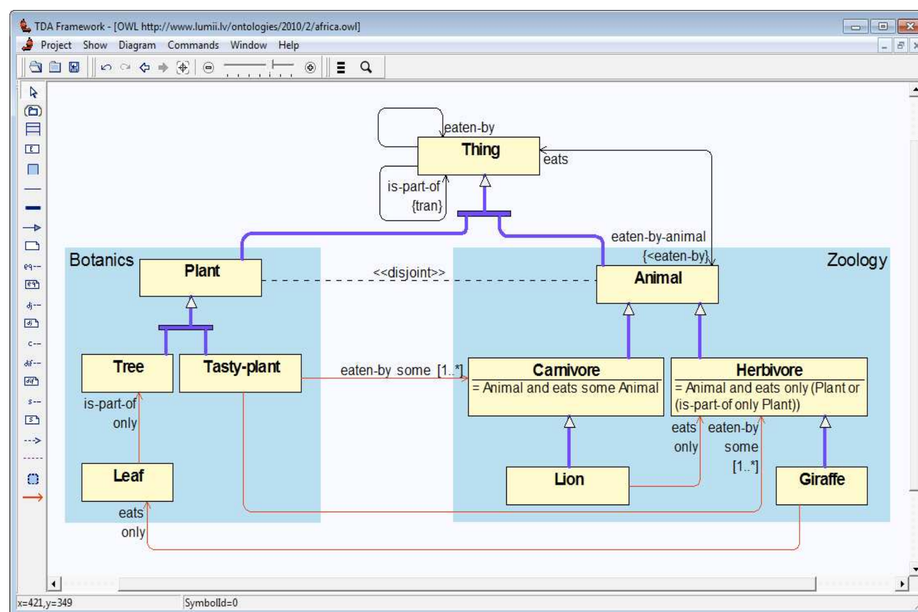


Fig. 6 An African Wildlife ontology in OWLGrEd editor

A more advanced service is full interoperability with Protégé 4 [9], an editor widely used by ontology developers. The interoperability is implemented via custom Protégé plug-in that allows to send and receive via TCP/IP socket an active ontology between our editor and Protégé. In both directions ontologies are sent in interchange format, but generally any OWL serialization is acceptable. Interoperability allows ontology developers to use Protégé without changing their habits and only afterwards

to visualize ontologies in external graphical editor using various automatic layout algorithms (a user can specify the way ontologies will be visualized by selecting notation options in preferences.) In the graphical editor ontology developers can also create new ontologies from scratch or graphically edit ontologies imported from Protégé in a WYSIWYG way; all graphically developed ontologies can afterwards be exported to Protégé from where they can be stored to various formats or checked with OWL reasoners.

5 Additional Features

Additional features described here relate both to the graphical presentation and to the graphical editor.

Ontology creation effectively consists of two stages – the logical ontology definition in Protégé-like environment and the graphical layout creation where logically closely related items are grouped together. The second stage is particularly crucial for making the ontology easy-to-read.

OWLGrEd provides an additional graphic comment – a colored background frame for the relatively autonomous sub-parts of the ontology. Although this graphic comment is somewhat fuzzy, it is very helpful for the reader of the ontology. This graphic feature is not new in software engineering [17], but we have made it applicable also to graphic ontologies, where it effectively functions as a weak ontology structuring feature.

The traditional UML structuring feature is packages and their import, while in OWL large ontologies can be split into smaller ontologies, which later can be imported into other ontologies. Although these two approaches are similar, as noted in [6], there are also some differences which shall not be ignored. The main difference is that due to globally unique URI naming of properties in OWL (as opposed to class-limited scope of properties in UML), the borders of imported OWL ontologies and UML packages may be quite different. For example, in OWL it is perfectly legal to define classes in one ontology and properties of these classes in another ontology, and afterwards to import both of these ontologies into some larger merged ontology. This effect requires to extend the concept of UML package in UMLOWLCoreExtended.

Finally, by relying on heavyweight extension of UML (as opposed to UML extension towards OWL by stereotypes [6]), we have enabled the use of UML stereotyping mechanism also for graphic OWL ontologies (e.g. using other icons than rectangles for depicting Pizza or Cheese classes).

6 Conclusion and Future Work

In this paper we have described a new, compact OWL graphical notation and a graphical editor implementing the notation. Our notation is based on UML class diagrams with additional constructs for OWL specific concepts – our aim is to cover most of OWL 2.0 constructions. Although UML and OWL may seem similar, UML

make closed world assumption whereas OWL make open world assumption that is resolved by changing the semantics of UML notation and adding new symbols. The use of expressions in Manchester encoding combined with graphical notation makes the proposed notation compact and easy understandable to those familiar with UML and Protégé.

The editor has a number of features to ease ontology exploration and development, e.g. automatic layout algorithms and options for selecting which concepts shall be displayed. We are planning to add an option to store graphic layout information inside ontologies (we consider adding it as a special kind of annotations). We would also like to improve integration with Protégé, in particular, to synchronize ontologies in both tools after every editing step - current implementation exchanges only whole ontologies. Finally, we also intend to implement the stereotyping mechanism in our editor.

References

1. OWL Web Ontology Language Reference. W3C Recommendation 10 February2004, http://www.w3.org/TR/owl-ref/
2. Pellet, http://clarkparsia.com/pellet/
3. Fact++, http://owl.man.ac.uk/factplusplus/
4. Unified Modeling Language: Infrastructure, version 2.1. OMG Specification ptc/06-04-03, http://www.omg.org/docs/ptc/06-04-03.pdf
5. Unified Modeling Language: Superstructure, version 2.1. OMG Specification ptc/06-04-02, http://www.omg.org/docs/ptc/06-04-02.pdf
6. Ontology Definition Metamodel OMG Document formal/2009-05-01, http://www.omg.org/spec/ODM/1.0
7. TopBraid Composer, http://www.topquadrant.com/products/TB_Composer.html
8. Brockmans S., Volz R., Eberhart A., and Loffler P. Visual Modeling of OWL DL Ontologies Using UML, ISWC, Hiroshima, Japan, 2004, pp. 198-213.
9. Protégé 4, http://protege.stanford.edu/
10. OWL 2 Web Ontology Language. W3C Recomendation 27 October 2009, http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
11. OWL Functional Syntax, http://www.w3.org/TR/owl2-syntax/
12. OWL 2 Manchester Syntax, http://www.w3.org/TR/owl2-manchester-syntax/
13. Kendall E., Bell R., Burkart R., Dutra M., Wallace E. Towards a Graphical Notation for OWL 2, CEUR Workshop Proceedings, Volume 529, Virginia, USA, 2009.
14. Barzdins J., Rencis E., and Kozlovics S. The Transformation-Driven Architecture, Proc. of 8th OOPSLA Workshop on Domain-Specific Modeling. Nashville, USA, 2008, pp.60-63.
15. BarzdinsJ., Cerans K., Kozlovics S., Rencis E., and Zarins, A. A Graph Diagram Engine for the Transformation-Driven Architecture, Proc. of 4th International

Workshop of Model-Driven Development of Advanced User Interfaces, Florida, USA, 2009, pp.29-32.
16. Antoniou G., van Harmelen F. A Semantic Web Primer, Second Edition, MIT Press, 2008.
17. Byelas H., Telea A. Visualization of Areas of Interest in Software Architecture Diagrams, SOFTVIS 2006, Brighton, United Kingdom, September 04-06, 2006, pp. 105-114.