OWLGrEd: a 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. There have been several attempts to visualize OWL ontologies with UML style diagrams. Unlike ODM approach of defining a UML profile for OWL, we propose an extension to UML class diagrams (hard extension) that allows a more compact OWL visualization. The compactness is achieved through the native power of UML class diagrams extended with optional Manchester encoding for class expressions thus avoiding many explicit anonymous classes typical in ODM. We have implemented the proposed compact visualization in a UML style graphical editor for OWL 2. The editor contains a rich set of graphical layout algorithms for automatic ontology visualization, search facilities, intelligent zooming, graphical refactoring and interoperability with Protégé 4.

Keywords: OWL, graphical editor, visualization.

1 Introduction

OWL is gradually becoming the most widely used knowledge representation language that has been successfully deployed in a number of applications. Due to formal semantics and availability of reasoners for OWL, it is gaining popularity also in the software engineering community so far largely dominated by UML. Many newcomers have a background in software engineering where UML diagrams are the prevalent form of data modeling and they share many characteristics with OWL ontologies. Although the two languages are similar and it would be natural to reuse the existing familiarity, the UML notation cannot be used as is, because some OWL constructs have no equivalents in UML. A number of solutions have been proposed [1, 2], the most notable is ODM [1] 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 is hard to understand and does not facilitate comprehensibility.

In our opinion the most important feature for archiving readable graphical OWL notation is its maximum compactness. We are using UML notation as far as possible, but for concepts that are unique to OWL we extend UML with additional symbols and textual expressions (chapter 2). To make the notation usable in practice we have built an editor and a number of features to ease ontology creation and exploration (chapter 3). The latest version of the editor can be downloaded from http://OWLGrEd.lumii.lv.

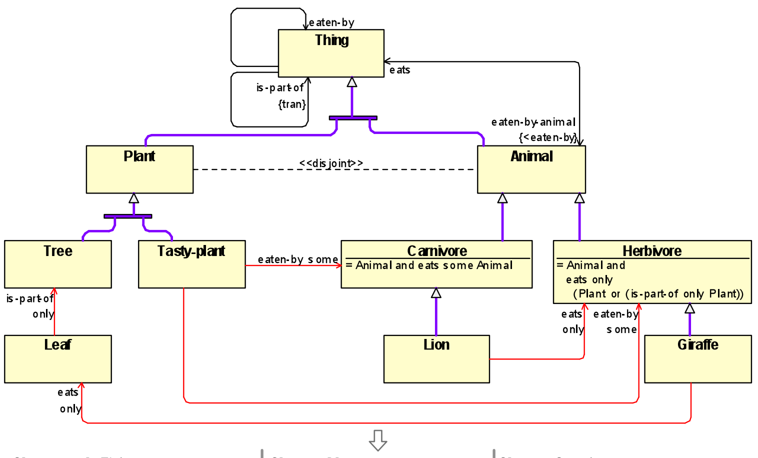
2 Extended UML Notation for OWL

The proposed graphical notation is based on UML class diagrams. For most features there is one to one mapping from OWL to UML concepts, e.g. ontologies to packages, OWL classes to UML classes, data properties to class attributes, object properties to associations, individuals to objects, etc. Meanwhile for OWL concepts not having a good UML equivalent, the following new extension notations were added:

- a field in classes for *equivalent class*, *superclass* and *disjoint class* expressions written in Manchester OWL syntax [3];
- a field in associations and attributes for specifying *equivalent*, *disjoint* and *super* properties as well as a field for specifying property characteristics, e.g., *functional*, *transitive*, etc.;
- anonymous classes containing *equivalent class expression* but no name;
- connectors for visualizing *disjoint*, *equivalent*, etc. axioms;
- boxes with connectors for n-ary *disjoint*, *equivalent*, etc. axioms;
- connectors for visualizing data property restrictions *some*, *only*, *exactly*, etc.

The main advantage of these extensions is the option to specify class expressions in compact textual form rather than using separate graphical element for each logical class, constructor (*and*, *or*, *not*) and restriction, in cases where the expression is referenced only once. If the expression is referenced in multiple places, it can optionally be shown as an anonymous class.

To better understand the proposed notation lets consider an example in Figure 1 that depicts the popular African wildlife ontology using our extended UML notation. At first glance it is very similar to an ordinary class diagram, therefore it should be immediately understandable by most people with software engineering background. On a closer inspection we can see that it also includes a number of extensions, namely classes *Carnivore* and *Herbivore* have a field that contains the equivalent class expression in Manchester syntax, there are a number of red connectors that correspond to property restrictions and there is a line between *Plant* and *Animal* for showing that they are disjoint. We believe that the example is intuitive and the meaning of the newly introduced symbols can be easily guessed, e.g. *Tasty-plant* must be eaten by some *Carnivore* and some *Herbivore*. Super properties are depicted as a text next to subproperty's name, e.g. *{<eaten-by}* next to subproperty name *eaten-by-animal* (symbol '<' corresponds to '*subproperty*' in UML notation).



```
Class: owl:Thing
ObjectProperty: is-part-of
  Characteristics:
    Transitive
ObjectProperty: eaten-by
ObjectProperty: eats
  Domain:
    Animal
  InverseOf:
    eaten-by-animal
ObjectProperty: eaten-by-animal
  Range:
    Animal
  InverseOf:
    eats
  SubPropertyOf:
    eaten-by
Class: Tree
  SubClassOf:
    Plant
```

```
Class: Plant
  SubClassOf:
    owl:Thing
  DisjointWith:
    Animal
Class: Tasty-plant
  SubClassOf:
    Plant,
    eaten-by some Carnivore,
    eaten-by some Herbivore
Class: Leaf
  SubClassOf:
    is-part-of only Tree
Class: Animal
  SubClassOf:
    owl:Thing
  DisjointWith:
    Plant
```

```
Class: Carnivore
  EquivalentTo:
    Animal
    and (eats some Animal)
Class: Lion
  SubClassOf:
    Carnivore,
    eats only Herbivore
Class: Giraffe
  SubClassOf:
    Herbivore,
    eats only Leaf
Class: Herbivore
  EquivalentTo:
    Animal and
    (eats only
      (Plant or
       (is-part-of only Plant)))
```

Fig. 1. Extended graphical notation and corresponding Manchester notation

3 Services of the Editor

The editor is implemented using transformation driven architecture (TDA) [4, 5] and it has a number of special services to ease ontology development. One of the services is graphical refactoring that allows modifying graphical notation without changing semantics as long as the same concept can be expressed through different constructions. This feature allows the user to choose the most compact graphical format depending on the context and the taste. A typical situation 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.

When ontologies become large, their management becomes more difficult and besides support for import of modularized ontologies additional features are required from the editor. First, a good automatic layout is crucial for understanding large ontologies therefore several alternative layout modes are supported. Second, searching for the specific element in large ontologies may become painful and irritating without an appropriate service. A search mechanism implemented in our editor allows finding the necessary element by specifying the value for one of its text fields. For example, it allows finding classes by their name or the value of any other text field.

A more advanced service is full interoperability with Protégé 4 [6], an editor widely used by ontology developers. The interoperability is implemented via custom Protégé plug-in that allows to send via TCP/IP socket an active ontology from our editor to Protégé, and vice versa. Ontologies in both directions are sent in interchange format, but generally any OWL 2.0 serialization is acceptable. Interoperability allows ontology developers to use Protégé without changing their habits and afterwards visualize ontologies in our graphical editor using various automatic layout algorithms along with manual layout tuning. Moreover, a user can specify the way ontologies will be visualized by selecting notation options in preferences. In our graphical editor ontology developers can create new ontologies from scratch or alternatively edit graphically ontologies imported from Protégé; all graphically developed ontologies can afterwards be exported to Protégé from where they can be stored to various formats or checked with OWL reasoners.

4 Conclusion and Future Work

In this paper we described a new, compact OWL graphical notation and a beta-version implementation of the actual graphical editor. Our notation is based on UML class diagrams with additional constructs for OWL specific concepts – our aim is to cover full OWL 2.0 specification. 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 allows exchanging only whole ontologies.

References

1. ODM UML profile for OWL, http://www.omg.org/spec/ODM/1.0/PDF/
2. TopBraid Composer, http://www.topquadrant.com/products/TB_Composer.html
3. OWL 2 Manchester Syntax, http://www.w3.org/TR/owl2-manchester-syntax/
4. Barzdins, J., Rencis, E., Kozlovics, S. The Transformation-Driven Architecture, Proc. of 8th OOPSLA Workshop on Domain-Specific Modeling. Nashville, USA, 2008, pp.60-63.
5. Barzdins, J., Cerans, K., Kozlovics, S., Rencis, E., Zarins, A. A Graph Diagram Engine for the Transformation-Driven Architecture, Proc. of 4th MDDAUI. Florida, USA, 2009, pp.29-32.
6. Protégé, http://protege.stanford.edu/