# OWLGrEd: a UML Style Graphical Editor for OWL

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 many attempts to visualize OWL ontologies but none of them is considered completely satisfactory and this is still an open problem. We propose a UML style graphical editor for OWL which not only visualizes ontologies using extended UML class diagram notation but also provides ontology editing facilities unavailable in most of the other tools. Moreover, the editor contains additional features for graphical ontology exploration and development including interoperability with Protégé 4.

**Keywords:** OWL, graphical editor, visualization.

## 1 Introduction

Thousands of ontologies were developed in the past years and surely more will be developed in the future, therefore availability of efficient ontology development tools including graphical editors is essential. Ontologies are usually described in Web Ontology Language (OWL) which originally was defined as an extension to RDF graphs, therefore one of OWL canonical forms is a set of subject-predicate-object triples. This format is very uniform, which makes it easy to parse and store in computers, but it is completely unusable for humans. Humans tend to think in terms of higher abstraction levels like classes, instances and relations, but the current ontology visualization tools like IsaViz [1], OWLViz [2], GrOWL [3], Welkin [4], etc. visualize ontologies by showing every RDF triple as two nodes with a labeled arrow between them. Thus the information gets cluttered and spread over a large area, making the structure hard to perceive.

For a graphical form to be useful it has to group related concepts together – the approach that has been successfully used, e.g. in UML class diagrams. Many concepts of OWL are very similar to those of UML class diagrams and therefore there have been attempts to define a UML profile for OWL [5] that would make it possible to use existing UML tools to create and visualize ontologies. However, OWL has more features than UML class diagrams like class expressions, anonymous classes, etc., which are commonly used, but have unintuitive graphical representation. Therefore, even though a UML profile is better than RDF graphs, it is still hardly comprehensible. Another option is to use Protégé OWL editor [6] that enables to load

and save ontologies, edit its classes and properties, and define a class taxonomy. It provides a detailed view for each concept in an ontology, but does not show well its overall conceptual structure.

Our proposal is to extend the UML class diagram notation with Manchester-like syntax for the missing OWL features (chapter 2), thus making the notation compact and comprehensible. We have developed an editor for this notation that has a number of features to ease ontology creation and exploration, e.g. different layout algorithms for automatic ontology visualization, search facilities, intelligent zooming, graphical refactoring and interoperability with Protégé (chapter 3). In fact, the application of UML class diagram notation to OWL is not completely new and has been implemented in TopBraid Composer [7]. However, it is based on simplified UML class diagram model, lacks graphical editing facilities, and the available graphical services are limited as well.

## 2 Compact OWL Graphical Notation

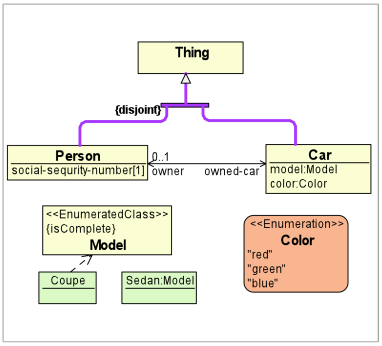
The proposed graphical notation is based on UML class diagrams. For most features there is one to one mapping from OWL to UML concepts, e.g. ontologies to packages, OWL classes to UML classes, data properties to class attributes, object properties to associations, individuals to objects, etc. Meanwhile there are added also a number of new graphical elements that are not part of UML notation. Classes have fields where OWL expressions can be inserted, e.g. equivalent class expressions, superclass expressions and disjoint class expressions. Similar fields are added to associations and attributes. Anonymous classes are shown as boxes with only equivalent class expression definition possibility. There are a number of ways to visualize anonymous superclasses:

1) as a textual expression inside a subclass box,
2) as a generalization line from the subclass to the corresponding anonymous class,
3) as multiplicity constraints in association, or
4) as a restriction line towards the corresponding class (e.g. the red line *eats* between *Lion* and *Herbivore* in fig. 2).

Ontology may be split into multiple diagrams inside a package with each diagram showing a different view of the ontology or its subset. Multiple generalization lines can be merged with a fork symbol, to reduce the number of incoming lines in a superclass. The editor provides means to switch from one graphical form to another (chapter 3), e.g. merge generalization lines within a fork.

To better explain the proposed notation let us consider an example in Figure 1. It shows a simple ontology representing people, cars, their properties and relations. This visualization uses only standard features of UML class diagrams. Classes are represented by rectangular boxes and data properties are shown as labels inside the class box. Object properties are represented with lines between boxes corresponding to their domain and range classes. If object property has an inverse then both are represented with the same line, e.g. *owner* and *owned-car*. Object properties can alternatively be shown as labels inside domain class box, e.g. object property *model*. Cardinality restrictions of such inline-shown properties are depicted in square

brackets next to the corresponding property name, in the same way as in regular class diagrams. If a range of a data property is an enumeration of values, then the enumeration is depicted as a box with rounded corners, e.g. *Color*. The fact that a class is defined by its instances is shown with a label <<*EnumeratedClass*>>, e.g. class *Model*.

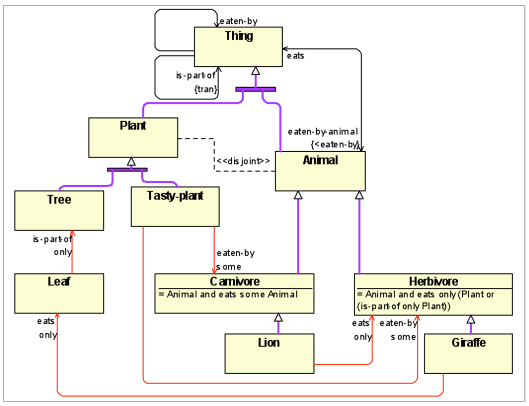



```
Class: owl:Thing
Class: Person
    SubClassOf:
        owl:Thing,
        social-sequrity-number exactly 1
    DisjointWith:
        Car
Class: Car
    SubClassOf:
        owl:Thing
    DisjointWith:
        Person
Class: Model
    EquivalentTo:
        {Coupe, Sedan}
Individual: Sedan
    Types:
        Model
Individual: Coupe
    Types:
        Model
DataProperty: color
    Domain:
        Car
    Range:
        {"blue" , "orange" , "red"}
```

```
ObjectProperty: property
    Domain:
        Person
    Range:
        Car
    InverseOf:
        owner
ObjectProperty: owner
    Characteristics:
        Functional
    Domain:
        Car
    Range:
        Person
    InverseOf:
        property
ObjectProperty: model
    Domain:
        Car
    Range:
        Model
DataProperty: social-sequrity-number
    Domain:
        Person
```

**Fig. 1.** Proposed graphical notation and corresponding Manchester notation

Next example already uses some graphical notations that are not a part of UML. Figure 2 depicts the popular African wildlife ontology. The red lines are 'some values from' and 'only values from' restrictions encoded as subclasses in OWL. It is obvious that this notation eases the comprehension of ontology, 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
Class: Leaf
   SubClassOf:
      is-part-of only Tree
Class: Plant
   SubClassOf:
      owl:Thing
   DisjointWith:
      Animal
Class: Tree
   SubClassOf:
      Plant
Class: Tasty-plant
   SubClassOf:
      Plant,
      eaten-by some Carnivore,
      eaten-by some Herbivore
Class: Animal
   SubClassOf:
      owl:Thing
   DisjointWith:
      Plant
Class: Herbivore
   EquivalentTo:
      Animal and
      (eats only
        (Plant or
          (is-part-of only Plant)))
```

```
Class: Giraffe
   SubClassOf:
      Herbivore,
      eats only Leaf
Class: Carnivore
   EquivalentTo:
      Animal
      and (eats some Animal)
Class: Lion
   SubClassOf:
      Carnivore,
      eats only Herbivore
ObjectProperty: eats
   Domain:
      Animal
   InverseOf:
      eaten-by-animal
ObjectProperty: is-part-of
   Characteristics:
      Transitive
ObjectProperty: eaten-by-animal
   Range:
      Animal
   InverseOf:
      eats
   SubPropertyOf:
      eaten-by
ObjectProperty: eaten-by
```

**Fig. 2.** Extended graphical notation and corresponding Manchester notation

## 3 Services of the Editor

A number of special services are implemented in our editor 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. 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.

When ontologies become large, their management becomes more difficult and additional features are required from the editor. First, a good automatic layout is crucial for understanding large ontologies and 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, 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 serialization is acceptable. Interoperability allows ontology developers to use Protégé without changing their habits and afterwards visualize ontologies in external graphical editor using different automatic layout algorithms as well as further 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 graphically edit 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 Implementation

The editor is implemented using transformation driven architecture (TDA) [8, 9, 10] technology. TDA stores its information in the form of MDA-style models that are connected by model transformations. The user interface in TDA is implemented by means of universal engines (e.g. a graph diagramming engine, a property editor engine, etc.). Each individual tool (e.g. OWLGrEd) is created through a specially designed TDA tool definition configurator that creates instances of Tool Definition Model storing all meta information about an individual tool – element types, element styles, constraints and relationships among elements.

The Tool Definition Model instances are then interpreted by a universal interpreter that in cooperation with other TDA engines processes all end-user's actions.

Furthermore, for OWLGrEd, as for other tools, specific transformations can be created to support domain specific needs. In our case, only transformations supporting interoperability with Protégé, and specific attribute and annotation parsers had to be created. Thanks to the use of TDA it took only six person-months to produce the beta-version of the OWLGrEd editor, including development of Protégé plug-in and the design of the actual graphical notation (this has been the most time consuming part). The latest editor version can be downloaded from http://OWLGrEd.lumii.lv.

## 5 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. IsaViz, http://www.w3.org/2001/11/IsaViz/
2. OWLViz, http://www.co-ode.org/downloads/owlviz/
3. GrOWL, http://www.uvm.edu/~skrivov/growl/
4. Welkin, http://simile.mit.edu/welkin/
5. UML profile, http://www.omg.org/spec/ODM/1.0/PDF/
6. Protege, http://protege.stanford.edu/
7. TopBraid Composer, http://www.topquadrant.com/products/TB_Composer.html
8. 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.
9. Barzdins, J., Cerans, K., Kozlovics, S., Rencis, E., Zarins, A. A Graph Diagram Engine for the Transformation-Driven Architecture, Proc. of 4th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI-2009). Florida, USA, 2009, pp. 29-32.
10. Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A.., GrTP: Transformation Based Graphical Tool Building, Proc. of 3th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI-2007). Nashville, USA, CEUR Workshop Proceedings, http://ceur-ws.org, vol. 297.