Baltic J. Modern Computing, Vol. 8 (2020), No. 3, 444-460
https://doi.org/10.22364/bjmc.2020.8.3.04

Ontology Export Patterns in OWLGrEd Editor

Jūlija OVČIŅŅIKOVA

Institute of Mathematics and Computer Science, University of Latvia
Raina blvd. 29, Riga, LV-1459, Latvia

julija.ovcinnikova@lumii.lv

Abstract. The OWLGrEd ontology editor allows graphical visualization and authoring of OWL 2.0 ontologies using a compact yet intuitive presentation that combines UML class diagram notation with textual Manchester syntax for expressions. For the full use of the graphical ontology tool, it is important to be able to export the ontology in one of OWL textual standards. We describe the OWLGrEd ontology export implementation using patterns that are ascribed to editor diagram abstract syntax elements. The OWLGrEd export patterns show the relationship between editor visual constructions and ontology textual form elements. The pattern-based method allows a tool end-user to define a custom field semantics in OWLGrEd extensions, and it can be generalized also for the model-to-text transformation within other similarly structured tools.

Keywords: OWL, OWLGrEd, ontology export patterns

1. Introduction

OWL 2 (Motik et al., 2012) is a major logic-based open-world knowledge representation language for the Semantic web. The presentation of OWL ontology in a comprehensible form is essential for both the ontology developers and ontology users. A number of approaches and tools have been developed to achieve better ontology comprehensibility by presenting the ontology graphically, including OWLViz (WEB, i), VOWL (Lohmann et al., 2016), OntoDia (Mouromtsev et al., 2015), ODM (WEB, h), TopBraid Composer (WEB, l), RDF visual graph editor (Chis-Ratiu and Buchmann, 2018) and OWLGrEd (Barzdins et al., 2010c). The benefit of the graphical presentation is that the concepts that are related in the ontology are also visualized together. The article (Dudás et al., 2018) has carried out a broad study on methods and tools for graphical ontology representation.

OWLGrEd is an editor where one can edit OWL 2 ontologies in a visual environment. OWLGrEd combines UML class diagram notation and textual OWL Manchester syntax (Horridge and Peter 2012) for expressions that may occur in ontology definitions. This type of representation ensures that visually related objects are displayed together. So, object properties are connected to the property domain and range classes and data properties are represented as their domain class attributes.

For graphical ontology representation tool such as OWLGrEd, it is important to be able to use the ontology defined therein in other tools as e.g. Protégé (WEB, j), allowing, for instance, the ontology inference by the means of the available reasoners as HermiT

(Glimm et al., 2014), Pellet (Sirin et al., 2007) or FaCT++ (Tsarkov and Horrocks, 2006). A variety of other systems, including e.g. a knowledge-based framework OBIS (Zviedris et. al., 2013; Cerans and Romane, 2015) or visual query tool ViziQuer/web (Cerans et al., 2018) use ontology definitions as their input data. For an ontology created in a visual editor to be used elsewhere, an exporter that converts the graphical syntax to some OWL textual syntax is needed.

The goals of this article are: (i) to demonstrate a model-to-text transformation method allowing for modular and extensible OWLGrEd ontology diagram export into textual form; (ii) to show a grammar-based textual mapping language capable of defining correspondence between the graphical presentation of ontology constructs within OWLGrEd editor and their representation in the textual OWL Functional syntax (Motik et al., 2012) form.

The OWLGrEd ontology export is implemented using patterns that are ascribed to graphical abstract syntax elements.

The OWLGrEd tool graphical abstract syntax that is based on the nodes, edges and fields, and OWL ontology axioms in OWL Functional syntax have very different structures. A pattern-based language is one of the methods how to connect this structures. The patterns allow in a natural way to get together the model elements with a static text.

A pattern-based exporter allows to achieve a modular definition of ontology diagram semantics, by linking each ontology axiom to one basic construction in the graphical diagram to which this axiom is directly ascribed (the axiom may use also information from other locations in the graph, this information is gathered by means of path expressions relative to the axiom ascription point).

The pattern-based approach described in this article is also used in OWLGrEd extension definition (Cerans et al., 2013; Cerans et al., 2019), where the person that configures the extension (the OWLGrEd tool developers, or any other person) can write an extension field semantics definition in the language described here.

The approach developed here can be also used to translate to the textual representation models created in another graphical syntax within the GrTP/TDA modelling tool building platform (Barzdins et al., 2007) or a conceptually similar platform as ajoo (Sprogis, 2016). For instance, a visual notation for SHACL (Knublauch and Kontokostas, 2017) language for validating RDF graphs could be implemented either in GrTP/TDA or ajoo platform and the pattern-based export approach also could be applied there.

The pattern language has a simple interpreter that handles each of the constructions individually. The chosen pattern-based architecture of the exporter is expected to allow it to be transferred from its current implementation in Lua programing language (WEB, k) with lQuery library (Liepins, 2012) for data model support to another programming environment (such as e.g. JavaScript (WEB, d) with jQuery (WEB, f)).

The method of pattern-driven export described in this article can be alternatively implemented in frameworks such as Spoofax (Kats and Visser 2010) and Xtext (Voelter, 2006) that are intended for domain specific language development. This paper demonstrates the possibility and the involved structures for using the grammar-based mapping definition principles in practice without invoking a general-purpose framework and staying with the language means that are integrated within OWLGrEd technological environment.

The OWLGrEd editor export implementation is a typical Model-to-text solution. There are existing Model-to-text languages and tools such as Acceleo Query Language (AQL) (WEB, a), Epsilon Generation Language (WEB, b), Xpand (WEB, m), JET (WEB, e), MOFScript (WEB, g). Our Model-to-text transformation works directly within the environment OWLGrEd editor is implemented in. It can also serve as illustration for the constructs needed for the transformation in a practical example.

The OWLGrEd export pattern notation has been announced in (Ovcinnikova and Cerans, 2016); it has not been explained in detail until this paper.

In the rest of the paper Section 2 reviews the OWLGrEd ontology editor together with its concrete and abstract syntax; Section 3 describes OWLGrEd editor export process and pattern language; Section 4 describes OWLGrEd extensions and their export, then Section 5 concludes the paper.

2. OWLGrEd Editor Syntax

OWLGrEd[1] provides a complete graphical notation for OWL 2, based on UML class diagrams. It visualizes OWL classes as UML classes, data properties as class attributes, object properties as associations, individuals as objects, cardinality restrictions on association domain class as UML cardinalities, etc. We enrich the UML class diagrams with the new extension notations, e.g. (cf. (Barzdins et al., 2010a; Barzdins et al., 2010c)) to provide visual notations for OWL constructs that do not have corresponding UML counterparts:

- fields in classes for equivalent class, superclass and disjoint class expressions written in Manchester OWL syntax (Horridge and Peter 2012);
- fields in associations and attributes for equivalent, disjoint and super properties and fields for property characteristics, e.g., functional, transitive, etc.;
- connectors (as lines) for visualizing binary disjoint, equivalent, etc. axioms;
- boxes with connectors for n-ary disjoint, equivalent, etc. axioms;
- connectors (lines) for visualizing object property restrictions some, only, exactly, as well as cardinality restrictions.

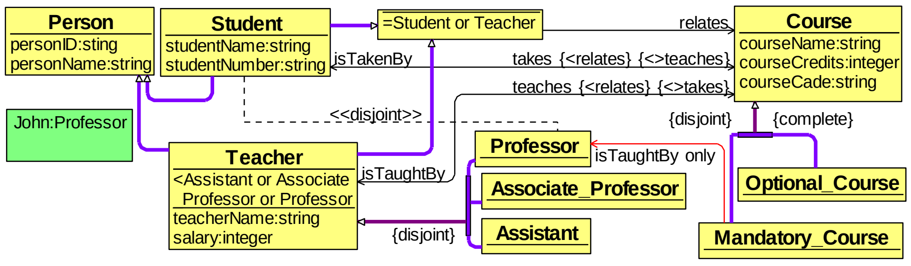


Fig. 1. A simple mini-University ontology in OWLGrEd

Figure 1 illustrates some basic OWLGrEd constructs of simple mini-University ontology. The notation is explained in more detail in (Barzdins et al., 2010a). The same ontology in OWL Functional syntax is can be found in Appendix 1.

[1] http://owlgred.lumii.lv/

OWLGrEd provides option to specify class expressions in compact textual form rather than using separate graphical element for each logical item within a class expression. Expression can optionally be shown as an anonymous class (e.g. *Student or Teacher* in Figure 1). An anonymous class is also used as a base for property domain/range specification, if this domain/range is not a named class.

The OWLGrEd tool allows both for ontology authoring (with option to save the ontology in a standard textual format) and ontology visualization that includes automated ontology diagram formation and layouting step, followed by optional manual diagram fine tuning to obtain the highest quality rendering of the ontology.

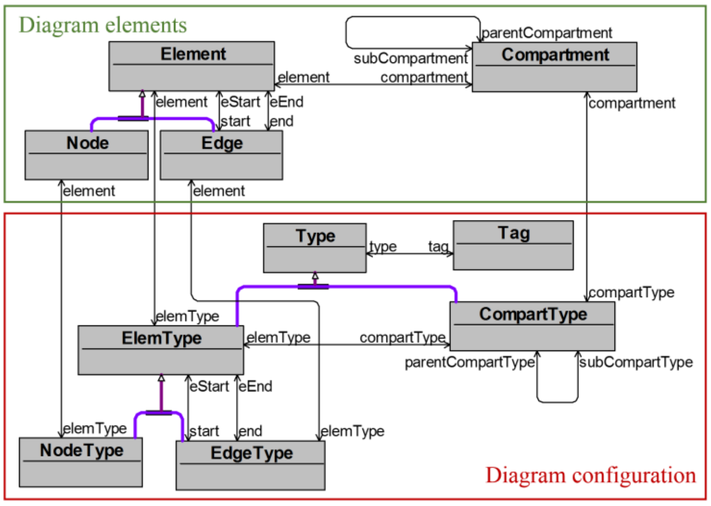


Fig. 2. OWLGrEd abstract syntax metamodel (fragment)

OWLGrEd editor is implemented in the GrTP platform (Barzdins et al., 2007). The platform hosts both a visual diagramming engine (Barzdins et al., 2009) and a model repository with a model transformation environment for holding both the editor (e.g. OWLGrEd) configuration and the ontology diagram abstract syntax structure. An essential repository structure fragment for the OWLGrEd editor is shown in Figure 2. OWLGrEd diagram visual elements correspond to the classes *Node* and *Edge*, together with *Compartment* class that corresponds to the text fields placed in nodes and attached to edges. On the configuration side, *NodeType* and *EdgeType* classes correspond to the types of nodes and edges that are allowed in diagrams of the respective type. Every element type has an ordered collection of *CompartType* class instances attached to it. These instances correspond to the list of compartment types of the diagram elements of this type. Each compartment type may consist of one or several sub-compartment types. Element and compartment types have *Tag* class linked to them. OWLGrEd export patterns for the specific element or compartment types are stored in the *Tag* class instances. The full tool platform metamodel is best explained in detail in (Barzdins et al., 2010b).

Figure 3 illustrates OWL class *Person* definition in OWLGrEd visual syntax, and the corresponding abstract syntax structure. The Person class box is represented in the OWLGrEd abstract structure as *Node* instance with *NodeType* equals to “Class”. The *Node* instance has one compartment, with compartment type “Name”, containing class name. Compartment type "Name" has *Tag* class instance, attached to it. This *Tag* instance contains export pattern for class declaration (export patterns are explained in Section 3). Name compartment has two sub-compartments with types "Name" and "Namespace", as well.

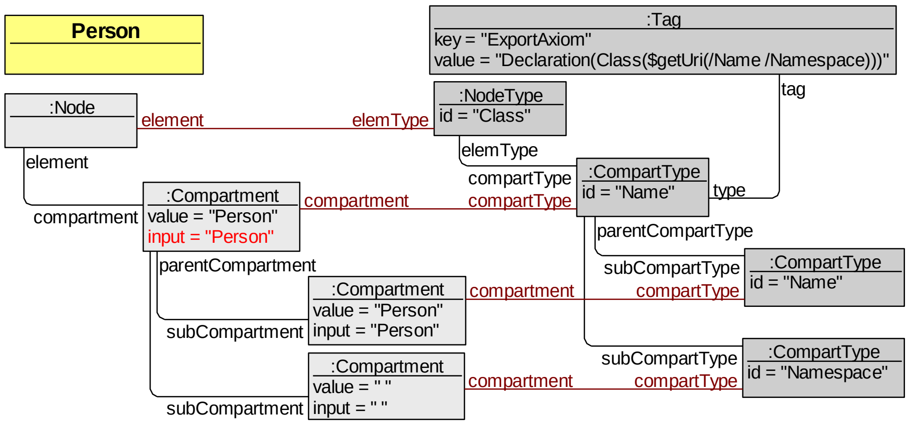


Fig. 3. OWLGrEd concrete and abstract syntax structures for a named class element (Person)

Another, wider OWLGrEd syntax example can be found in the Appendix 2.

Full compartment type structure for the class element is in the Appendix 3.

There are similarly organized structures for other visual OWLGrEd diagram elements, as well.

3. Ontology export process and patterns

Graphical ontology diagram translation into the textual format, includes rendering of the graphical ontology diagram, from Figure 1, into OWLGrEd abstract syntax (performed while editing the diagram) corresponding to Figure 2 and explicit abstract syntax translation into the text. In this section the way, how the abstract syntax translation into the text is realized in the OWLGrEd tool, is described.

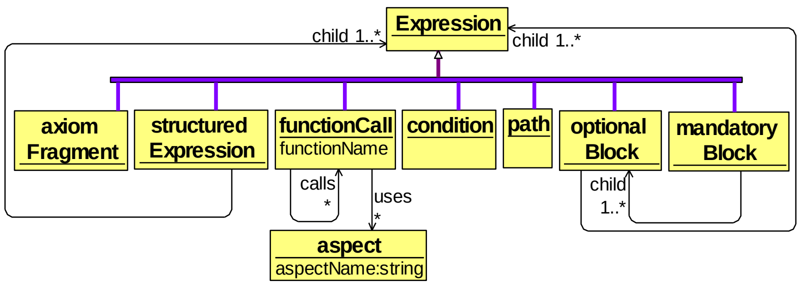


Fig. 4. Export patterns top-level structure

To ensure the OWLGrEd ontology export into OWL 2 syntax, an export pattern language and its parser was created. The parser translates the export patterns into OWL 2 Functional syntax that can be later transformed into any popular OWL 2 syntax using OWL API (Horridge and Bechhofer, 2011). As a result of the mini-University ontology diagram, shown in Figure 1, export OWL Functional syntax representation of the Appendix 1 is obtained.

The patterns are ascribed as *Tag* class instances to the type elements (cf. Figure 2, 3) and are evaluated in the context of the respective data element.

Each element or compartment type in OWLGrEd editor that represents an OWL 2 axiom has information about defined export patterns stored in *Tag* instance attached to it. OWLGrEd exporter walks through all diagram elements and compartments, that corresponding type has the export tag connected to it, parses the export pattern, and, using the parsing result and ontology diagram data, generates the OWL Functional syntax axiom.

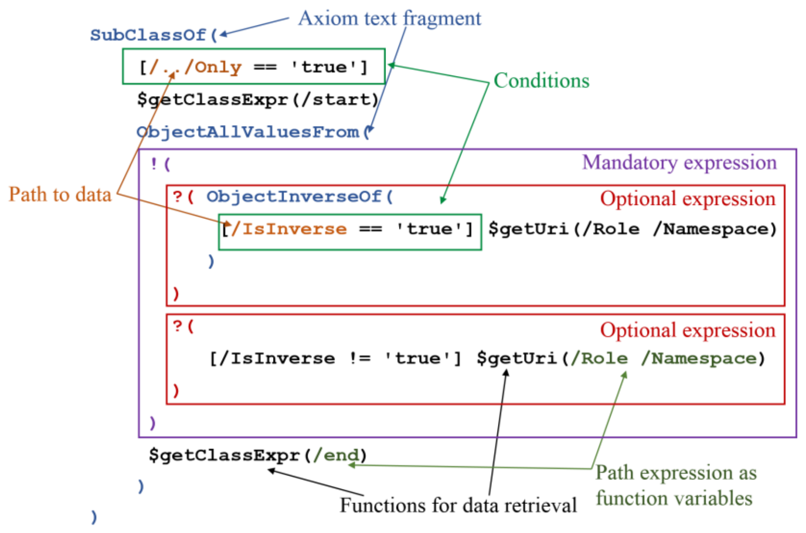
Figure 4 illustrates OWLGrEd export pattern top-level structure.

OWL export pattern consists of the following structural constructions:

- **Text fragment construction** – the part of OWL Functional syntax axiom text, such as axiom name. In the export pattern language these expressions are written in plain text.
- **Structured Expression construction** – starts with Text fragment (axiom name), followed by the list of expressions. Can be used e.g. as top-level expression constructions for the OWL Functional syntax axiom.
- **Path expression construction** – defines the path from the current location in the type structure to the required data. Can be used either directly (the result will be the compartment or the compartment set located in the given path) or as path before function or condition construction (function/condition construction will calculate the result from the location the path is pointing to). Each path element construction consists of "/" symbol and the sub-compartment type name. The two dots represent navigation to the parent level in the compartment type structure.
- **Function construction** – retrieves data from repository, calculates and returns axiom fragment (or textual value that can be used otherwise). Function construction starts with "$" symbol, followed by the function name and optional function arguments (path expressions) in brackets.
- **Condition construction** – checks, whether the axiom or axiom fragment can be applied based on the given data. Condition construction is included in square brackets. Condition may consist of several OR condition parts separated by "||" where each OR condition part consists of a first argument, an operator and a second argument. Each argument can be a path expression, a function or a constant value such as a number or a string. Several conditions can be followed by each other. In this case, all conditions must be true for the axiom or axiom fragment to be generated.
- **Optional construction** – defines an axiom fragment that is optional in a given axiom. Optional construction starts with "?" symbol, followed by optional axiom definition part in brackets. Optional construction block consists of one or several expressions. Expression may be any export pattern language top-level structure (cf. Figure 4.).
- **Mandatory construction** – defines an axiom fragment that is mandatory in the axiom. Mandatory expression starts with "!" symbol, followed by several

optional constructions in brackets, where exactly one of the optional constructions is required.

The OWLGrEd export pattern example with highlighted pattern constructions, is shown in Figure 5. Each element or compartment type may have more than one export pattern. If at least one condition or the mandatory construction in a pattern is not fulfilled, the pattern will not generate the OWL axiom.



```
SubClassOf(
  [/../Only == 'true']
  $getClassExpr(/start)
  ObjectAllValuesFrom(
   !(
     ?( ObjectInverseOf(
          [/IsInverse == 'true'] $getUri(/Role /Namespace)
        )
     )
     ?(
        [/IsInverse != 'true'] $getUri(/Role /Namespace)
     )
   )
   $getClassExpr(/end)
  )
)
```

Fig. 5. Export pattern structure example

Important part of OWLGrEd export pattern language are functions, used for retrieving and transforming the OWLGrEd diagram data into OWL axiom fragment.

The classification and description of the functions used in the ontology export pattern definition, is given in the following list:

Use context information of the entire diagram (e.g. full namespace URIs, diagram class name list, etc.)**:**

- ***getUri(compartment Name, compartment Namespace)***. Returns ontology element URI for the given *Name* and optional *Namespace* compartments. If *Namespace* compartment is not specified, the ontology local namespace is used. If both compartments are not specified, the current context compartment is used as *Name* argument.
  - Input Examples: *$getUri(/Name /Namespace), $getUri(/Expression), $getUri*
  - Output examples: *:Student, :studentName, foaf:Person*
- ***getRoleExpr.*** Returns association role URI or *ObjectInverseOf* axiom with inverse role expression. The function is always called from the association compartment context. From the current compartment it finds the association *Role* compartment by going up in the compartment tree structure. Then from the *Role* compartment the function goes down to the *Name* compartment and generates the association role URI. If there is no name present, the function finds in the compartment structure the inverse role of the current role, then its *Name*

compartment and generates the *ObjectInverseOf* axiom with inverse role expression.

- Input Examples: *$getRoleExpr*
- Output examples: *:takes, ObjectInverseOf(:isTakenBy)*

- ***getAnnotationProperty(compartment AnnotationType, compartment Namespace)***. Returns *AnnotationProperty* URI. Function, from the current annotation compartment context, finds *AnnotationType* and *Namespace* compartments by following the given paths and generates the AnnotationProperty URI.
  - Input Examples: *$getAnnotationProperty(/AnnotationType /Namespace)*
  - Output examples: *rdfs:comment, owlgred:Container*

Some other functions use context information of the entire diagram, as well. There are listed in other function classification categories, by their main purposes. These functions are: getFSExpression, getClassExpr, getObjectUri, getHasKeyProperties, getAttributeKind, getTypeExpression, getFSDataTypeRestriction.

Perform a search for another structural element (by a specified path, condition)**:**

- ***getClassExpr(element Class).*** Returns a class expression for the given class element. The function can be called without arguments from a class element or its compartment context, or it can be called from explicitly specified (e.g. by a path expression) class element or a class element set. For each class element found, the function goes down to its *Name* compartment and generates the class URI. If the class is anonymous, i.e. there is no class name, (cf. class *Student or Teacher* in Figure 1), the function goes down in the compartment structure into the first *EquivalentClass* compartment and generates the axiom fragment (Barzdins et al., 2010c) from it. The decision, if a class is named or anonymous, and the class expression lookup are implemented within the function.
  - Input Examples: *$getClassExpr, $getClassExpr(/eEnd/start[$count > 1])*
  - Output examples: *:Student, ObjectUnionOf(:Student :Teacher)*
- ***getObjectUri(element Object).*** Returns an object URI for the given element. The function can be called without arguments from the current object element or its compartment context, or it can be called from explicitly specified (e.g. by a path expression) element or element set. For each object element found, the function goes down to the *Title* compartment, then down to the *Name* compartment and generates the object URI.
  - Input Examples: *$getObjectUri, $getObjectUri(/eEnd/start)*
  - Output examples: *:Dave*
- ***subject.*** Provides the compartment context support URI by calling the appropriate of functions: ***getUri***, ***getClassExpr***, ***getRoleExpr***, ***getObjectUri***. Function is used in the User Fields extension (Cerans et al. 2013) for the semantic pattern definition.

Perform a syntax transformation:

- ***getFSExpression(compartment Expression).*** Returns OWL Functional syntax fragment from the compartment containing expression written in OWL Manchester syntax. The function can be called without arguments from the current compartment context, or it can be called from an explicitly specified (e.g. by a path expression) compartment.

  - Input Examples: *$getFSExpression, $getFSExpression(/Expression)*
  - Output examples: *:Assistant, ObjectUnionOf(:Assistant :Associate_Professor :Professor), DataMaxCardinality(1 :courseName xsd:string)*
- ***getTypeExpression(compartment Type, compartment Namespace)***. Returns class attribute type (a data type name for data properties and a class name for object properties) in OWL Functional syntax. The function if called from an attribute compartment context, generates the type expression from its *Type* and *Namespace* sub-compartments, or the arguments can be specified explicitly. The function uses information from the entire ontology property and data type sets.
  - Input Examples: *$getTypeExpression(/Type /Namespace)*
  - Output examples: *xsd:string, :stringID*
- ***getFSDataTypeRestriction***. Returns DataTypeRestriction axiom fragment from the current data type compartment that contains the expression written in OWL Manchester syntax.
  - Input Examples: *$getFSDataTypeRestriction*
  - Output examples: *xsd:string xsd:pattern "[0-9]*"*

Aggregate information from multiple structural elements:
- ***getHasKeyProperties(string PropertyKind).*** Returns class *HasKey* expressions for the object or data properties. Function is always called from the current *Key* compartment context. The function receives input parameter that can be 'ObjectProperty' or 'DataProperty', it uses the entire ontology Object or Data property set.
  - Input Examples: *$getHasKeyProperties('ObjectProperty'), $getHasKeyProperties('DataProperty')*
  - Output examples: *courseName, ObjectInverseOf(:teaches) :enrolled*

Perform selection of certain types of information:
- ***getMultiplicity(string MultiplicityType)***. Returns cardinality fragment from the current multiplicity compartment context.
  - Input Examples: *$getMultiplicity('Exact'), $getMultiplicity('Min')*
  - Output examples: *0, 1*

Get supporting information:
- ***getAttributeKind(compartment Type, compartment isObjectAttribute)***. Checks, if the given class attribute is object or data property. Function is similar to the *getTypeExpression* function, just instead of calculating the type expression, it returns class attribute kind. Function returns a string expression with value 'ObjectProperty' or 'DataProperty' telling that the attribute is an object property or a data property within the ontology.
  - Input Examples: *$getAttributeType(/Type /isObjectAttribute), $getAttributeType(/../Type/Type /../isObjectAttribute)*
  - Output examples: *DataProperty, ObjectProperty*
- ***value(compartment Value)***. Returns a compartment value. The function can be called without arguments from the current compartment context, or it can be called from an explicitly specified (e.g. by a path expression) compartment.
  - Input Examples: *$value, $value(/ValueLanguage/Language)*

- Output examples: *Data type for IDValue, Assistant or Associate_Professor or Professor*

- ***elemType***. Returns an element type of the current context element.
- ***count***. Returns instance count of the given element set.
- ***getContainer***. Returns the name of the container where the current element is located.
- ***isURI***. Checks, if the compartment value is URI. Used for Annotation generation.
- ***isEmpty***. Checks, if the compartment exists.

The export pattern language functions uses an information from the so-called aspects – the lists of the different sorts of the ontology information.

- ***Object property list*** – the list of all object properties in the diagram.
- ***Object data list*** – the list of all data properties in the diagram.
- ***Class list*** – the list of all classes in the diagram.
- ***Data type list*** – the list of all data type in the diagram.
- ***Annotation prefixes definition*** – the predefined list of the annotation prefixes.
- ***Ontology prefixes definition*** – the list of all prefixes in the diagram.
- ***Built-in Datatype prefixes definition*** – the predefined list of the data type prefixes.

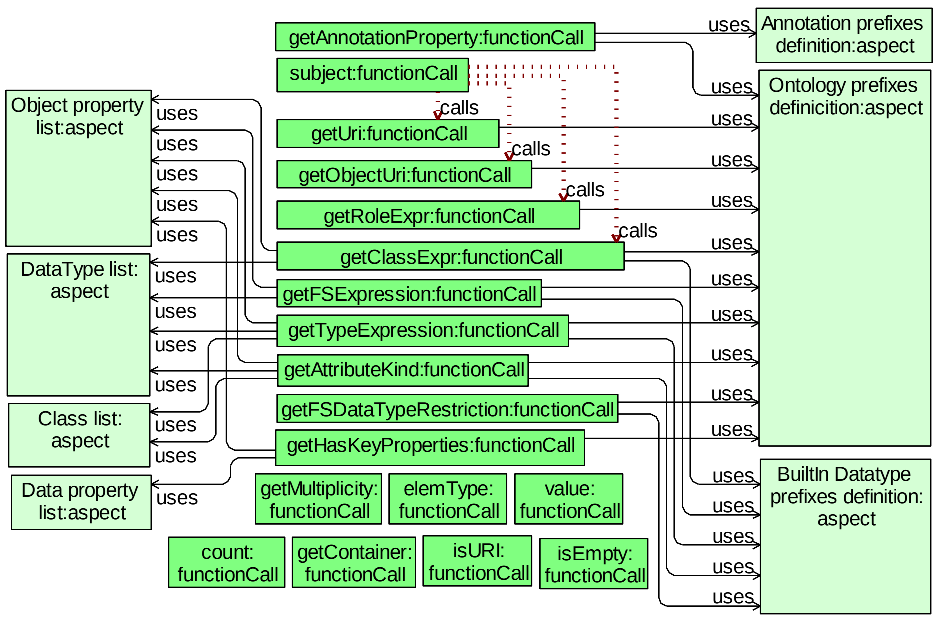

getAnnotationProperty:functionCall
subject:functionCall
getUri:functionCall
getObjectUri:functionCall
getRoleExpr:functionCall
getClassExpr:functionCall
getFSExpression:functionCall
getTypeExpression:functionCall
getAttributeKind:functionCall
getFSDataTypeRestriction:functionCall
getHasKeyProperties:functionCall
getMultiplicity: functionCall
elemType: functionCall
value: functionCall
count: functionCall
getContainer: functionCall
isURI: functionCall
isEmpty: functionCall
Annotation prefixes definition:aspect
Ontology prefixes definicition:aspect
BuiltIn Datatype prefixes definition: aspect
Object property list:aspect
DataType list: aspect
Class list: aspect
Data property list:aspect
uses
calls

Fig. 6. Export pattern functions and aspects interconnection

Figure 6 shows interconnection of the export pattern functions and aspects. For example, to translate an expression from the OWL Manchester syntax into functional syntax, function ***getFSExpression*** uses an ontology prefix definitions, a list of object properties, a list of data types and a built in data type prefixes definitions.

Figure 7 illustrates the generation of the Class declaration axiom. Generation of the Class declaration axiom starts from the compartment that is connected to the *compartType* whose id equal to "Name". This compartment type has an export pattern, stored in a related *Tag* instance. The export pattern for the class declaration is defined as the following string: "Declaration(Class($getUri(/Name /Namespace)))".

The first part of the export pattern: "**Declaration(Class(**" is a text fragment construction and it will be translated into OWL axiom as it is. The function **getUri** has two arguments /Name and /Namespace indicating that from the current Name compartment we need to go one level down in the compartment tree structure and find sub-compartments connected to the compartment types whose id are equal to "Name" and "Namespace" respectively. Then function getUri, based on these two compartment values, generates the class URI. Since the namespace compartment is empty, the ontology local namespace is used. The export pattern ends with another text fragment construction "**))**", and the class declaration axiom is completed.

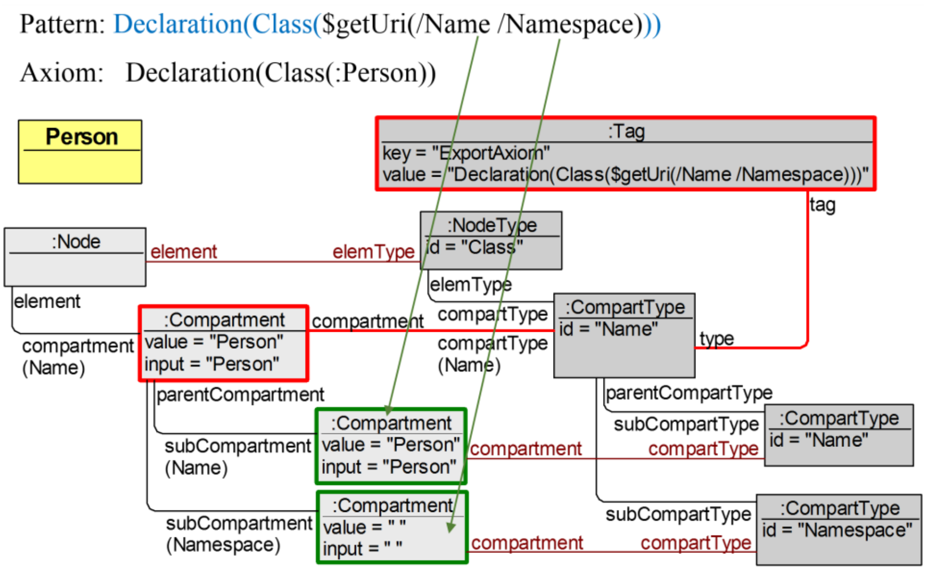


Fig. 7. Export pattern example for class declaration

The described OWL Functional syntax generation patterns using the defined functions allow to generate from Figure 1 ontology diagram the OWL Functional syntax text, as shown in Appendix 1.

Since the text generation patterns are defined on the level of the abstract syntax of the ontology editor, they could be applied also for other notations that are defined on the basis of GrTP/TDA tool building metamodel. Such notations, besides the OWLGrEd editor, include e.g. UML Class, Activity, UseCase and StateChart diagrams, as developed within the GradeTwo tool (WEB, c)

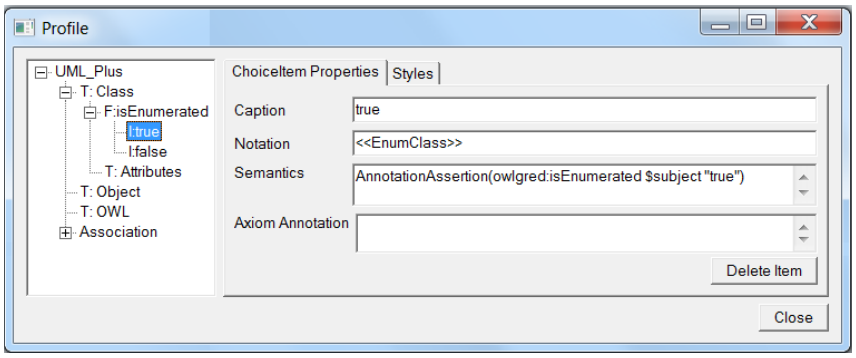
4. OWLGrEd Extensions

The OWLGrEd ontology editor provides an option to its end users to define extensions to its visual symbol appearance (Cerans et.al., 2013), (Cerans et.al., 2019). The ontology export pattern language, as described in Section 3, can be used in semantics definition for user-defined fields in OWLGrEd editor extensions. The user that configures the extension, writes the semantics expression herself, using the pattern language described here.

In general, the editor extensions with symbol fields and graphical effects enhance the ontology presentation options by introducing domain-specific notations into the ontology presentation; they are handled by a generic User Fields extension (Cerans et al., 2013) that is currently part of the default OWLGrEd editor configuration.

The available editor enhancements, supported by the User Fields extension include:

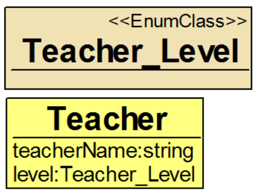
- custom fields, together with their semantics mappings (e.g. “enumerated class”);
- custom visual effects for text and choice fields and symbols dependent on concrete text or choice field values (e.g. a brown/darker enumerated class color);
- views applying certain visual effects to the entire diagram (e.g. hiding certain information from the presentation).



Caption: true
Notation: <<EnumClass>>
Semantics: AnnotationAssertion(owlgred:isEnumerated $subject "true")
Axiom Annotation:

Fig. 8. User Fields extension dialog for semantic pattern definition

The user-defined fields are created by filling in instances of a User Fields configuration model described in (Cerans et al., 2013), including for each field user can specify the corresponding semantics expression pattern. Figure 8 demonstrates the User Fields extension dialog for the semantic pattern definition, involving also field semantics definition within the described pattern language.



```
Declaration(Class(:Teacher))
Declaration(Class(:Teacher_Level))
Declaration(ObjectProperty(:level))
Declaration(DataProperty(:teacherName))
Declaration(AnnotationProperty(owlgred:isEnumerated))
ObjectPropertyDomain(:level :Teacher)
ObjectPropertyRange(:level :Teacher_Level)
DataPropertyDomain(:teacherName :Teacher)
DataPropertyRange(:teacherName xsd:string)
AnnotationAssertion(owlgred:isEnumerated :Teacher_Level "true")
```

Fig. 9. An ontology example with custom user-defined field

Figure 9 shows, an ontology diagram example with the custom user-defined field, as well as OWL Functional syntax axioms generated from it. A Teacher_Level class has check-box field “isEnumerated” for enumerated classes that, when checked, attaches the stereotype <<EnumClass>> to the class symbol and changes its background color to slightly darker (orange-brown). The user-defined field can be associated with the export pattern that is translated into this field during the ontology import and that is produced during the ontology saving in the textual notation. The User Fields extension in the export uses the same pattern language as the basic OWLGrEd export. For instance, the stereotype at the Teacher_Level class has the export pattern attached to it: AnnotationAssertion(owlgred:isEnumerated $subject "true"), and is saved to the: AnnotationAssertion(owlgred:isEnumerated :Teacher_Level "true") OWL axiom.

5. Conclusions

The article has shown the possibility to create the visual OWLGrEd ontology export into the textual format by means of a structured export process, split over diagram syntactic elements.

For each visual diagram element, the corresponding OWL Functional syntax axiom text is generated by a simple pattern language, built over a small set of implemented context lookup, abstract syntax navigation and text transformation primitives. The principal classes of functions necessary for the transformation have been: use context information of the entire diagram, perform a search for another structural element, perform a syntax transformation, aggregate information from multiple structural elements, perform selection of certain types of information and get supporting information.

Declarative definitions of the correspondence between the OWLGrEd constructs and the OWL axioms relevant parts allows any diagram (even if the diagram is not complete) to be interpreted as an OWL ontology.

The modular architecture of the OWL axiom text exporter is important for further definition of OWLGrEd extensions that are principal in OWLGrEd practical applications (cf. (Cerans et al., 2019)). The architecture enables maintaining the OWLGrEd editor, as well (e.g. by introducing the changes to the configuration structure).

Described pattern-based language can be used outside the OWLGrEd tool, for the translation of other graphical models, like a possible graphical notation of SHACL language, if implemented within GrTP or ajoo platform, into the textual form.

The pattern-based structural OWLGrEd export implementation is expected to support also the envisaged migration of the editor to a more user-friendly web environment: only the generic language interpreter and the specific function implementations would need to be migrated from the current Lua/lQuery environment to the JavaScript/jQuery to achieve the ontology export from the editor in the new environment.

References

Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A. (2010a). OWLGrEd: a UML Style Graphical Notation and Editor for OWL 2. In *Proc. of OWLED 2010*, 2010

Barzdins, J., Cerans, K., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A. (2010b). An MDE-Based Graphical Tool Building Framework. *Scientific Papers, University of Latvia*, Vol. 756. Computer Science and Information Technologies. Riga, Latvia, 2010, pp. 121–138.

Barzdins, J., Cerans, K., Kozlovics, S., Rencis, E., Zarins, A. (2009). A Graph Diagram Engine for the Transformation-Driven Architecture. In *Proc. of Workshop on Model Driven Development of Advanced User Interfaces (IUI 2009)*. Florida, USA.

Barzdins, J., Cerans, K., Liepins, R., Sprogis, A. (2010c). UML Style Graphical Notation and Editor for OWL 2. In *Proc. of BIR'2010*, LNBIP, Springer 2010, Vol. 64, p. 102-113.

Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins R., Sprogis, A. (2007). GrTP: Transformation Based Graphical Tool Building framework, In *Proc. of MoDELS 2007 Workshop on Model Driven Development of Advanced User Interfaces*; Nashville, TN; USA.

Cerans, K., Romane, A. (2015). OBIS: Ontology-Based Information System Framework. *CAiSE Forum, CEUR Workshop Proceedings*, Vol.1367, pp.65-72

Cerans, K., Ovcinnikova, J., Liepins, R., Grasmanis, M. (2019). Extensible Visualizations of Ontologies in OWLGrEd. *ESWC*, 2-6 June, Portorož, Slovenia

Cerans, K., Ovcinnikova, J., Liepins, R., Sprogis, A. (2013). Advanced OWL 2.0 Ontology Visualization in OWLGrEd // Caplinskas, A., Dzemyda, G., Lupeikiene, A., Vasilecas, O. (eds.) *Databases and Information Systems VII*, IOS Press, Frontiers in Artificial Intelligence and Applications, Vol. 249, pp.41-54, 2013

Cerans, K., Sostaks, A., Bojars, U., Barzdins, J., Ovcinnikova, J., Lace, L., Grasmanis, M., Sprogis, A. (2018). ViziQuer: A Visual Notation for RDF Data Analysis Queries. *MTSR, Metadata and Semantic Research*, Springer Verlag CCIS Series, Vol. 846, pp. 50-62.

Chis-Ratiu, A., Buchmann, R.A. (2018). Design and Implementation of a Diagrammatic Tool for Creating RDF graphs. *PrOse@PoEM*.

Dudás, M., Lohmann, S., Svátek, V., Pavlov, D. (2018). Ontology visualization methods and tools: a survey of the state of the art. *Knowledge Eng. Review, 33*, e10.

Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z. (2014). HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning, 53*, 245-269.

Horridge, M., Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. *Semantic Web, 2*, 11-21.

Horridge, M., Peter, F. (2012). *OWL 2 Web Ontology Language Manchester Syntax (Second Edition).*Available at `http://www.w3.org/TR/owl2-manchester-syntax/`

Kats, L.C., Visser, E. (2010). The spoofax language workbench: rules for declarative specification of languages and IDEs. // Rinard, M. (eds.), In *Proc. of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM.

Knublauch, H., Kontokostas, D. (2017). Shapes Constraint Language (SHACL). Available at `https://www.w3.org/TR/shacl/`

Liepins, R. (2012). Library for model querying: IQuery. In *Proc. of 12th Workshop on OCL and Textual Modeling, OCL 2012* - Being Part of the ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012; Innsbruck; Austria.

Lohmann, S., Negru, S., Haag F., Ertl, T. (2016). Visualizing Ontologies with VOWL. *Semantic Web 7(4)*, 399-419.

Motik, B., Patel-Schneider, P.F., Parsia, B. (2012). *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax.* Available at `https://www.w3.org/TR/owl2-syntax/`

Mouromtsev, D., Pavlov, D., Emelyanov, Y., Morozov, A., Razdyakonov, D., Galkin, M. (2015). The Simple Web-based Tool for Visualization and Sharing of Semantic Data and Ontologies. *International Semantic Web Conference*.

Ovcinnikova, J., Cerans, K. (2016). Advanced UML Style Visualization of OWL Ontologies. In *Proc. of VOILA 2016*. CEUR, Vol. 1704, CEUR-WS.org, 2016, pp.136-142.

Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *J. Web Semant., 5*, 51-53.

Sprogis. A. (2016). ajoo: WEB Based Framework for Domain Specific Modeling Tools. // *Frontiers of AI and Applications*, Vol. 291, Databases and Information Systems IX, IOS Press, pp. 115-126, Available at `http://ebooks.iospress.com/volumearticle/45704`

Tsarkov, D., Horrocks, I. (2006). FaCT++ Description Logic Reasoner: System Description. *IJCAR*, *Lecture Notes in Computer Science*, Vol. 4130. Springer, Berlin, Heidelberg.

Voelter, M. (2006). oAW xText: A framework for textual DSLs. *Modeling Symposium at Eclipse Summit* 2006.

Zviedris, M., Romane, A., Barzdins, G., Cerans, K. (2013). Ontology-Based Information System. *In JIST. Springer Lecture Notes in Computer Science*, Vol.8388, pp.33-47 (2013).

WEB (a). *Acceleo Query Language (AQL)*. `https://www.eclipse.org/acceleo/documentation/`

WEB (b). *Epsilon Generation Language*. `https://www.eclipse.org/epsilon/doc/book/`

WEB (c). *GradeTwo Tool*. `http://gradetwo.lumii.lv/`

WEB (d). *JavaScript*. `https://www.javascript.com/`

WEB (e). *JET*. `https://www.vogella.com/tutorials/EclipseJET/article.html`

WEB (f). *JQuery*. `https://jquery.com/`

WEB (g). *MOFScript*. `https://marketplace.eclipse.org/content/mofscript-model-transformation-tool`

WEB (h). *ODM UML profile for OWL*. `http://www.omg.org/spec/ODM/1.0/PDF/`

WEB (i). *OWLViz*. `http://www.co-ode.org/downloads/owlviz/`

WEB (j). *Protégé*. `https://protege.stanford.edu/`

WEB (k). *The Programming Language Lua*. `https://www.lua.org/`

WEB (l). *TopBraid Composer*. `http://www.topquadrant.com/tools/modeling-topbraid-composer-standard-edition/`

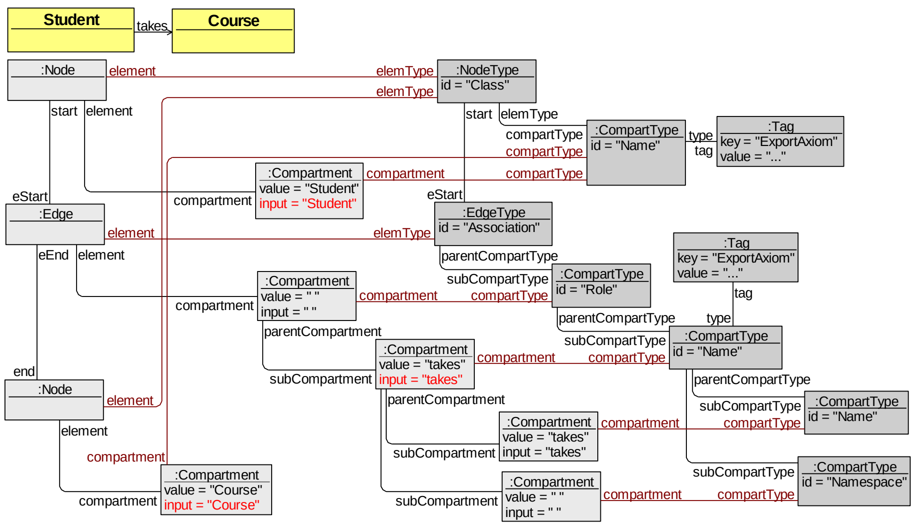
WEB (m). *Xpand*. `https://www.eclipse.org/modeling/m2t/?project=xpand`

Received June 11, 2020, revised September 3, 2020, accepted September 5, 2020

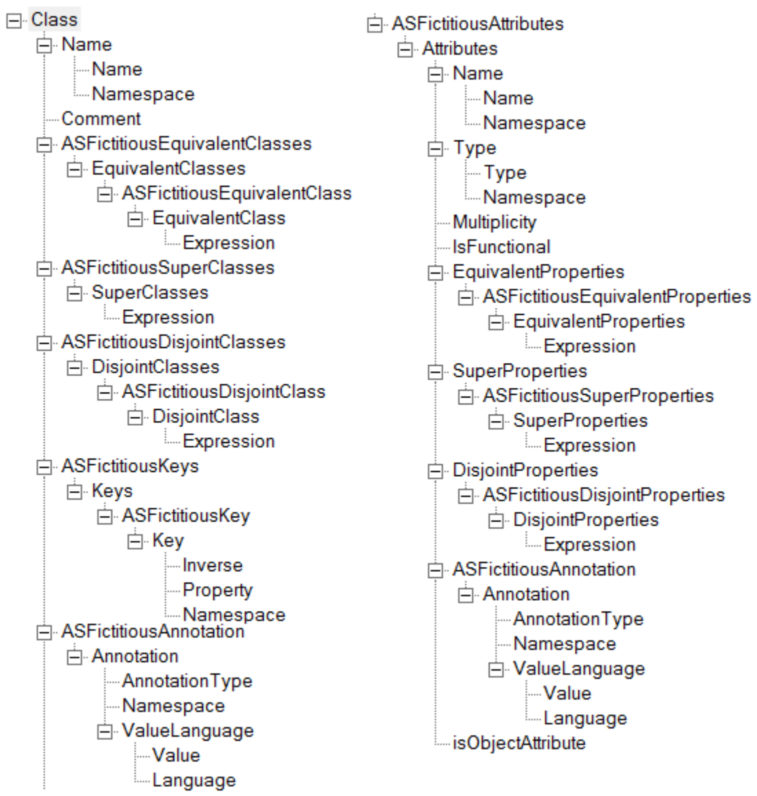
Appendix 1.
A mini-University ontology in OWL Functional syntax

```
Prefix(:=<http://lu.lv/kc/2010/7/MiniUniv.owl#>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSche
ma#>)

Ontology(<http://lu.lv/kc/2010/7/MiniUniv.owl>

Declaration(Class(:Assistant))
Declaration(Class(:Associate_Professor))
Declaration(Class(:Course))
Declaration(Class(:Mandatory_Course))
Declaration(Class(:Optional_Course))
Declaration(Class(:Person))
Declaration(Class(:Professor))
Declaration(Class(:Student))
Declaration(Class(:Teacher))
Declaration(ObjectProperty(:isTakenBy))
Declaration(ObjectProperty(:isTaughtBy))
Declaration(ObjectProperty(:relates))
Declaration(ObjectProperty(:takes))
Declaration(ObjectProperty(:teaches))
Declaration(DataProperty(:courseCade))
Declaration(DataProperty(:courseCredits))
Declaration(DataProperty(:courseName))
Declaration(DataProperty(:personID))
Declaration(DataProperty(:personName))
Declaration(DataProperty(:salary))
Declaration(DataProperty(:studentName))
Declaration(DataProperty(:studentNumber))
Declaration(DataProperty(:teacherName))
Declaration(NamedIndividual(:John))
Declaration(Datatype(:sting))
SubClassOf(:Assistant :Teacher)
SubClassOf(:Associate_Professor :Teacher)
EquivalentClasses(:Course
ObjectUnionOf(:Optional_Course
:Mandatory_Course))
SubClassOf(:Mandatory_Course :Course)
SubClassOf(:Mandatory_Course
ObjectAllValuesFrom(:isTaughtBy :Professor))
DisjointClasses(:Mandatory_Course
:Optional_Course)
SubClassOf(:Optional_Course :Course)
SubClassOf(:Professor :Teacher)
DisjointClasses(:Professor :Student)
SubClassOf(:Student :Person)
SubClassOf(:Student ObjectUnionOf(:Student
:Teacher))
SubClassOf(:Teacher :Person)
SubClassOf(:Teacher ObjectUnionOf(:Assistant
:Associate_Professor :Professor))
SubClassOf(:Teacher ObjectUnionOf(:Teacher
:Student))
InverseObjectProperties(:isTakenBy :takes)
ObjectPropertyDomain(:isTakenBy :Course)
ObjectPropertyRange(:isTakenBy :Student)
InverseObjectProperties(:isTaughtBy :teaches)
ObjectPropertyDomain(:isTaughtBy :Course)
ObjectPropertyRange(:isTaughtBy :Teacher)
ObjectPropertyDomain(:relates
ObjectUnionOf(:Teacher :Student))
ObjectPropertyRange(:relates :Course)
SubObjectPropertyOf(:takes :relates)
ObjectPropertyDomain(:takes :Student)
ObjectPropertyRange(:takes :Course)
DisjointObjectProperties(:takes :teaches)
SubObjectPropertyOf(:teaches :relates)
ObjectPropertyDomain(:teaches :Teacher)
ObjectPropertyRange(:teaches :Course)
DataPropertyDomain(:courseCade :Course)
DataPropertyRange(:courseCade xsd:string)
DataPropertyDomain(:courseCredits :Course)
DataPropertyRange(:courseCredits xsd:integer)
DataPropertyDomain(:courseName :Course)
DataPropertyRange(:courseName xsd:string)
DataPropertyDomain(:personID :Person)
DataPropertyRange(:personID :sting)
DataPropertyDomain(:personName :Person)
DataPropertyRange(:personName xsd:string)
DataPropertyDomain(:salary :Teacher)
DataPropertyRange(:salary xsd:integer)
DataPropertyDomain(:studentName :Student)
DataPropertyRange(:studentName xsd:string)
DataPropertyDomain(:studentNumber :Student)
DataPropertyRange(:studentNumber xsd:string)
DataPropertyDomain(:teacherName :Teacher)
DataPropertyRange(:teacherName xsd:string)
ClassAssertion(:Professor :John)
DisjointClasses(:Assistant :Associate_Professor
:Professor)
)
```

Appendix 2. OWLGrEd abstract syntax example



Appendix 3. OWLGrEd class node type structure



```
Class
  Name
    Name
    Namespace
  Comment
  ASFictitiousEquivalentClasses
    EquivalentClasses
      ASFictitiousEquivalentClass
        EquivalentClass
          Expression
  ASFictitiousSuperClasses
    SuperClasses
      Expression
  ASFictitiousDisjointClasses
    DisjointClasses
      ASFictitiousDisjointClass
        DisjointClass
          Expression
  ASFictitiousKeys
    Keys
      ASFictitiousKey
        Key
          Inverse
          Property
          Namespace
  ASFictitiousAnnotation
    Annotation
      AnnotationType
      Namespace
      ValueLanguage
        Value
        Language

ASFictitiousAttributes
  Attributes
    Name
      Name
      Namespace
    Type
      Type
      Namespace
    Multiplicity
    IsFunctional
    EquivalentProperties
      ASFictitiousEquivalentProperties
        EquivalentProperties
          Expression
    SuperProperties
      ASFictitiousSuperProperties
        SuperProperties
          Expression
    DisjointProperties
      ASFictitiousDisjointProperties
        DisjointProperties
          Expression
    ASFictitiousAnnotation
      Annotation
        AnnotationType
        Namespace
        ValueLanguage
          Value
          Language
    isObjectAttribute
```