A Graphical Tool for Automatic Generation of OWL Ontologies

Aissam BELGHIAT
Department of Computer Science
University of Md Boudiaf, Msila, 28000, Algeria
Belghiatissam@gmail.com

Mustapha BOURAHLA
Department of Computer Science
University of Md Boudiaf, Msila, 28000, Algeria
mbourahla@hotmail.com

***Abstract*—The ontologies are became the backbone of the semantic web, the development of these ontologies from scratch is a very difficult task. The use of models is useful to deal with such problems since it allows visually modeling of the ontologies themselves. In this paper, we propose an approach based on the combined use of Meta-modeling and Graph Grammars to automatically generate a visual modeling tool for OWL ontologies. In our approach registered in the MDA architecture, the UML Class diagram formalism is used to define a meta-model of class diagrams. The meta-modeling tool ATOM3 is used to generate a visual modeling tool according to the proposed class diagram meta-model. We have also proposed a graph grammar to automatically generate OWL ontologies of the graphically specified class diagram models. This allows the user to pass a very important step in the development process of OWL ontologies. Our environment is illustrated through an example.**

***Index Terms*—UML, Ontology, OWL, ATOM3, MDA.**

I. INTRODUCTION

The ontologies are became the backbone of the semantic web, this last provides various languages for representing ontologies including XML, RDF, DAML, and OWL. OWL (Ontology Web Language) is the most widely used of these languages because of its high expressive power and the fact that it is the W3C standard ontology language for the Semantic Web [10].

In the other hand, UML is the unified object oriented modeling language which became an important standard widely used by domain experts for expressing their domain knowledge. The development of OWL ontologies from scratch is a very difficult task. The use of models is useful to deal with such problems since it allows visually modeling of the ontologies themselves. An OWL ontology visualized as UML diagrams allow us to better describe our ontology to humans, and most people can understand an UML class diagram at a glance. In this paper we propose an approach based on the combined use of Meta-modeling and Graph Grammars to automatically generate a visual modeling tool for OWL ontologies. We implement these concepts presented above in ATOM3: A Tool for Multi-formalism and Meta-Modeling [1]. We propose an UML class diagram meta-model and we use the meta-modeling tool AToM3 to generate automatically a visual modeling tool to process UML class diagram models. We also define a graph grammar to translate the models created in the generated tool to OWL ontologies represented in RDF/XML format.

The rest of the paper is organized as follows: In Section 2, we present some related works. In Section 3, we present some basic notions about UML, OWL. In Section 4, we present concepts about model and graph transformation, and then we give an overview of the AToM3 tool [1]. In Section 5, we describe our approach that provides a graphical environment for automatic generation of OWL ontologies. In Section 6, we illustrate our tool through an example. Finally concluding remarks drawn from the work and perspectives for further research are presented in Section 7.

II. RELATED WORKS

The idea of our work is not innovating, indeed several works exist in the literature tackle this subject. In [14] the authors proposed a transformation of UML towards DAML at the end of the Nineties, by showing similarities and differences between the two languages. In [15] the work of "Converting UML to OWL Ontologies" proposed a transformation of Ontology UML Profile (OUP) towards an ontology OWL. In [6] the OMG notices the interest of such subject and proposed in its turn the ODM which provides a profile for writing RDF and OWL within UML, it also includes partial mappings between UML and OWL as well as mappings amongst RDF, RDFS, Common Logic and Topic Maps, it should be noted that several works are carried out like answer to the call of the OMG and gathered in the ODM that we do not evoke here. In [9], the author presented an implementation of the ODM using ATL language. In [5], the author used a style sheet "OWLfromUML.xsl" applied to an XMI file (intermediate format of UML model) to generate an ontology OWL DL represented as RDF/XML format. And finally in [16], the authors proposed a detailed comparison between UML and OWL that carried out in 2008. In the other side Atom3 has been proven to be a very powerful tool allowing the meta-modeling and the transformations between formalisms, in [1] and other works we can found treatment of class diagrams, activity, and other UML diagrams. In these works the Meta modeling allows visual modeling and graph grammar allows the transformation.

Obviously, the heart of our work is articulated on transformation rules and their implementation. In pre-ceding

works, the transformation rules are more specific and reflect a general opinion of the author often related to a specific field which he works on (specific transformation). In this paper we propose another vision different from that approached in preceding works either in the proposition of transformation rules, or in theirs implementation, this vision is to propose the transformation rules in a level of abstraction close to the application in order to obtain usable ontologies, because more the selected level of abstraction is close to the application minus ontology is reusable, but more it is usable. Then we propose a graph grammar implementation for these rules.

III. OWL FROM UML CLASS DIAGRAM

UML (Unified Modeling Language) is a language to visualize, specify, build and document all the aspects and artifacts of a software system [7]. UML defines thirteen diagrams; some of them represent the system statically while others show the functionalism of the system. The class diagram is considered very important for object oriented modeling; it shows the internal structure of a system and makes it possible to provide an abstract representation of its objects [2].

OWL (Ontology Web Language), was recommended by the W3C in 2004, and its version 2 in 2009, is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL1 offers three sub-languages with increasing expression intended for specific communities of developers and users: OWL Lite, OWL DL, and OWL Full [10] whereas OWL2 defines three new profiles: OWL2 EL, OWL2 QL, and OWL2 RL [13].

UML and OWL have different goals and approaches; however they have some overlaps and similarities, especially for representation of structure (class dia-grams). UML and OWL comprise some components which are similar in several regards, like: classes, associations, properties, packages, types, generalization and instances [6]. UML is a notation for modeling the artifacts of objects oriented software, whereas OWL is a notation for knowledge representation, but both are modeling languages.

IV. GRAPH TRANSFORMATION

A. Overview

Modeling and model transformation play an essential role in the MDA "Model Driven Architecture". MDA recommends the massive use of models in order to allow a flexible and iterative development, thanks to refinements and enrichments by successive transformations.

A model transformation is a set of rules that allows passing from a meta-model to another, by defining for each one of elements of the source their equivalents among the elements of the target. These rules are carried out by a transformation engine; this last reads the source model which must be conform to the source meta-model, and applies the rules defined in the model transformation to lead to the target model which will be itself conform to the target meta-model. The principle of model transformation is illustrated by figure 1:

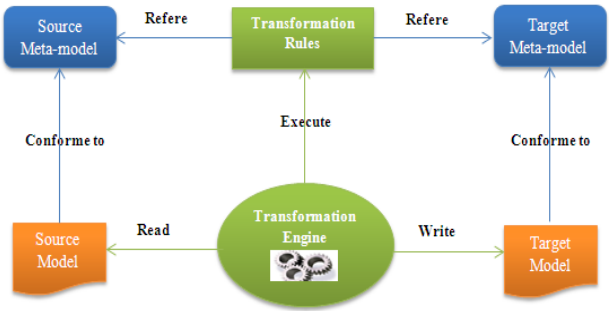


Fig. 1. Model transformation principle.

Graph transformation was largely used for the expression of model transformation [4]. Particularly transformations of visual models can be naturally formulated by graph transformation, since the graphs are well adapted to describe the fundamental structures of models.

The set of graph transformation rules constitutes what is called the model of graph grammar; each rule of a graph grammar is composed of a left hand side (LHS) pattern and of a right-hand sided (RHS) pattern.

Therefore, the graph transformation is the process to choose a rule among the graph grammar rules, apply this rule on a graph pattern that is matched with the LHS pattern to produce the RHS pattern, and reiterate the process until no rule can be applied [4].

B. AToM3

AToM3 [1] "A Tool for Multi-formalism and Meta-Modeling" is a visual tool for model transformation, written in Python [8] and is carried out on various platforms (Windows, Linux, …). It implements various concepts like multi-paradigm modeling, meta-modeling and graph grammars. It can be also used for simulation and code generation.

AToM3 provides visual models those are conform to a specific formalism, and uses the graph grammar to go from a model to another.

In the next sections, we will discuss how we use AToM3 to meta-model class diagrams and how to generate OWL models by applying a graph grammar.

V. OUR APPROACH

Our solution is implemented in AToM3. Our choice is quickly related to AToM3 because of the advantages which it presents like its simplicity, and its availability. In our approach we have to profit from the convergence between UML and OWL especially for representation of structure (class diagrams), most people can model and understand UML class diagram at a glance, it is not the case in different representations of ontologies like RDF/XML, Turtle or other representations. We have to benefit of visualizing OWL ontologies as UML class diagram models.

For the realization of this application we have to propose and to develop a meta-model of class diagram (figure 2), this meta-model allows us to edit visually and with simplicity class diagrams on AToM3 canvas. In addition to meta-model

proposed we develop a graph grammar made up of several rules which allows transforming progressively all what is modeled on the canvas towards an OWL ontology stored in a disk file (fig.2). The graph grammar is based on transformation rules; those rules try to transform the class diagram in the implementation level, always in order to obtain at the end usable ontologies.

For the ontology, the choice among OWL profiles is made on OWL DL because it places certain constraints on the use of the structures of OWL such as separation two to two between classes, datatypes, datatype properties, objects properties, annotation properties, ontologies properties, individuals, data values, and integrated vocabulary [11]. That means, for example, a class cannot be at the same time an individual [12]. These constraints enable us to lead to our objective which is an ontology well reflecting what is modeled in a class diagram.

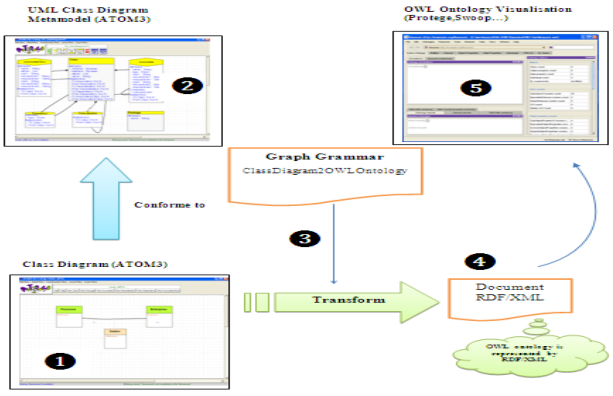


Fig. 2. Transformation sequence.

A. Transformation rules

Our approach is realized according to suggested transformation rules (Table I). We propose a set of rules in particular for classes' transformation, enumerations, associations, roles, dependencies, association classes, and almost all the elements of a class diagram. The level of abstraction in those rules is close to the application in order to have usable ontologies. For lack of space, we have presented some of these rules.

TABLE I. UML TO OWL TRANSFORMATION RULES.

Class
An UML class is transformed to an OWL class; the name of the class is preserved.
Inheritance
The specialized class is defined subclass of the generalized class.
Class Attributes
An attribute is transformed into a property, and the transformation is carried out according to the type of attribute. If the type of the attribute is a primitive type, the attribute is transformed into datatype property. If the value of the attribute is a class, it is transformed into object property.
Bidirectional association
Associations are transformed into object properties. An inverse object property is generated automatically named (Inverse-associationname).
Association class
An association-class is transformed to OWL class (implementation level), named (ac-ssociationclassname). The latter is connected to the left part by a relation named (AG_AC-associationclassname), and to the right part by a relation named (AD_AC-associationclassname). We named also the two new roles on the two new association ends (RG_AC- associationclassname) and (RD_AC- associationclassname). After these transformations on the association class we find ourselves on the situation of transformation of binary associations (which is treated previously).

B. Datatypes transformation

UML data types are transformed into XML schema (XSD) data types because OWL uses the majority of the datatypes integrated into XML schema. The calls of these datatypes are done through datatype URI address reference http://www.w3.org/2001/XMLSchema [11]. The instances of the primitive types used in UML itself include: Boolean, Integer, String, and UnlimitedNatural [7]. Table II presents the UML primitive datatypes and their transformations.

TABLE II. DATATYPES TRANSFORMATION.

UML	XSD
Integer	xsd:integer
Boolean	xsd:boolean
String	xsd:string
UnlimitedNatural	xsd:nonNegativeInteger xsd:positiveInteger

C. Meta-model of UML Class diagram

To build UML class diagram models in AToM3, we have to define a meta-model for them. Our meta-model is composed of two classes and four associations developed by the meta-formalism (CD_classDiagramsV3), and the constraints are expressed in Python [8] code (fig. 3):

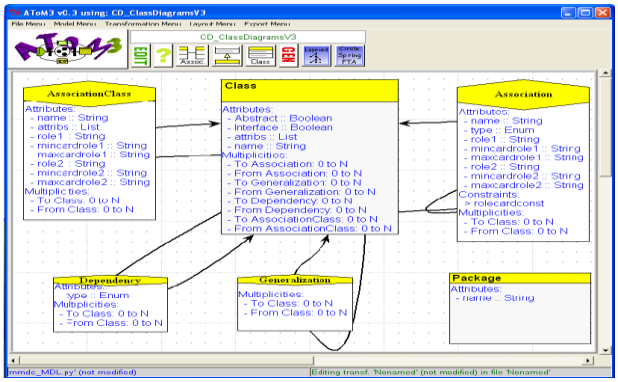


Fig. 3. Class diagram meta-model.

After we built our meta-model, it remains only its generation. The generated meta-model comprises the set of classes modeled in the form of buttons which are ready to be employed for a possible modeling of a class diagram.

D. The Proposed Graph grammar

To perform the transformation between class diagrams and OWL ontologies, we have proposed a graph grammar composed of an initial action, ten rules, and a final action. For lack of space, we have not presented all the rules.

Initial Action: Ontology header

Role: In the initial action of the graph grammar, we created a file with sequential access in order to store generated OWL code. Then we begin by writing the ontology header which is fixed for all our generated ontologies (fig. 4).

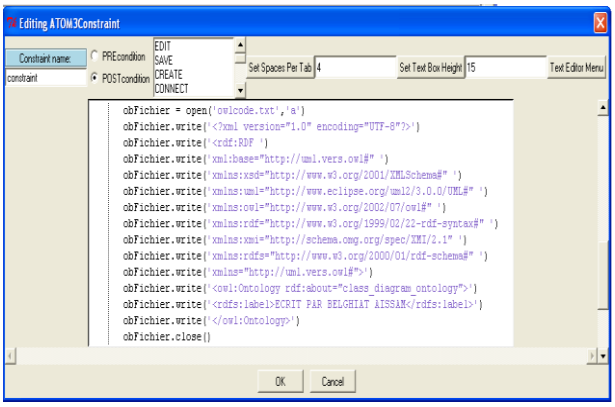


Fig. 4. Ontology header definition.

Rule 1: Class transformation

Name: class2class

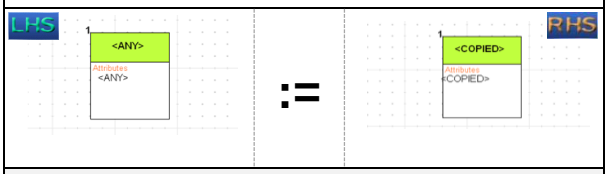
Priority: 1

Role: This rule transforms an UML class towards an OWL class (see Table III). In the condition of the rule we test if the class is already transformed, if not, in the action of the rule we reopen the OWL file to add the OWL code of this class.

TABLE III. CLASS TRANSFORMATION.

Condition

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not hasattr(node, "rule executed")
```



Action

```
    node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
    classname = node.name.getValue()
    node.rule_executed = True
    abst = node.Abstract.getValue()[1]
    interf = node.Interface.getValue()[1]
    if abst == 1:
        self.getMatched(graphID,
self.LHS.nodeWithLabel(1)).name.setValue('Abstract-'+classname)
    elif interf == 1:
        self.getMatched(graphID,
self.LHS.nodeWithLabel(1)).name.setValue('Interface-'+classname)
    obFichier = open('owlcode.txt','a')
    node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
    classname = node.name.getValue()
    obFichier.write('<owl:Class rdf:ID='+'"'+classname+'"'+'/>')
    obFichier.close()
```

Rule 2: Generalization transformation

Name: gener2sclass

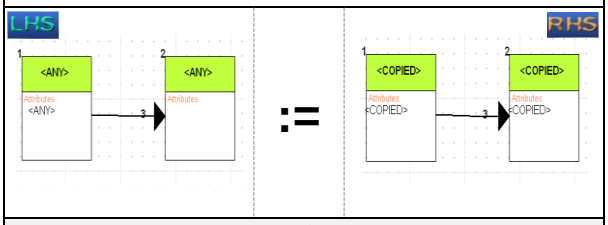
Priority: 2

Role: This rule transforms a generalization by indicating that the specialized class is defined as sub-class of the generalized class (Table IV).

TABLE IV. GENERALIZATION TRANSFORMATION.

Condition

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
gen = self.graphRewritingSystem.parent.ASGroot.listNodes['Generalization']
return gen != [] and not hasattr(node, "rule4 executed")
```



Action

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
node.rule4_executed = True
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
class1name = node.name.getValue()
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
class2name = node.name.getValue()
obFichier = open('owlcode.txt','a')
obFichier.write('<owl:Class rdf:ID="'+class1name+'">')
obFichier.write('<rdfs:subClassOf rdf:resource="#'+class2name+'"/>')
obFichier.write('</owl:Class>')
obFichier.close()
```

Rule 3: Binary association transformation

Name: asso2prop

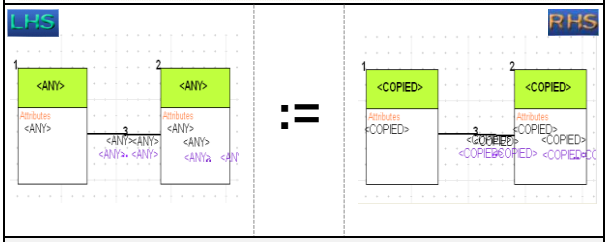
Priority: 3

Role: This rule transform an association of the class diagram towards an OWL object property, it allows also the transformation of roles and cardinalities of this association.

TABLE V. ASSOCIATION TRANSFORMATION.

Condition

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
return not hasattr(node, "rule4_executed")
```

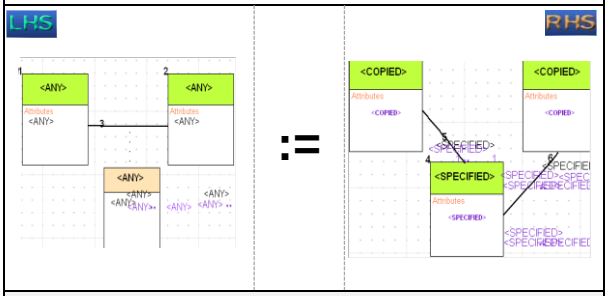


Action

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
assoname = node.name.getValue()
typ = node.type.getValue()[1]
node.rule4_executed = True
role1name = node.role1.getValue()
role2name = node.role2.getValue()
role1min = node.mincardrole1.getValue()
role1max = node.maxcardrole1.getValue()
role2min = node.mincardrole2.getValue()
role2max = node.maxcardrole2.getValue()
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
class1name = node.name.getValue()
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
class2name = node.name.getValue()
######################## Transformation des associations ##########################
if typ == 0 or typ == 2:# association bidirectionelle ou aggregation
    obFichier = open('owlcode.txt','a')
    obFichier.write('<owl:ObjectProperty rdf:ID="'+assoname+'">')
    obFichier.write('<rdfs:domain rdf:resource="#'+class1name+'"/>')
    obFichier.write('<rdfs:range rdf:resource="#'+class2name+'"/>')
    obFichier.write('<owl:inverseOf rdf:resource="#Inverse-'+assoname+'"/>')
    obFichier.write('</owl:ObjectProperty>')
    obFichier.write('<owl:ObjectProperty rdf:ID="Inverse-'+assoname+'">')
    obFichier.write('<rdfs:domain rdf:resource="#'+class2name+'"/>')
    obFichier.write('<rdfs:range rdf:resource="#'+class1name+'"/>')
    obFichier.write('<owl:inverseOf rdf:resource="#'+assoname+'"/>')
    obFichier.write('</owl:ObjectProperty>')
    obFichier.close()
elif typ == 1 or typ == 3:# association unidirectionelle ou composition
```

Rule 4: **Association-class transformation**
Name: ac2class
Priority: 1
Role: This rule allows the promotion of association class to a full class (see Table VI), that reflects what we show in the transformation rules. This class takes as name the name of the LHS class-association preceded by (AC-). Two binary associations are created in the RHS named AG_AC, AD_AC, thus two new roles RG_AC and RD_AC as illustrated in the transformation rules.

TABLE VI. ASSOCIATION-CLASS TRANSFORMATION.

Condition
No condition.

Action
No action

Final Action: **Definition of the end of ontology**
Role: In the final action of the graph grammar, we end our ontology, we will have to open our file and to add '</rdf:RDF>' (see fig. 5).

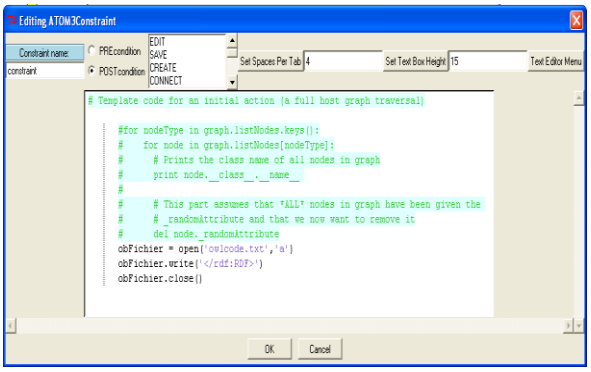


Fig. 5. End of ontology.

VI. EXAMPLE

Let us apply our approach on a simple example of development of a small OWL ontology which describes a population; it describes a group of people and their relations of parenting.

The population which we wish to describe is made up of humans, and divided into two subclasses Man and Woman. Human can have a relationship with another human. A man and a woman can be married. This small ontology can be modeled quite simply with the class diagram (fig. 6).

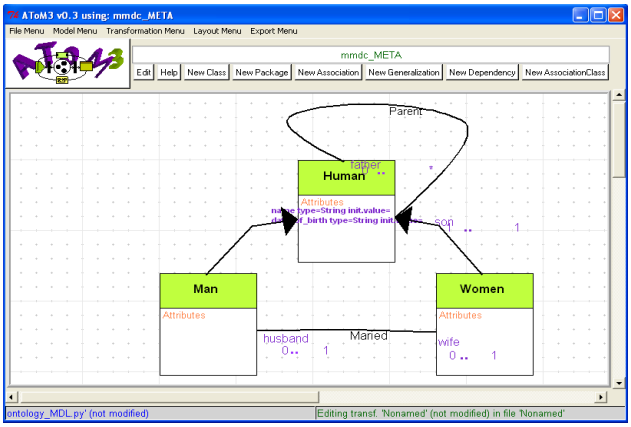


Fig. 6. Class diagram of our ontology.

We start the execution of our graph grammar; we obtain the intermediate graph (see fig. 7). In parallel, there is an automatic generation of the file which contains OWL code stored on hard disc (see fig. 9):

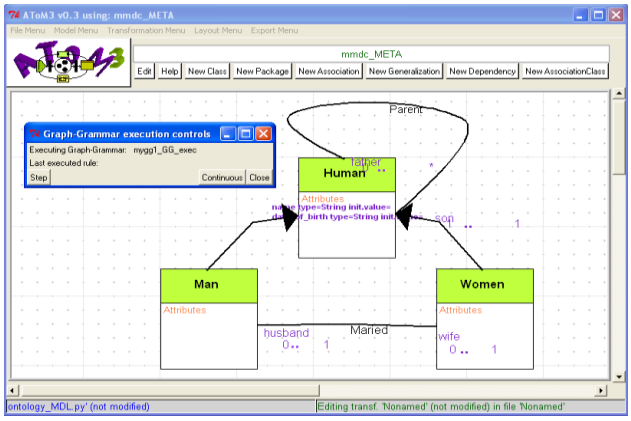


Fig. 7. Intermediate graph.

After the execution of the graph grammar on our example we obtain our ontology generated and stored. We can visualize this ontology by specialized tools such as SWOOP (fig. 8):

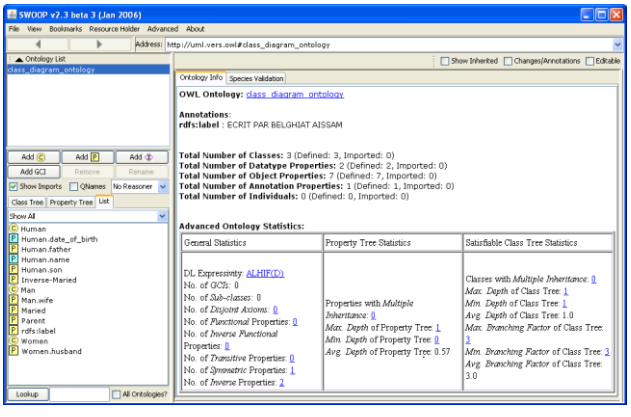


Fig. 8. Ontology Visualisation by SWOOP.

Figure 9 presents the source RDF/XML of our generated ontology.

View Source - http://uml.vers.owl#class_diagram_ontology

File Edit Update Model

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY class_diagram_ontology "http://uml.vers.owl#class_diagram_ontology">
  <!ENTITY owl "http://www.w3.org/2002/07/owl#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF xml:base="&class_diagram_ontology;"
         xmlns:owl="&owl;"
         xmlns:rdf="&rdf;"
         xmlns:rdfs="&rdfs;">

<!-- Ontology Information -->
  <owl:Ontology rdf:about="#class_diagram_ontology"
                rdfs:label="ECRIT PAR BELGHIAT AISSAM"/>

<!-- Classes -->
  <owl:Class rdf:about="#Human">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        <owl:onProperty rdf:resource="#Human.son"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0</owl:minCardinality>
        <owl:onProperty rdf:resource="#Human.father"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Man">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
        <owl:onProperty rdf:resource="#Man.wife"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0</owl:minCardinality>
        <owl:onProperty rdf:resource="#Man.wife"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#Women">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">0</owl:minCardinality>
        <owl:onProperty rdf:resource="#Women.husband"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
        <owl:onProperty rdf:resource="#Women.husband"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
<!-- Annotation Properties -->
  <owl:AnnotationProperty rdf:about="&rdfs;label"/>

<!-- Datatype Properties -->
  <owl:DatatypeProperty rdf:about="#Human.date_of_birth">
    <rdfs:domain rdf:resource="#Human"/>
    <rdfs:range rdf:resource="&xsd;string"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:about="#Human.name">
    <rdfs:domain rdf:resource="#Human"/>
    <rdfs:range rdf:resource="&xsd;string"/>
  </owl:DatatypeProperty>

<!-- Object Properties -->
  <owl:ObjectProperty rdf:about="#Human.father">
    <rdfs:domain rdf:resource="#Human"/>
    <rdfs:range rdf:resource="#Human"/>
    <rdfs:subPropertyOf rdf:resource="#Parent"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Human.son">
    <rdfs:domain rdf:resource="#Human"/>
    <rdfs:range rdf:resource="#Human"/>
    <rdfs:subPropertyOf rdf:resource="#Parent"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Inverse-Maried">
    <rdfs:domain rdf:resource="#Women"/>
    <rdfs:range rdf:resource="#Man"/>
    <owl:inverseOf rdf:resource="#Maried"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Man.wife">
    <rdfs:domain rdf:resource="#Man"/>
    <rdfs:range rdf:resource="#Women"/>
    <rdfs:subPropertyOf rdf:resource="#Maried"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Maried">
    <rdfs:domain rdf:resource="#Man"/>
    <rdfs:range rdf:resource="#Women"/>
    <owl:inverseOf rdf:resource="#Inverse-Maried"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Parent">
    <rdf:type rdf:resource="&owl;SymmetricProperty"/>
    <rdfs:domain rdf:resource="#Human"/>
    <rdfs:range rdf:resource="#Human"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#Women.husband">
    <rdfs:domain rdf:resource="#Women"/>
    <rdfs:range rdf:resource="#Man"/>
    <rdfs:subPropertyOf rdf:resource="#Inverse-Maried"/>
  </owl:ObjectProperty>
</rdf:RDF>
```

Fig. 9. Generated OWL ontology.

VII. Conclusion

We saw in this paper how to implement an application which makes a graphical environment for the automatic generation of OWL ontologies based on class diagram models and by using graph transformation in particular the powerful tool AToM3. For the realization of this application we developed a meta-model for UML class diagrams, and a graph grammar composed of several rules which enables us to transform all what is modeled visually in our AToM3 generated environment to an OWL ontology stored in a hard disk file.

In future work, we plan to extend the environment to automatically generate OWL/SWRL from UML/OCL models in AToM3.

References

[1] AToM3. Home page: http://atom3.cs.mcgill.ca.2002.

[2] Laurent AUDIBERT, “UML2”, http://www.lipn.univparis13.fr/audibert/pages/enseignement/cours.htm, 2007.

[3] Fowler, Martin, “UML Distilled - Third Edition - A Brief Guide to the Standard Object Modeling Language”, 2003.

[4] G. Karsai, A. Agrawal, “Graph Transformations in OMG’s Model-Driven Architecture”, Lecture Notes in Computer Science, Vol 3062, Springer, juillet 2004.

[5] Sebastian Leinhos, http://diplom.ooyoo.de, 2006.

[6] OMG, “Ontology Definition Metamodel”, V1.0, http://www.omg.org/spec/ODM/1.0, May 2009.

[7] OMG, “OMG Unified Modeling Language, Infrastructure,v2.3”,http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF, May 2010.

[8] Python. Home page: http://www.python.org.

[9] SIDo Group, “ATL Use Case - ODM Implementation (Bridging UML and OWL)”,http://www.eclipse.org/m2m/atl/usecases/ODMImplementation/, 2007.

[10] W3C OWL Working Group, “OWL Web Ontology Language-Overview”, http://www.w3.org/TR/2004/rec-owl-features-20040210/. Recommendation 10 Feb 2004.

[11] W3C OWL Working Group, “OWL Web Ontology Language–Guide”, http://www.w3.org/TR/2004/REC-owl-guide-20040210. Recommendation 10 February 2004.

[12] W3C OWL Working Group, “OWL Web Ontology Language-Reference”,http://www.w3.org/TR/2004/rec-owl-ref-20040210. W3C Recommendation 10 February 2004.

[13] W3C OWL Working Group, “OWL 2 Web Ontology Language Document Overview”. http://www.w3 .org/TR/2009/REC-owl2-overview-20091027.W3C Recommendation 27 October 2009.

[14] Kenneth Baclawski2 and all “Extending UML to Support Ontology Engineering for the Semantic Web”.

[15] Dragan Gašević, Dragan Djurić, Vladan Devedžić, Violeta Damjanović “Converting UML to OWL Ontologies”, 2004.

[16] Kilian Kiko, Colin Atkinson, “A Detailed Comparison of UML and OWL”,2008.