# USER-GUIDED TRANSFORMATION OF OWL ONTOLOGIES TO UML CLASS DIAGRAMS

**A THESIS SUBMITTED TO**
**THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES**
**OF**
**THE MIDDLE EAST TECHNICAL UNIVERSITY**

**BY**

**ÖZER ÖZDİKİŞ**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS**
**FOR**
**THE DEGREE OF MASTER OF SCIENCE**
**IN**
**COMPUTER ENGINEERING**

**JUNE 2007**

Approval of the Graduate School of Natural and Applied Sciences

_____

Prof.Dr. Canan ÖZGEN

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Prof.Dr. Volkan ATALAY

Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Assoc.Prof.Dr. Halit OĞUZTÜZÜN

Supervisor

Examining Committee Members

| | | |
|---|---|---|
| Prof.Dr. Adnan Yazıcı | (METU, CENG) | _____ |
| Assoc.Prof.Dr. Halit Oğuztüzün | (METU, CENG) | _____ |
| Assoc.Prof.Dr. Cem Bozşahin | (METU, CENG) | _____ |
| Assoc.Prof.Dr. Ali Doğru | (METU, CENG) | _____ |
| Dr. Semih Çetin | (Cybersoft) | _____ |

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

**Name, Surname: Özer Özdikiş**

**Signature:**

# ABSTRACT

USER-GUIDED TRANSFORMATION OF OWL ONTOLOGIES
TO UML CLASS DIAGRAMS

Özdikiş, Özer
MS, Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

June 2007, 83 pages

Using domain knowledge represented as ontologies for the design of domain specific architectures is a promising approach for better reusability. Tool support is essential for this approach to be effective. We present a flexible, user-guided transformation method to generate a framework architecture model from a domain model. The latter is in the form of an OWL ontology, and the former is in the form of a UML class diagram. We introduce a transformation tool that allows the user to configure mappings from the source OWL constructs to the target UML constructs. The mappings are defined in the metamodel level between Ontology Definition Metamodel and UML Metamodel; and then applied on the input ontology to generate the desired UML model. This user-guided approach is demonstrated on a case study that involves building a framework architecture for ontology-driven trajectory simulation development.

Keywords: Domain engineering, ontology languages, model transformations, object-oriented design methods, domain-specific architectures.

# ÖZ


OWL ONTOLOJİLERİNİN UML SINIF ÇİZENEKLERİNE KULLANICI YÖNLENDİRMESİYLE ÇEVRİLMESİ

Özdikiş, Özer
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün

Haziran 2007, 83 sayfa

Yeniden kullanılırlık açısından, alana-özgü mimarilerin tasarımında ontolojiler ile ifade edilen alan bilgisinin kullanılması gelecek vaadeden bir yaklaşımdır. Bu yaklaşımın etkin olabilmesi için yazılım araçlarının desteği önemlidir. Bu çalışmada, alan modelinden çatı mimarisi modeli üretmek için esnek ve kullanıcının yapılandırabildiği bir dönüşüm metodu sunulmuştur. Alan modeli bir OWL ontolojisi ile gösterilirken, çatı mimarisi modeli UML sınıf çizeneği şeklindedir. Sunulacak olan dönüşüm uygulaması, kaynaktaki OWL yapılarından hedefteki UML yapılarına eşleşmeler tanımlayabilmesi için kullanıcıya imkan sağlar. Eşleşmeler metamodel seviyesinde Ontoloji Metamodeli ve UML Metamodeli arasında tanımlanır; ve okunan ontoloji üzerinde uygulanarak istenen UML modeli oluşturulur. Kullanıcı yönlendirmesi içeren bu yaklaşım, yörünge

simülasyonlarının ontoloji-güdümlü olarak geliştirilmeleri için bir çatı mimarisi oluşturulmasını kapsayan bir örnek olay üzerinde gösterilmiştir.

Anahtar Kelimeler: Alan mühendisliği, Ontoloji dilleri, model çevrimleri, nesneye-dayalı tasarım metodları, alana-özgü mimariler

# ACKNOWLEDGEMENTS

I send my deepest regards and gratitude to Assoc. Prof. Halit Oğuztüzün and Umut Durak and for their expertise, insight, and guidance, and for their support during my study.

I am grateful to Serkan Kırbaş, Siyamed Şeyhmus Sınır and Celal Kavuklu for their comments, feedbacks and support.

Also, I would like to thank to my family for their limitless patient and love which motivated me to complete this study.

# TABLE OF CONTENTS

ABSTRACT ..... iv

ÖZ ..... vi

ACKNOWLEDGEMENTS ..... viii

TABLE OF CONTENTS ..... ix

LIST OF TABLES ..... xi

LIST OF FIGURES ..... xii

LIST OF ABBREVIATIONS ..... xvi

1. INTRODUCTION ..... 1

    1.1. Problem Statement ..... 1

    1.2. Roadmap ..... 3

2. BACKGROUND ..... 4

    2.1. Model-Driven Engineering ..... 4

    2.2. OMG's Meta Object Facility (MOF) ..... 5

    2.3. Eclipse Modeling Framework (EMF) ..... 6

3. CONFIGURABLE TRANSFORMATION APPROACH ..... 8

    3.1. Overview of the System ..... 8

    3.2. The OWL to UML Tool : Transformation Configuration ..... 11

        3.2.1 Available OWL constructs at the source ..... 11

3.2.2 Available UML constructs at the target ..... 24
3.2.3 Mappings from source OWL to target UML ..... 27
3.3. The OWL to UML Tool : Operational Details ..... 43
4. CASE STUDY ..... 47
5. CONCLUSION AND FUTURE WORK ..... 59
5.1. Related Work ..... 59
5.2. Contributions and Limitations ..... 61
REFERENCES ..... 64
APPENDIX A ..... 68
A. TOOL USER GUIDE ..... 68
A.1. Transformation Tool Window ..... 68
A.2. Configuration of constraints for rules ..... 74
A.3. Adding / Removing / Configuring mappings ..... 77
A.4. Running the transformation ..... 83

# LIST OF TABLES

TABLES

Table 1 OWL Class Constructs ..... 12

Table 2 OWL Property Constructs ..... 13

# LIST OF FIGURES

FIGURES

Figure 1 Relations between modeling layers ..... 7

Figure 2 Three different class-to-class mapping possibilities..... 10

Figure 3 Structure of a transformation configuration..... 11

Figure 4 Anonymous unionOf class defined as the equivalent class in [17]..... 14

Figure 5 OWL class defined as an intersectionOf two restrictions in [9] ..... 15

Figure 6 OWL class with several superclass definitions..... 16

Figure 7 LandWaterObject class defined as the complement of Ocean in [15] ... 17

Figure 8 Nearshore class defined as the intersection of OceanRegion and CoastalRegion in [15] ..... 18

Figure 9 WineColor class defined as one of White, Rose and Red in [36]. ..... 19

Figure 10 LatLonBox class including a MinCardinalityRestriction in [18]. ..... 20

Figure 11 RockGlacier class including an AllValuesFromRestriction in [15]. .... 21

Figure 12 CoastalRegion class including a SomeValuesFromRestriction in [15]. 22

Figure 13 LandSurfaceLayer class including a hasValueRestriction in [15]. ..... 22

Figure 14 Property with domain and range values in [18]. ..... 23

Figure 15 Structure of our UML Metamodel..... 25

Figure 16 UML class as a result of the given OWL class – UML class mapping configuration for the OWL class in the input ontology ..... 29

Figure 17 UML superclass relationship as a result of the given OWL class – UML class mapping configuration..... 31

Figure 18 UML class and its operations as a result of the given OWL class – UML class mapping configuration..... 32

Figure 19 UML class and its operation with parameters as a result of the given OWL class – UML class mapping configuration ..... 33

Figure 20 UML class as a result of the given OWL object property – UML class mapping configuration for the OWL object property in the input ontology. 34

Figure 21 Aggregation association between UML classes as a result of the given OWL class - UML association mapping configuration ..... 36

Figure 22 Directed association between UML classes as a result of the given OWL class - UML association mapping configuration ..... 37

Figure 23 Composition association between UML classes as a result of the given OWL class - UML association mapping configuration ..... 38

Figure 24 Composition association between UML classes as a result of the given OWL property - UML association mapping configuration..... 39

Figure 25 OWL class – UML operation mapping configuration to read the parameter names from the input OWL class and write into the operation of a UML class ..... 41

Figure 26 OWL class – UML operation mapping configuration with constraints 43

Figure 27 User Interface provided by our tool..... 44

Figure 28 Steps through the production of target UML model from the source ontology..... 46

Figure 29 A sample portion of TSONT ontology ..... 48

Figure 30 Constraint definition for the first rule..... 49

Figure 31 Mappings in the first rule..... 50

Figure 32 Mapping configuration to set the superclasses ..... 51

Figure 33 Mapping for the definition of operation names ..... 52

Figure 34 Mapping for the definition of attributes ..... 52

Figure 35 Mapping for the definition of associations ..... 53

Figure 36 Constraint definition for the second rule ..... 54

Figure 37 Mappings in the second rule ..... 55

Figure 38 Mapping for the definition of parameters of type “in” ..... 55

Figure 39 Mapping for the definition of parameters of type “out” ..... 56

Figure 40 Output UML class diagram, result of the transformation from the input OWL ontology ..... 57

Figure 41 Reuse Scenario in Case of Lynx Development (adapted from [21]) .... 58

Figure 0.1 OWL-to-UML Transformation Window ..... 69

Figure 0.2 Rule for OWL class ..... 70

Figure 0.3 Rule for OWL datatype properties ..... 71

Figure 0.4 Rule for OWL object properties ..... 72

Figure 0.5 Set input/output files ..... 73

Figure 0.6 Constraint configuration window for OWL classes ..... 74

Figure 0.7 Constraint configuration window for OWL datatype properties ..... 75

Figure 0.8 Constraint configuration window for OWL object properties ..... 76

Figure 0.9 Mapping between an OWL class and a UML class ..... 77

Figure 0.10 Mapping configuration between an OWL class and a UML class .... 78

Figure 0.11 Mapping between an OWL class and a UML association ..... 79

Figure 0.12 Mapping configuration between an OWL class and a UML association ..... 80

Figure 0.13 Mapping configuration for cardinality restrictions..... 81

Figure 0.14 Mapping between an OWL class and a UML operation..... 82

Figure 0.15 Mapping configuration between an OWL class and a UML operation ..... 83

# LIST OF ABBREVIATIONS

MDD: Model Driven Development

MDA: Model Driven Architecture

MDE: Model Driven Engineering

W3C: WWW Consortium

TSONT: Trajectory Simulation Ontology

OMG: Object Management Group

CIM: Computation Independent Model

PIM: Platform Independent Model

PSM: Platform Specific Model

ODM: Ontology Definition Metamodel

EMF: Eclipse Modeling Framework

EODM: EMF Ontology Definition Metamodel

IODT: Integrated Ontology Development Toolkit

SWEET: Semantic Web for Earth and Environmental Terminology

MTF: Model Transformation Framework

XMI: XML Metadata Interchange

XSLT: Extensible Stylesheet Language Transformations

OCL: Object Constraint Language

# CHAPTER 1

# INTRODUCTION

## 1.1. Problem Statement

Transformation of domain models, in the form of OWL ontologies, to framework architectures, in the form of UML class diagrams, is an issue that arises in the context where domain engineering and model-driven development meet. We hold the view that this issue can be resolved to the satisfaction of the user with the aid of the user-guided model transformation tools.

Domain engineering aims to support systematic reuse of domain knowledge, architectural designs, and code. It comprises three fundamental activities, namely, domain analysis, infrastructure specification, and infrastructure implementation. Domain analysis is the process of identification, acquisition, organization and evolution of reusable information on a problem domain. It results in a domain model, where a domain is understood as an area of interest for some enterprise. A domain model, however, is not amenable to operational reuse as there exists a gap between the kinds and the forms of domain knowledge captured in a domain model and the contents and forms of the software assets that can be reused in the construction of applications. Infrastructure specification aims to close this gap. It is the process of selection and organization of reusable information in the model to fit the patterns of reuse in the environment of the developer. As a result of

infrastructure specification, an architecture involving reusable software components is specified. The infrastructure implementation step produces software components [1]. The use of models to represent the outcomes of domain engineering activities, and the use of model transformations and code generation as a means to carry out these activities bring Model Driven Development (MDD), in particular, OMG's Model Driven Architecture (MDA), to bear on Domain Engineering [2], [3].

As information is gathered during domain analysis, representing the knowledge for ease of both human understanding and machine processability becomes a problem [4]. Using ontologies is proposed as a solution to this problem [5]. Using ontologies as domain models is called ontology-based domain engineering. One of the promises of ontology-based domain engineering is to derive reusable components from ontologies [6]. Ontology as a domain model, however, is the subject matter expert view of the domain. Tool-supported methods of generating the software developer view of the domain is desirable. Our present focus is on tool support for flexible transformations from a domain ontology into a framework architecture, a model that is comprehensible and extendable by the software engineer. A framework architecture, described in UML, involves class diagrams along with other kinds of diagrams, such as sequence diagrams.

As an ontology language, W3C's OWL is a well accepted semantic Web standard [7], [8]. When it comes to transforming available OWL ontologies into UML class diagrams, the "one size fits all" approach does not work. Because every domain may require a different transformation procedure depending on the context, information type, conventions, or even the personal preferences of the software engineer who specifies the infrastructure. A hard-coded transformation approach would not be flexible enough in general to be effective in practice. What we propose is a configurable transformation approach. That means, mappings from OWL ontology constructs to UML class diagram constructs will be guided by the user.

Our implementation for this approach provides a user interface to design mappings between the OWL and UML metamodel constructs. These mapping definitions are applied on the OWL ontology (domain model) and a UML class diagram (as a constituent of a framework architecture) is produced. This reconfigurable transformation tool has been employed in an industrial development effort, in which a domain ontology, called Trajectory Simulation ONTology, in short TSONT [9], is transformed to an object-oriented application framework architecture. This case study will also be discussed.

### 1.2. Roadmap

In Chapter 2, we provide the details about the background of our work. The modeling concepts and the infrastructure we used to transform our source model (i.e. ontology model) to target model (i.e. UML model) are explained.

In Chapter 3, we provide the details related to our user-guided transformation approach and the tool we developed. We describe the source OWL constructs and target UML constructs that the user can use with our tool. The possible mappings the user can define between these constructs are explained. Finally, operational details about the transformation process, like how the execution runs are provided.

In Chapter 4, we introduce a case-study. This case-study takes a trajectory simulation ontology used at TÜBİTAK-SAGE as input and produces a UML class diagram. We describe the configuration we designed to get the desired output. The results of this transformation are discussed.

Finally in Chapter 5, we provide some information about the related work in the literature. We explain what disadvantages they had, which differences our approach introduce. We discuss our contributions and the limitations. In this chapter, we also describe potential subjects for future work.

# CHAPTER 2

# BACKGROUND

## 2.1. Model-Driven Engineering

"A model represents some concrete or abstract thing of interest, with a specific purpose in mind [2]". Models provide us a better understanding of the area of interest in a more structured way. UML, a widely used modeling language for the object-oriented development standard, is a good example for the use and importance of modeling.

Software development can be divided into two phases, namely analysis phase and design phase. The analysis phase includes the domain information and the definition of features the user needs. The design phase, on the other hand, includes architectural and detailed design specifications and finally the implementation. Model Driven Engineering (MDE) is based on the idea of refinement of abstract models into more specific and more concrete models to produce the final implementation [34]. Object Management Group (OMG) developed a standard called Model Driven Architecture (MDA) to establish a formal way for this refinement process. The purpose is to separate the specification of functionality from the specification of the development platform [2].

A three level hierarchy is defined in MDA for the automation of code generation using models [2], [3]. These are Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). CIM is

a view of a system from the computation independent viewpoint. “The computation independent viewpoint focuses on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden or as yet undetermined [2]”. PIM is a view of a system from the platform independent viewpoint. PIM describes the system, but does not refer to the platform on which the system is implemented. PSM, in contrast, refines the PIM with the details that specify how that system utilizes a particular platform.

The idea is to capture domain knowledge at a conceptual level with CIM and generate the models at PIM and PSM as automated as possible, to finally get a running code. Ontologies can be regarded as CIMs [35]. They are the representation of the domain information prepared by the domain and knowledge engineers with the help of subject matter experts, without any concern about system development. Our specific aim is to contribute to the automation of transformation from a CIM, which is represented by an OWL ontology, to a PIM, which is represented by a framework architecture description involving UML class diagrams.

### 2.2. OMG’s Meta Object Facility (MOF)

For the automatic refinement of concrete models from abstract models, the models must be connected. In other words, the relationships between the models must be defined, model elements must be traceable. This is achieved by metamodels. Metamodels define sets of valid models, facilitating their transformation, serialization and exchange [34].

Our transformation method from CIM to PIM (i.e. from OWL to UML) is based on the four-layer metamodeling hierarchy of OMG’s Meta Object Facility (MOF) [10]. MOF is an extensible model driven integration framework for defining, manipulating, and integrating metadata and data in a platform independent manner. It provides a meta-metamodel at the top level, which is called M3 layer of the four-layer metamodeling hierarchy. This meta-metamodel in MOF can be

used to define more specific metamodels at M2 layer, such as the UML metamodel. We have defined an internal, simplified UML metamodel complying with the OMG's specification [11]. UML models conforming to the UML metamodel are at layer M1. Finally, the M0 layer includes the instances created from a UML model.

### 2.3. Eclipse Modeling Framework (EMF)

Similar to UML models, ontology models must also conform to a metamodel. Our ontology modeling hierarchy is based on Eclipse Modeling Framework (EMF) [12] as it provides us with the necessary packages to process ontology models in Java environment. EMF, is the Java implementation of OMG's MOF specification by Eclipse. The meta-metamodel at M3 layer in EMF is called Ecore. IBM implemented an Integrated Ontology Development Toolkit (IODT) for ontology driven development built on EMF [13]. IODT includes a library called EMF Ontology Definition Metamodel (EODM) which is an implementation of the OMG's Ontology Definition Metamodel [14]. We chose EODM as our metamodel at M2 layer for ontologies. EODM has OWL parsing, serialization, and reasoning features.

In the transformation from CIM to PIM, we have an ontology model as our source and a UML model as our target. Our approach facilitates the definition of mapping rules between the EODM and UML metamodels at M2 layer. These rules are applied on a given OWL ontology to generate a UML class diagram. The modeling layers which are used in this transformation are shown in Figure 1.

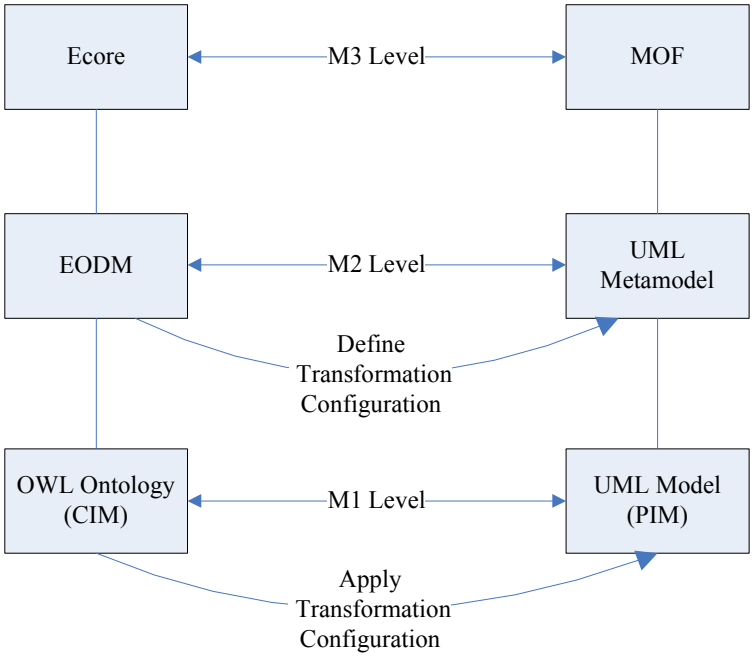



Figure 1 Relations between modeling layers

# CHAPTER 3

# CONFIGURABLE TRANSFORMATION APPROACH

The first section is the "Overview of the System" which depicts a view of the transformation process at an abstract level. It introduces the structures that we provide the user to design his transformation configuration.

In the second section "The OWL to UML Tool : Transformation Configuration", details about the transformation configuration are given. The question "which source and target constructs can be mapped in what way?" is answered. In other words, we list the source OWL constructs and target UML constructs that can be used in our tool. And we define the mapping possibilities between these source and target constructs.

In the "Operational Details" section, we explain the structure of the tool. We also give some implementation details.

The example ontologies in this chapter are earthrealm ontology [15], numerics ontology [17], space ontology [18], which are taken from the NASA's SWEET (Semantic Web for Earth and Environmental Terminology) project [16] and wine ontology [36] from W3C.

## 3.1.Overview of the System

Our user-guided approach lets the user configure the transformation he needs. The transformation configuration is composed of transformation rules. A rule is a

collection of mappings from an OWL construct to some UML constructs. There are 3 types of rules depending on the type of OWL constructs in the source, which are classes, object properties, and datatype properties. These 3 rule types restrict the source OWL constructs for the mappings defined in a rule. For example, if the user defines a rule for OWL classes, then he will be able to define mappings from OWL class constructs to UML constructs in that rule.

Following the description of the source construct, the user must specify, in terms of mappings, how to use them in the transformation. Depending on the source-target combinations, different mapping possibilities are provided to the user. The user can define 3 types of mappings in a rule: to a UML class, to an association, and to an operation parameter. A rule can include any number of mappings. A mapping is the prescription of how the UML construct should be built using the source OWL construct. The user configures these mappings by selecting the desired values provided for the transformation.

In

Figure 2, results of different mapping configurations are shown. In this example, an OWL class is transformed to a UML class. Its super class definitions are used as a source to set the attribute names, super class relationships, and operation names in the target UML model.

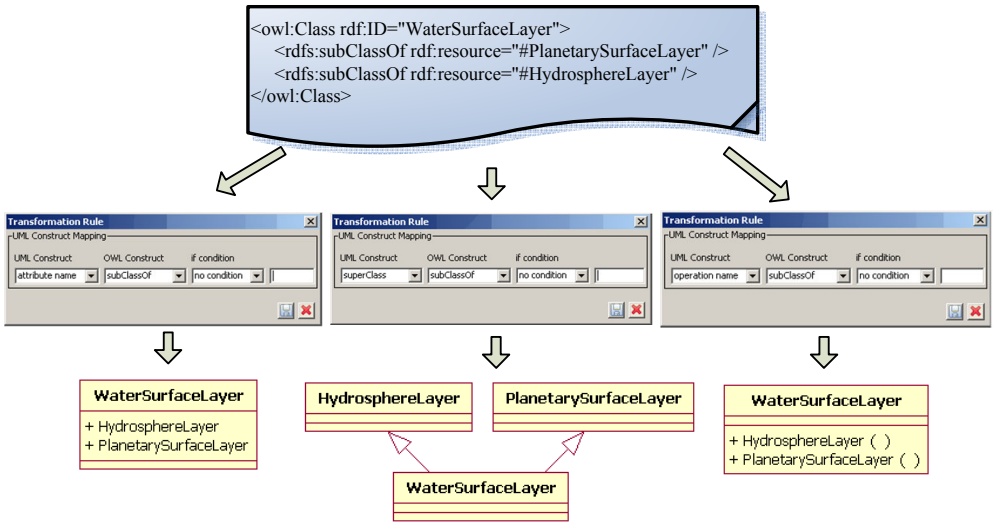



Figure 2 Three different class-to-class mapping possibilities

Constraints can be defined to restrict the entities to be evaluated in a specific rule or mapping. Constraints are actually condition-value pairs applied on an OWL construct. While evaluating an OWL construct in a rule, these condition-value pairs are used to find out if that construct is selected and should be processed. For example, the source OWL classes can be restricted as those that are the subclasses of a specific class. The structure of a complete transformation configuration is depicted in Figure 3.

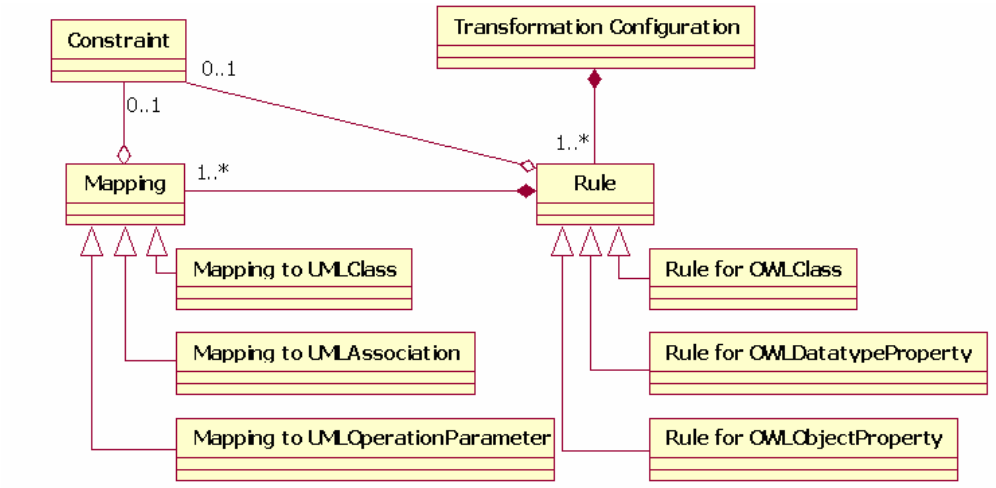



Figure 3 Structure of a transformation configuration

### 3.2.The OWL to UML Tool : Transformation Configuration

We have implemented a transformation tool that provides an interface to the user to configure the transformation. There are available OWL constructs at the source and available UML constructs at the target. The user defines his rules and mappings between these constructs and finally gets the UML model he needs.

#### 3.2.1 Available OWL constructs at the source

An OWL ontology involves classes, object properties, and datatype properties. A class has a classname and possibly super classes. A class may be defined as an intersection of, union of or complement of other classes. Further, a class may have different types of restrictions. These restrictions define the values (restrictionValue) that the class must take for a property (onPropertyName). A class may also be an enumeration class, which includes the list of individuals that are the members of the class. A property has a property name, domain and range information, and possibly super properties. Our tool lets the user use definitions of classes, object properties, and datatype properties as a source for the mappings to

UML. Extracted information from OWL class and property constructs that are available to be used in the mappings are listed in Table 1 and Table 2, respectively.

Table 1 OWL Class Constructs

| **OWL Class Construct** | | **Extracted source values to be used in mappings** |
|---|---|---|
| subclassOf | | names of the super classes of the corresponding OWL class |
| complementOf | | names of the classes which are defined as the complement of the corresponding OWL class in the ontology |
| intersectionOf | | names of the classes which are defined as the intersection of the corresponding OWL class in the ontology |
| unionOf | | names of the classes which are defined as the union of the corresponding OWL class in the ontology |
| One of | | Individuals in the "one of" list if the class is an Enumeration class |
| MinCardinalityRestriction<br>MaxCardinaltiyRestriction<br>AllValuesFromRestriction<br>SomeValuesFromRestriction<br>HasValueRestriction | onPropertyName | Restrictions on an OWL class are always defined for some OWL property. These property names can be used as a data source in transformation. |
| | restrictionValue | Restrictions on an OWL class have always a value. These restriction values can be used as a data source in transformation. |

Table 2 OWL Property Constructs

| **OWL Property Construct** | **Extracted source values to be used in mappings** |
|---|---|
| subpropertyOf | names of the super properties of the corresponding OWL property |
| Domain | names of the OWL classes defined as the domain of the property |
| Range | names of the OWL classes (for object property) or values (for datatype property) defined as the range of the property |

OWL includes a concept called "anonymous classes". Anonymous classes have no name. They are defined inside another class, and can not be used by other classes. For example the class definition in Figure 4 includes an anonymous unionOf definition. In the English language, the meaning of this OWL class definition is "ScalarOrVariableOrOperation is equivalent to a class, which is the union of Operation, Scalar and Variable". In this example, UnionOf represents an abstract class.

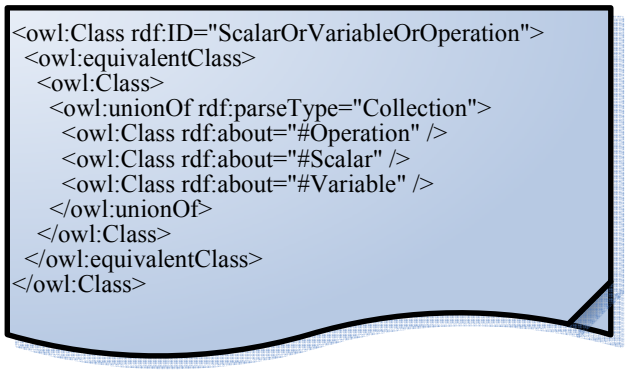



Figure 4 Anonymous unionOf class defined as the equivalent class in [17]

The class constructs (except subClassOf and oneOf) which were listed in Table 1 are used either in subClassOf or equivalentClass definitions like in Figure 4. In other words, in OWL, if a class C is to be defined as a unionOf some classes, the unionOf must be an anonymous class, and C must be either subClassOf or equivalentClass of this anonymous class. As a result of this, we read the subClassOf and equivalentClass definitions first while extracting the desired values. The desired value is then searched in these subClassOf and equivalentClass definitions. Using the same example in Figure 4, if the user wants to use the unionOf values as the data source, 1) we fetch the equivalentClass, 2) find out that the class inside the equivalentClass is a unionOf definition, 3) read the class names in the unionOf definition, namely Operation, Scalar and Variable, 4) and finally use these 3 values as the source data in the transformation.

The intersectionOf and unionOf constructs can include OWL restrictions like in Figure 5. In this example, Acceleration_Vector class is defined as an intersectionOf two restrictions of type allValuesFrom. If the user configures a mapping from the allValuesFrom restrictions in the source, during the processing of this class, all allValuesFrom restrictions inside this intersectionOf definition are evaluated.

```
<owl:Class rdf:about="#Acceleration_Vector">
 <owl:equivalentClass>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
     <owl:allValuesFrom><owl:Class rdf:ID="Acceleration_Column_Matrix"/></owl:allValuesFrom>
     <owl:onProperty><owl:FunctionalProperty rdf:about="#hasColumnMatrix"/></owl:onProperty>
    </owl:Restriction>
    <owl:Restriction>
     <owl:allValuesFrom><owl:Class rdf:ID="Coordinate_System"/></owl:allValuesFrom>
     <owl:onProperty><owl:FunctionalProperty rdf:about="#hasCoordinateSystem"/></owl:onProperty>
    </owl:Restriction>
   </owl:intersectionOf>
  </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

Figure 5 OWL class defined as an intersectionOf two restrictions in [9]

**OWL Class : subClassOf**

The subClassOf in OWL defines the subclass-superclass relationships among classes. A class may have several subClassOf definitions, i.e. it can have several superclasses. A superclass definition can be a named class, whereas it can also be an anonymous class like a restriction definition. Since anonymous classes do not have class names, we do not process them for the subClassOf definitions. In other words, only concrete named classes are taken as the source values for subClassOf definitions. In the example class in Figure 6, only GlobalPlanetaryLayers and AtmosphereLayer will be processed as the source values for subClassOf definitions of Atmosphere class.

```
<owl:Class rdf:ID="Atmosphere">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasLowerBoundaryReferenceHeight"/>
   <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0</owl:hasValue>
  </owl:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf rdf:resource="#GlobalPlanetaryLayers"/>
 <rdfs:subClassOf rdf:resource="#AtmosphereLayer"/>
</owl:Class>
```

Figure 6 OWL class with several superclass definitions

SubClassOf relationship is a transitive relationship. That means, if class C1 is subclass of C2, and C2 is subclass of C3, then C1 is also subclass of C3. While evaluating the subClassOf construct in the source, values list can be formed either in a recursive way or only the direct subclasses can be used. We provide an option to the user to choose the one he needs. He can configure whether to use only the direct subClassOf values or all subClassOf values retrieved recursively. For example, if the user configures the direct subClassOf relationship, while evaluating C1, only C2 will be processed as the subclass of C1. If the recursive retrieval is activated, both C2 and C3 will be processed.

**OWL Class : complementOf**

ComplementOf relationship in OWL is used inside a subClassOf or equivalentClass definition like in Figure 7. When extracting the value for a mapping with complementOf in the source, the tool reads the subClassOf and equivalentClass definitions and if it finds a complementOf class, takes the name of the complementOf class as the source value. In this example, the value “Ocean” will be evaluated as the source for the mapping. It is also possible that the complementOf class is not concrete but an anonymous class. That means, it does not have a class name, for example if it is a restriction definition. Since these

anonymous classes do not have class names, they can not be the data source for the complementOf mappings.

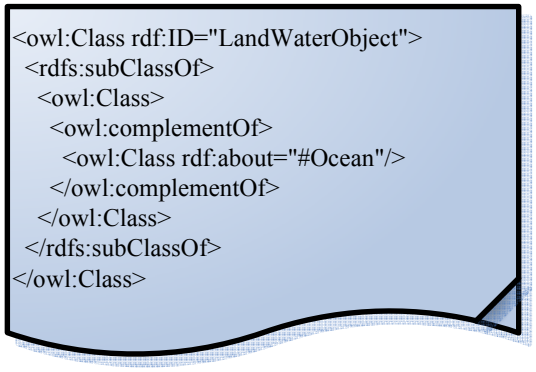



Figure 7 LandWaterObject class defined as the complement of Ocean in [15]

**OWL Class : intersectionOf/unionOf**

IntersectionOf and unionOf relationships in OWL are used inside a subClassOf or equivalentClass, similar to complementOf. In the example shown in Figure 8, the NearShore is defined as the subclass of a class, which is an intersection of OceanRegion and CoastalRegion. When extracting the values for a mapping with intersectionOf in the source, the tool reads the subClassOf and equivalentClass definitions, finds that there is an intersectionOf definition, and takes the class names inside the intersectionOf as the source values. In this example, the values OceanRegion and CoastalRegion will be evaluated as the source for the mapping.

```
<owl:Class rdf:ID="Nearshore">
 <owl:subClassOf>
  </owl:Class>
    <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#OceanRegion"/>
      <owl:Class rdf:about="#CoastalRegion"/>
    </owl:intersectionOf>
  </owl:Class>
 </owl:subClassOf>
</owl:Class>
```

Figure 8 Nearshore class defined as the intersection of OceanRegion and CoastalRegion in [15]

It is also possible that the classes in the intersectionOf and unionOf definitions are not concrete but anonymous classes (see Figure 5). Since these anonymous classes do not have class names, they can not be the data source for the intersectionOf and unionOf mappings. However, although these anonymous classes are not handled in the mappings from intersectionOf and unionOf, they can be processed in the mappings from restriction definitions. This is explained in more detail later.

**OWL Class : oneOf**

OneOf construct defines an enumeration of the members of a class. It describes and limits the possible values for an instance of this class. In Figure 9, the WineColor is an enumeration of White, Rose, and Red. That means, WineColor can be one of these 3 values. For example, WhiteWine can be described as Wine with color White.

```
<owl:Class rdf:ID="WineColor">
  <owl:oneOf rdf:parseType="Collection">
    <owl:Thing rdf:about="#White"/>
    <owl:Thing rdf:about="#Rose"/>
    <owl:Thing rdf:about="#Red"/>
  </owl:oneOf>
</owl:Class>
```

Figure 9 WineColor class defined as one of White, Rose and Red in [36].

The values inside this enumeration can be used as source values for the transformation. If the user configures a mapping for oneOf, while processing the WineColor class, Wine, Rose and Red will be used as source values in the transformation.

**OWL Class : min/maxCardinalityRestriction**

A min/maxCardinalityRestriction defines the minimum/maximum number of OWL individuals that an instance of that class must be related with for a particular property. In the example in Figure 10, the minCardinalityRestriction enforces that an instance of LatLonBox must have at least 1 value for the hasLatitudeInterval property.

```
<owl:Class rdf:ID="LatLonBox">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasLatitudeInterval"/>
      <owl:minCardinality>1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 10 LatLonBox class including a MinCardinalityRestriction in [18].

If the user configures a mapping using minCardinality, during the evaluation of LatLonBox class at the transformation, hasLatituteInterval and 1 will be available in the source as minCardinalityRestriction.onProperty and minCardinalityRestriction.value respectively. If a class is defined to be the intersection of or union of several min/maxCardinalityRestrictions (like in Figure 5), all these restriction definitions will be available as source values.

**OWL Class : allValuesFromRestriction**

This restriction limits the possible values that an OWL class can be related with for a specific property. For the onProperty in the restriction, the restricted class can have the property values only from the class identified in the restriction definition. For the example in Figure 11, the instances of RockGlacier must have values from Rock_body on the property contains.

```
<owl:Class rdf:ID="RockGlacier">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:ID="#contains"/>
   <owl:allValuesFrom rdf:resource="#Rock_body"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Figure 11 RockGlacier class including an AllValuesFromRestriction in [15].

If the user configures a mapping for allValuesFromRestriction.onProperty, contains will be processed as the source value while processing the RockGlacier class. If the user configures the mapping for allValuesFromRestriction.value, the Rock_body will be used as a source value.

For the case in Figure 5, the values in both restriction definitions will be used as the source values if the user configures a mapping using allValuesFromRestriction.

The value of the restriction can also be an enumeration class, i.e. anonymous oneOf definition. In that case, if the user wants to use the restriction values as the data source for transformation, all individual names in the enumeration are used as source values.

**OWL Class : someValuesFromRestriction**

This restriction states that the restricted class must have some values on a specific property. Not all values have to be instances of a specific class, like in allValueFromRestriction. Example OWL class in Figure 12 has such a restriction. According to this definition, instances of CoastalRegion must have some instances of LansRegion for the property hasCommom2DBorderWith.

```
<owl:Class rdf:ID="CoastalRegion">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:ID="#hasCommon2DBorderWith"/>
   <owl:someValuesFrom rdf:resource="#LandRegion"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Figure 12 CoastalRegion class including a SomeValuesFromRestriction in [15].

If the user configures a mapping using this restriction, the name of the property or the value of the restriction can be used as the source values. Similar to allValuesFromRestrictions, the value of the restriction can be an enumeration. In this case, the individuals can be used as source values.

**OWL Class : hasValueRestriction**

HasValueRestriction forces a class to have a specific value on a property. In the example in Figure 13, the instances of LandSurfaceLayer class must have the value 0 for the property hasUpperBoundaryReferenceHeight.

```
<owl:Class rdf:ID="LandSurfaceLayer">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#hasUpperBoundaryReferenceHeight"/>
   <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0</owl:hasValue>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Figure 13 LandSurfaceLayer class including a hasValueRestriction in [15].

The name of the onProperty and the restriction value can be used as the data source for the transformation. The value of the restriction must be a typed literal, which can be a string, integer, double, float, or boolean.

**OWL Property : subPropertyOf**

Similar to the subClassOf relationship among classes, properties can have the subPropertyOf relationships. If a property is a subproperty of another property, it inherits the domain and range information of its superproperty. The names of the properties defined in the subPropertyOf constructs can be used as the source values in the transformation. The subPropertyOf values can also be found in a recursive way just like for classes.

**OWL Property : domain/range**

Properties bind the domain classes with the classes in the range. The domain and range classes defined for the properties can be used in the transformation as source. For example, the hasCapital property in Figure 14 has AdministrativeRegion as its domain class, and City as its range class. That means, the instances of AdministrativeRegion class can have the instances of City on property hasCapital. If the user designs a mapping for domain or range, these values (i.e. AdministrativeRegion or City) are used as source values in the transformation.

```
<owl:ObjectProperty rdf:ID="hasCapital">
  <rdfs:domain rdf:resource="#AdministrativeRegion"/>
  <rdfs:range rdf:resource="#City"/>
</owl:ObjectProperty>
```

Figure 14 Property with domain and range values in [18].

### 3.2.2 Available UML constructs at the target

A UML model can include the definitions for the UML classes, generalizations, associations, attributes, operations, and parameters for operations. A class has a name and possibly some attributes and operations, which are identified by their names within a class. An operation can have parameters of type “in”, “out”, “in-out”, and “return”. There can be generalizations and associations defined between classes. Generalizations represent the subclass-superclass relationships. Associations can be of type aggregation, composition, or directed association, possibly with multiplicity information attached at either end. One restriction in this metamodel definition is that the class names in the model and operation names in a class must be unique. This is because the OWL classes in the ontology are also identified with their names.

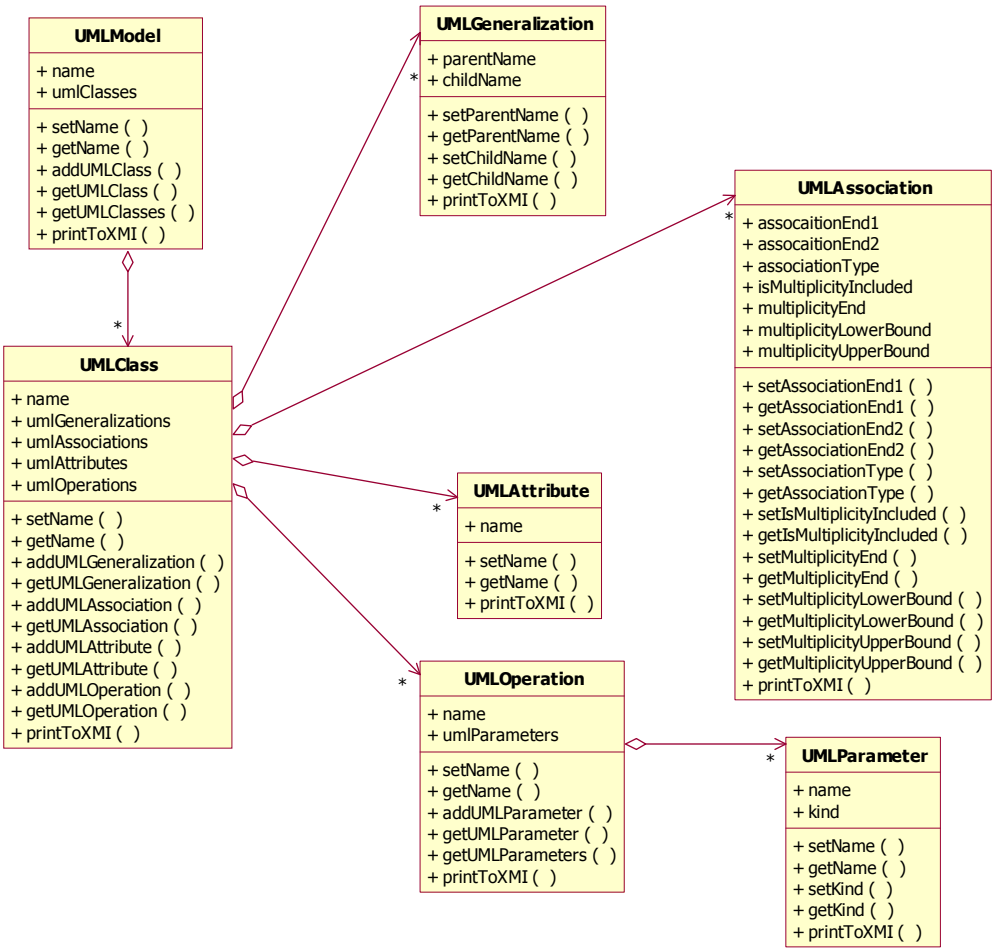



Figure 15 Structure of our UML Metamodel

Below we list the classes included in our UML metamodel.

**UMLModel**

UMLModel represents the target model we produce as a result of the transformation process. It is at the top of the UML modeling structures. It encapsulates a list of UMLClass instances. The package name for the target class diagram is always taken from the name of the ontology. Since we do not provide

any user-guidance to set the package name, we did not create any separate object for packages.

**UMLClass**

The objects of this type represent the classes in the UML class diagram. A UMLClass has an attribute called “name”, which is actually the class name in the UML class diagram. A UMLClass also includes a list of UMLGeneralizations, UMLAssociations, UMLAttributes and UMLOperations. The classname is the key to find a specific class in the UMLModel. That means, two classes with the same name can not exist in the model. During the transformation, if a mapping tries to add a class with the same name, it is not allowed; instead the existing class is used for this mapping. We do not provide the user to set the visibility of the classes, instead we create them all public.

**UMLGeneralization**

UMLGeneralization represents the subclass-superclass relationships between UML classes. Generalizations are identified by the names of the subclass and superclass that the generalization relationship is bound to. Between two classes, there can be at most one generalization.

**UMLAssociation**

A UMLAssociation represents the association between two classes in the UML class diagram. It has two association ends, representing the classes which are bound through the association. ClassNames in UMLClass objects are used as identifiers for the association ends. UMLAssociation can be of 3 different types, which are Directed, Composition and Aggregation. The type is represented by the associationType variable in UMLAssociation. Association can also include multiplicity information. Whether an association includes multiplicity is kept in isMultiplicityIncluded. The details about the multiplicity (i.e. the lower and upper bounds, at which end the multiplicity should be displayed) are also kept in UMLAssociation. A UMLAssociation is identified with the names of the

UMLClass instances at both ends. So, if a mapping tries to add a UMLAssociation, it is checked first if the association already exists. If exists, the modification configured in the mapping is done on the existing UMLAssociation. For example, one mapping can add associations, one other mapping can set the multiplicities for them.

**UMLAttribute**

A UMLAttribute represents an attribute of a class in the UML class diagram. A UMLAttribute is identified with its name in a UMLClass. And a UMLClass can not have two UMLAttributes with same name. We do not provide the user to set the visibility of attributes, instead we create them all public.

**UMLOperation**

UMLClasses can also include instances of UMLOperations. A UMLOperation has a name, which is actually the name of the operation in the UML class. A UMLOperation, just like UMLAttribute, is identified with its name. So, a UMLClass can not have two UMLOperations with the same name. UMLOperation can include a list of UMLParameter objects. We do not provide the user to set the visibility of operations, instead we create them all public.

**UMLParameter**

The instances of UMLParameters represent the parameters of UMLOperations. A UMLParameter has a name, which represents the parameter name of operation, and a type, which specifies whether the parameter is “in”, “out”, “in-out” or “return”. A parameter is identified with its name.

### 3.2.3 Mappings from source OWL to target UML

Our tool provides an interface to the user to define mappings between the available OWL constructs at the source and the UML constructs at the target, which were defined above. Mappings can be classified into 3 types regarding the target constructs. These are

- Mappings to UML classes
- Mappings to UML associations
- Mappings to UML operation parameters

During the transformation process, mappings to UML classes are resolved first. After the classes are ready in the target UML model, mappings to associations, and finally, operation parameters are evaluated. The name of the ontology is carried over to the target model as the package name.

**Mappings to UML classes**

This mapping type is used to introduce new classes into the UML model, with their super class relationships, attributes, and operation names. Depending on the type of the rule, OWL classes or OWL properties in the input ontology are traversed considering the constraint definitions. For each selected OWL class/property, a new UML class is created in the UML model with the same OWL class/property name. If a class with the same name already exists in the UML model, it is not created again since the class names are supposed to be unique in a UML model according to our metamodel definition. In addition to the new UML class definitions, user can define further mappings between the OWL and UML class constructs. The OWL constructs listed in Table 1 and Table 2 can be used to define the super classes, attributes, or operation names of the corresponding UML class.

We use the earth-realm ontology for illustration [15]. In this ontology, there is a class named Nearshore which is defined as the intersection of OceanRegion and CoastalRegion. Assume that the user adds a mapping from OWL class to UML class in a rule. And in the mapping configuration, he maps the "intersectionOf" in OWL to "attribute name" in UML as shown in Figure 16. During the transformation process, a UML class with name *Nearshore* is created in the model. Since there is also a mapping from "intersectionOf" to "attribute name",

attributes named *OceanRegion* and *CoastalRegion* are added to the UML class *Nearshore*.

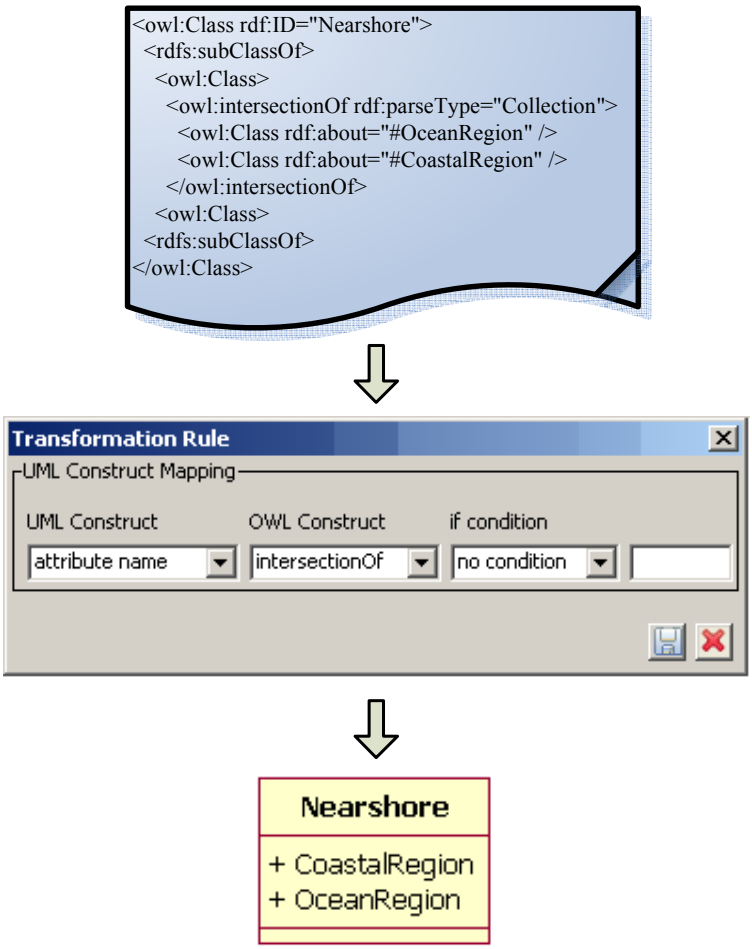



Figure 16 UML class as a result of the given OWL class – UML class mapping configuration for the OWL class in the input ontology

In addition to constraints per rule, the user can define constraints also for these single mappings. For example, in the mapping above, user could define a

constraint as “starts with = ocean”, and this would add just *OceanRegion* as an attribute to the class *Nearshore*; *CoastalRegion* would be filtered out.

Another example for mappings to UML classes can be given to set the superclass relationships in UML. In the earth-realm ontology [15], there is a class called SeaFloorWaterLayer with an allValuesFrom restriction. The restriction is defined on the property isAdjacentTo and the value of this restriction is SeaFloor. If the user defines a mapping as in Figure 17, the value of the restriction will be used to set the superclass in UML. During the transformation process, the UML class *SeaFloorWaterLayer* will be created (if it already exists in the target UML model because of a previously executed mapping, it will not be created again. That previously created UML class will be used). According to the mapping, the value of allValuesFrom restriction must be used to define the superclass of *SeaFloorWaterLayer*. So the UML class called *SeaFloor* will be searched in the target model and if it is found, a superclass relationship is added between *SeaFloorWaterLayer* and *SeaFloor*.

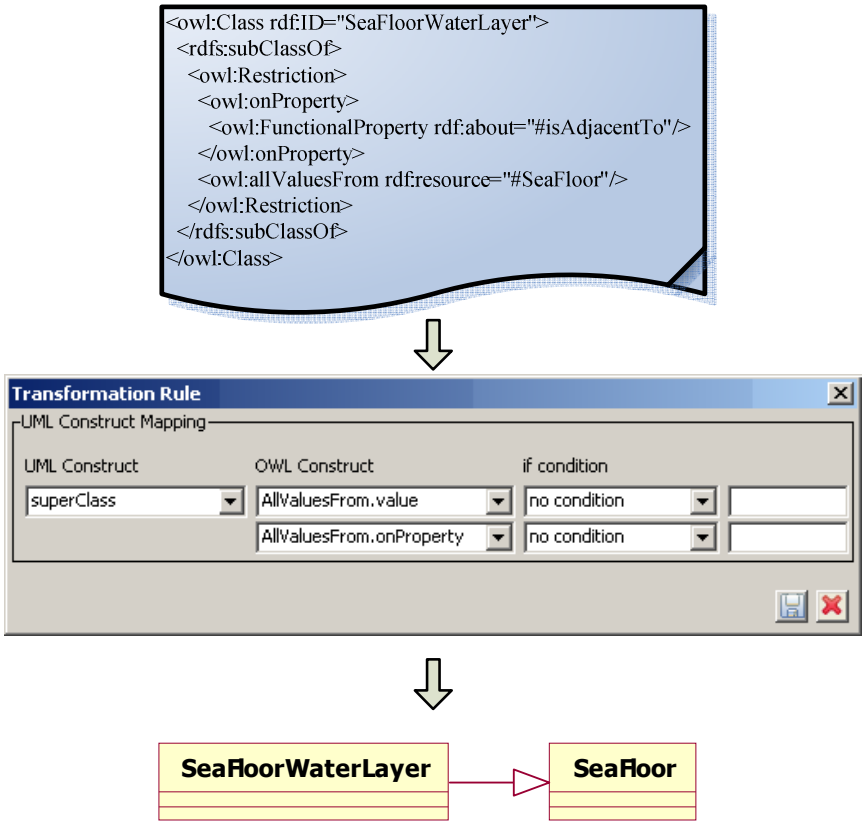



Figure 17 UML superclass relationship as a result of the given OWL class – UML class mapping configuration

In Figure 18, an example for setting the operation names in target UML class is given. In this example, operation names are taken form the subClassOf definitions in the source classes. Atmosphere class in earth-realm ontology [15] has 3 subclasses, one of which is an anonymous hasValue restriction. As explained in (3.2.1 Available OWL constructs at the source), the anonymous classes in subClassOf definitions are not used in mappings from subClassOf, but only the names of the concrete classes are used. During the transformation process, the UML class *Atmosphere* will be created (or if it already exists, previously created

one will be used). According to the mapping, *GlobalPlanetaryLayers* and *AtmosphereLayer* will be added as operation names to the *Atmosphere* class.

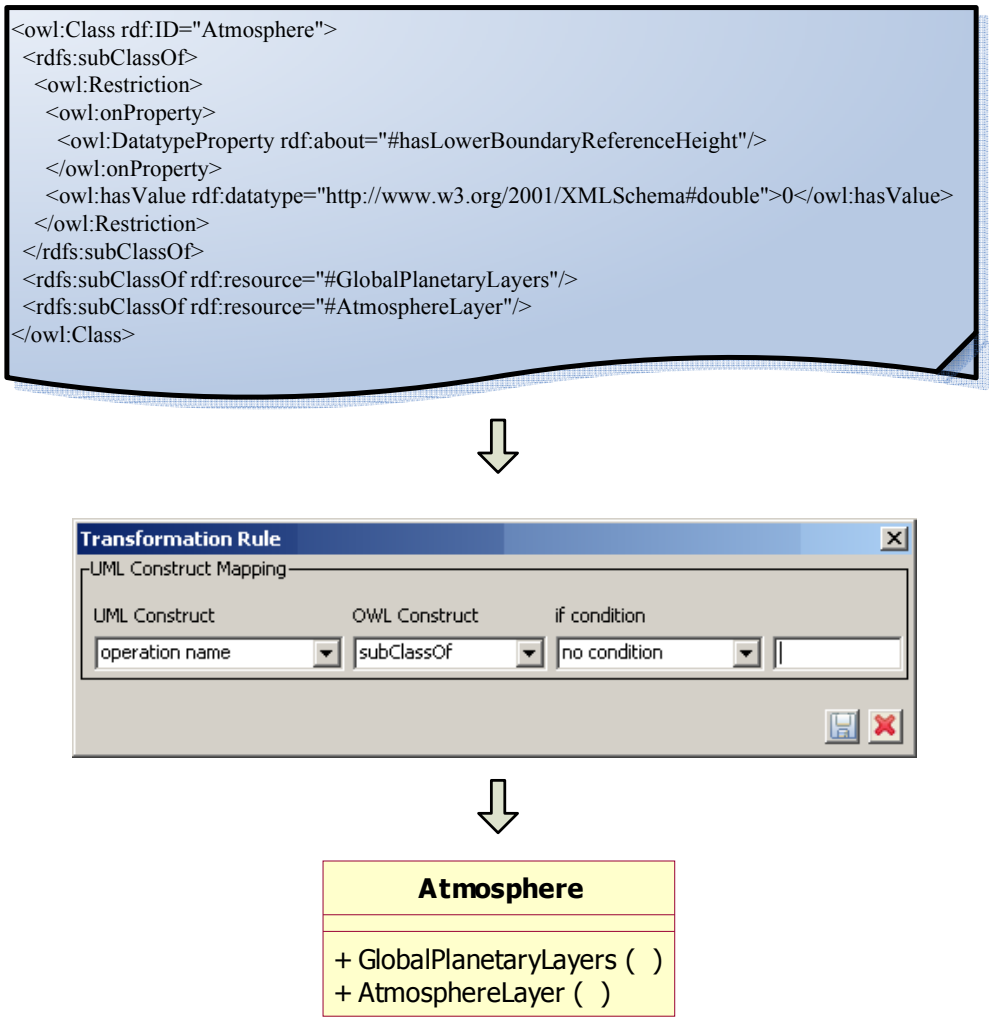



Figure 18 UML class and its operations as a result of the given OWL class – UML class mapping configuration

There is a special feature if the operation names are to be taken from a restriction. That feature provides the possibility to take the operation name from the onProperty definition, and the parameter name from the restriction value (and vice versa). In the example in Figure 20, operation name is taken from the onProperty name of the restriction (i.e. isAdjacentTo) and the parameter of this operation is taken from the restriction value (i.e. SeaFloor).

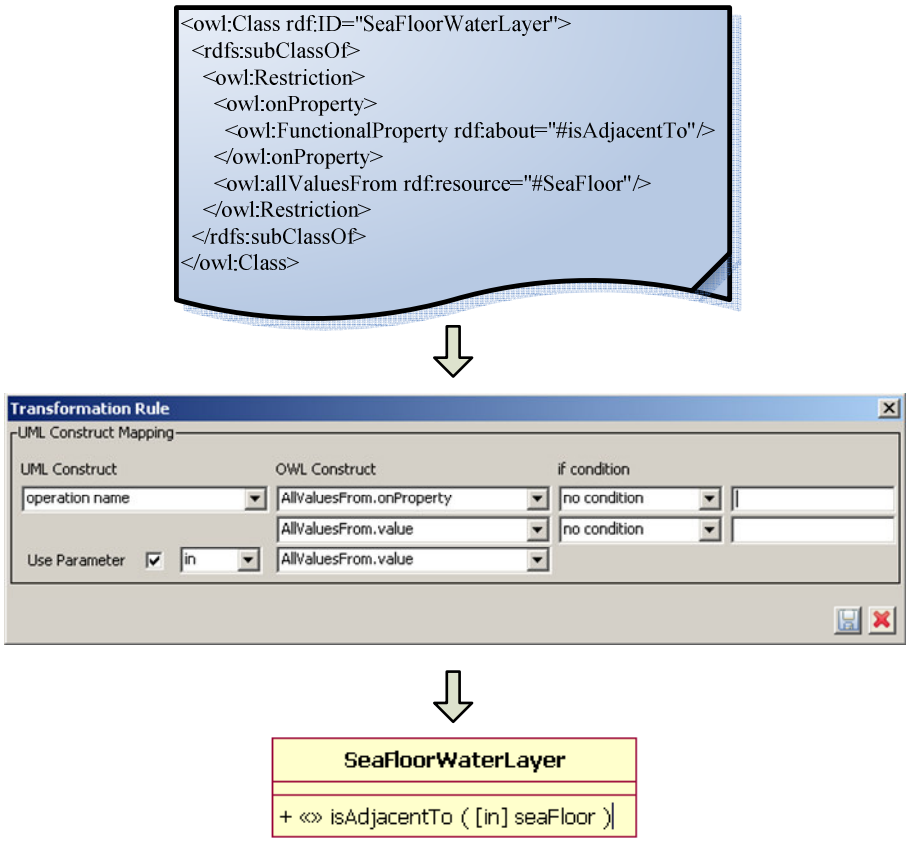



Figure 19 UML class and its operation with parameters as a result of the given OWL class – UML class mapping configuration

The information about the UML class definitions can be read from OWL properties too. The only difference from the class-to-class mapping is that the properties and their constructs in Table 2 are used as class names, super class names, attributes, and operations in the UML model. In Figure 20, there is an example using the object property named hasUnit in the numerics ontology [17]. If the user configures the mapping from "domain" to "operation name", the value defined as the domain of the object property is added as an operation to the UML class.

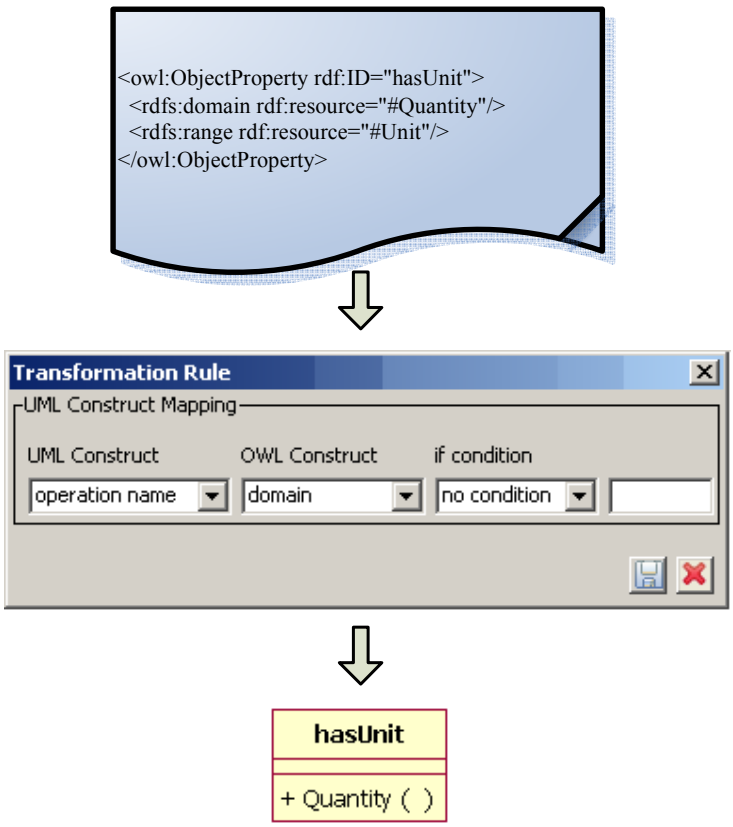



Figure 20 UML class as a result of the given OWL object property – UML class mapping configuration for the OWL object property in the input ontology

**Mappings to UML associations**

This mapping provides the opportunity to define associations between the UML classes in the target UML model. OWL classes/properties in the input ontology are traversed according to the constraint defined for the rule. The name of the currently traversed OWL class/property is used to identify the first end point in the UML association. Depending on the rule type and the mapping configuration, OWL constructs in Table 1 and Table 2 are used to find the second end point in the association. In the mapping definition, the user selects the association type to be used, where the choices are aggregation, composition, and directed association.

In the mapping, in addition to the association type, it is also possible to define constraints for each unique association definition. A special case for the min/max cardinality restrictions is that the user can define multiplicity of the association. The min/max cardinality restrictions have integer values. If the second association end is to be taken from the onPropertyName of the restriction, value of the restriction can be attached to the association as a multiplicity. This value can be used as exact, lower bound or upper bound multiplicity. End1 or end2 can also be selected for the desired placement of the multiplicity.

In the space ontology [18], there is a class named `LatLonBox` which has a minCardinalityRestriction on a property `hasLatitudeInterval`, with value `1`. Let the user create a mapping from OWL class to UML association and configure it as "UML aggregation association = OWL MinCardinality.onProperty". Also he makes the settings for the multiplicity in the association as shown in Figure 21. Through the execution of the transformation, the tool will try to add an association from UML class *LatLonBox* to UML class *hasLatitudeInterval*, and it will set the multiplicity as "*1..**" at the first end of the association. It is a prerequisite that both association ends exist as classes in the UML model. That means, if UML class *LatLonBox* or *hasLatitudeInterval* do not exist in the target model, no

association can be set between them. This is why this mapping type is processed after all “Mappings to UML classes” are processed.

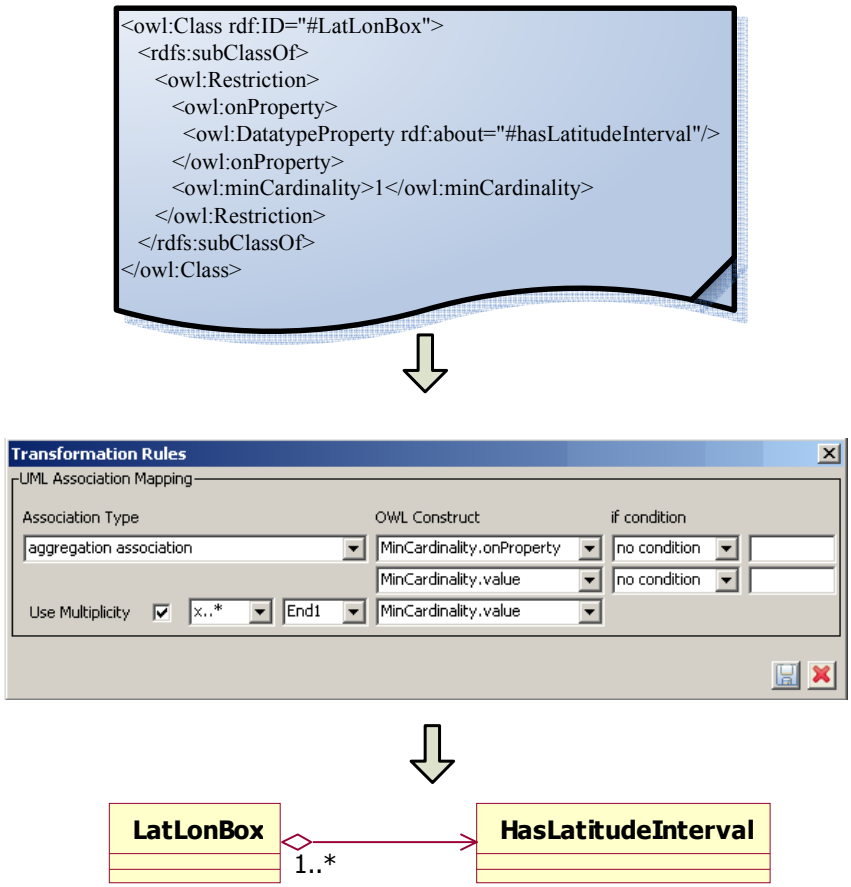



Figure 21 Aggregation association between UML classes as a result of the given OWL class - UML association mapping configuration

As an example for the directed association, we use the OceanCrustLayer class in the earth-realm ontology [15]. OceanCrustLayer has two subClassOf definitions. In the mapping configuration, the user sets the directed association to be taken from the subClassOf definitions. So, during the transformation, while

OceanCrustLayer is being processed, the names of the subClassOf classes are read. The tool tries to add directed associations between *OceanCrustLayer* and two subClassOf classes, i.e. *CrustLayer* and *OceanLayer*. To be able to insert an association between two classes, the both ends of the association must exist. Because of this, the UML classes *OceanCrustLayer*, *CrustLayer*, and *OceanLayer* must exist in the target UML model.

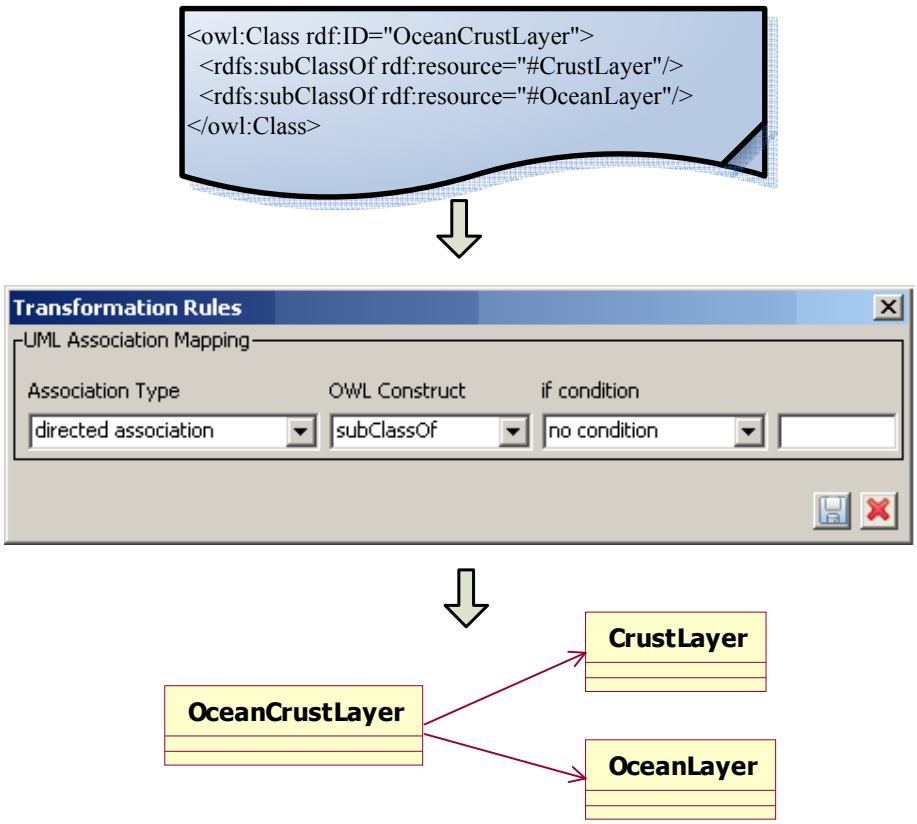



Figure 22 Directed association between UML classes as a result of the given OWL class - UML association mapping configuration

The same OWL class in the above example can be used to create composition association in the target model. In this case, assume that the user entered an if condition to the mapping, for the condition "begins with" and value "Ocean". As a result of this condition, only the association from *OceanCrustLayer* to *OceanLayer* is added. The association from *OceanCrustLayer* to *CrustLayer* is not added since *CrustLayer* does not satisfy the condition. Figure 23 shows the mapping for composition association with mapping constraint and the result.

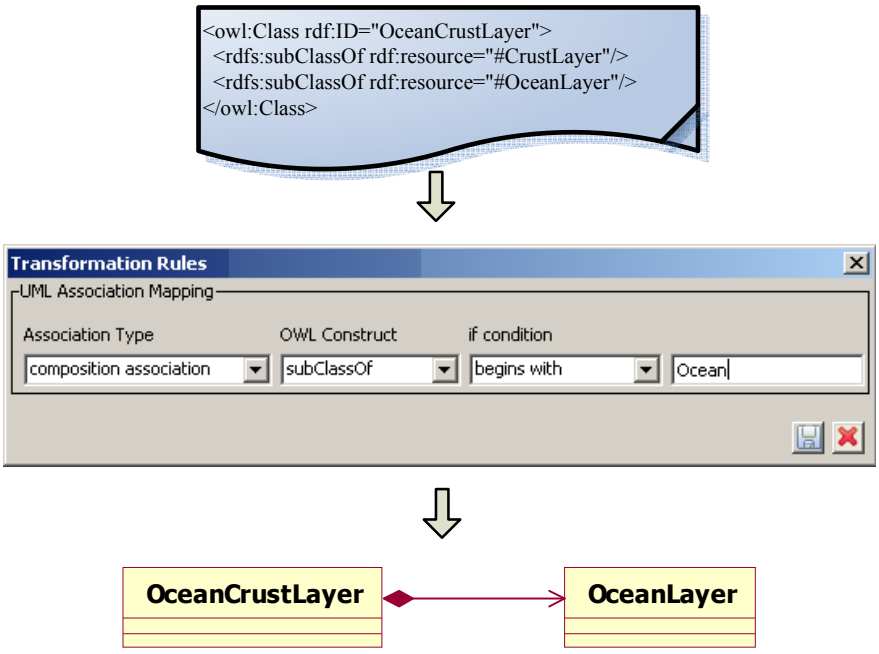



Figure 23 Composition association between UML classes as a result of the given OWL class - UML association mapping configuration

Associations can be taken from the OWL Properties too. In the space ontology [18], there is an ObjectProperty hasDirection, which has the range value Direction. If the user defines a mapping as in Figure 24, the result will be an

association between *hasDirection* and *Direction* classes in target UML model. We assume that *hasDirection* and *Direction* classes were created in the UML model beforehand with some mappings to UML classes.

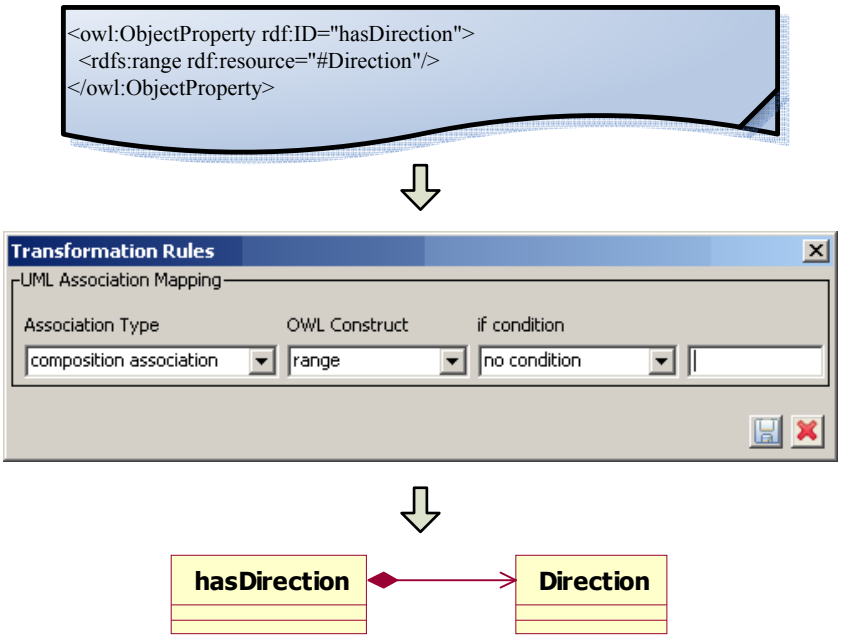



Figure 24 Composition association between UML classes as a result of the given OWL property - UML association mapping configuration

**Mappings to UML operation parameters**

There may be cases where some class in the ontology defines an operation with parameters. This mapping is used to fill the parameter names of operations in UML classes. Parameter names can be read from the OWL class or property constructs in Table 1 or Table 2, respectively. The transformation for this type of mapping works as follows: OWL classes/properties in the input ontology are traversed one by one according to the constraint definitions for the rule. For each

selected OWL class/property named C, an operation with name *C* is searched in UML classes of the target model. For each operation named *C*, the OWL construct defined in the mapping is used to feed the parameter names to this operation. This mapping type is evaluated as the last mapping type, so that we already have the operation names available in the UML classes of the result model.

For example, in the numerics ontology [17], there is a class named Quantity which is defined as the union of Vector, Scalar, and Tensor. Assume that the user adds a mapping like the one in Figure 20, so we will have the class *hasUnit* with an operation named *Quantity* in the target UML model. Let the user design a mapping from OWL class to UML operation parameter for the OWL construct "unionOf" as shown in Figure 25. At transformation time, when OWL class Quantity is processed, all UML classes with an operation named *Quantity* are searched for, and *hasUnit* will be found. This mapping will add parameters named *Vector*, *Scalar*, and *Tensor* to the *Quantity* operation of the *hasUnit* class.

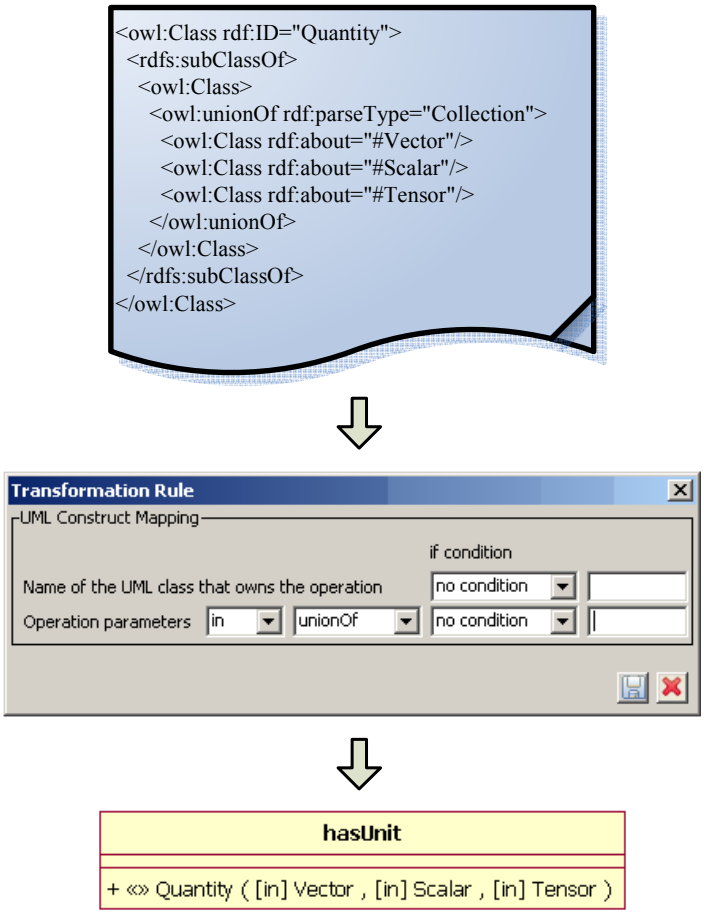



Figure 25 OWL class – UML operation mapping configuration to read the parameter names from the input OWL class and write into the operation of a UML class

The precondition for this mapping is that the correct operations are defined in desired UML classes beforehand. This is achieved by the operation name mappings in UML class transformations (see Mappings to UML Classes). So, mapping to UML operation parameters is just a matter of feeding the parameter names to the previously defined operations of previously defined classes.

The name of the owner UML class can be bound to a constraint in the mapping. In the example scenario, with a constraint defined for the owner class name, hasUnit could have been filtered out. So the parameter settings for *Quantity* operation in this UML class would have been ignored. It is also possible to define conditions on each individual parameter name. Parameter types can be defined as one of the four specified in the standard [11], namely "in", "out", "in-out", or "return". Using the constraint definitions on parameter names, user can add different types of parameters to an operation in different mapping configurations.

As a second example, we will try to feed the parameter names to the *AtmosphereLayer* operation of *Atmosphere* class, which was created in Figure 18. In Figure 26, the result of the mapping configuration for AtmosphereLayer is shown. In this configuration, the user states that the parameter names must be taken from the values of allValuesFrom restrictions, if the restriction is defined on a property whose name includes "Substance". The type of the parameter must be "inout". Moreover, parameters will be added if the name of the UML class which owns the operation equals "Atmosphere". During the processing of AtmosphereLayer class in this mapping, Air will be taken as the selected restriction value since it is defined on the property primarySubstance. The tool will search for the operations called *AtmosphereLayer* in target UML classes, and if it finds a UML class whose name equals "Atmosphere", it will added the parameter *Air* in its operation.

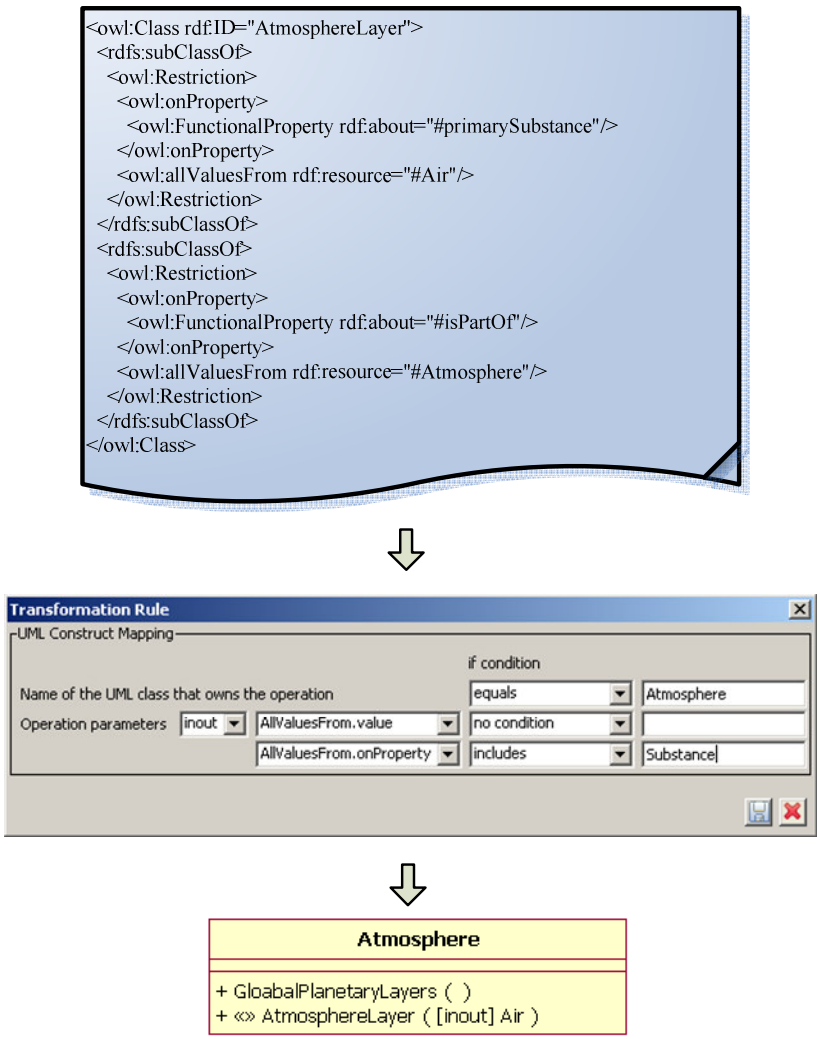



Figure 26 OWL class – UML operation mapping configuration with constraints

Similar to OWL Classes, OWL Properties can also be the source for parameter names.

### 3.3.The OWL to UML Tool : Operational Details

For the design of mappings between the OWL and UML constructs, we provided graph panels to the user as shown in Figure 27. Each graph corresponds to a rule.

On these graphs, the graph cells that represent the OWL and UML constructs are tied to each other, which correspond to mappings in a rule. Each graph includes exactly one graph cell for the OWL construct and one or more graph cells for the UML constructs. We used JGraph for the graphing infrastructure [37]. JGraph provides us the necessary packages to draw graph components like graph cells and edges between cells.

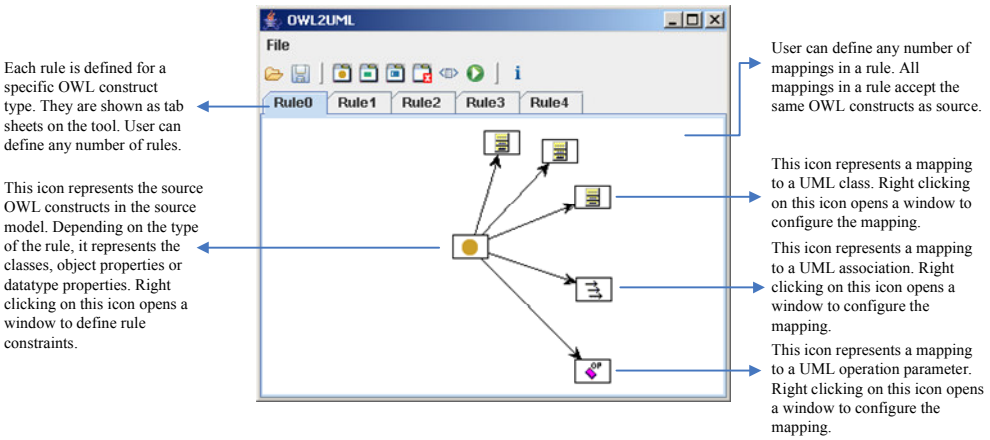



Figure 27 User Interface provided by our tool

The graph cell for the OWL construct represents the OWL classes or properties in the source, depending on the rule type. This graph cell enables the user to define constraints on the source OWL constructs.

The type of the UML graph cell defines the target UML construct to which we want to transform the constructs in the source. There are 3 types of UML graph cells corresponding to the 3 different mapping types. What is visualized within the graph is an abstract view of the mapping. User sets the details of the mappings by right clicking on the graph cells and making configurations on separate windows, which were given in Figures in (3.2.3 Mappings from source OWL to target

UML). The whole transformation configuration can be exported to files so that they can be imported and used later again.

The possible condition clauses in constraint definitions are “regular expression”, “equals”, “begins with”, “ends with”, “includes” and negations thereof. The “regular expression” choice is used to check the validity of a construct complying with a regular expression. Regular expressions must be written according to the regular-expression format of the Pattern class in regex library of Java [19].

Rules, mappings, and constraints can be defined in any order. Once they are defined, user can start the transformation process. Transformation begins with the parsing of the input OWL file using the OWLParser provided in EODM libraries. So we have an OWLOntology instance to process and a UML model instance to fill at the beginning. The tool traverses all the graphs (rule definitions), identifies the selected OWL classes/properties according to the constraints and adds UML classes to the UML model. After this step, all UML classes will be available in the UML model to define their operations, attributes, superclass relationships, and associations. This is achieved by a second traversal of the mapping graphs. This time, the execution is delegated to the specialized mapping processors depending on the mapping type. These mapping processors can set the attributes, operations, relationships of the UML classes, and parameters of the UML operations. After all the mappings are processed, all UML structures in the target model are serialized into an XMI file [20]. This XMI file can be imported and interpreted by any standard-compliant UML tool. Steps through the transformation process are shown in Figure 28.

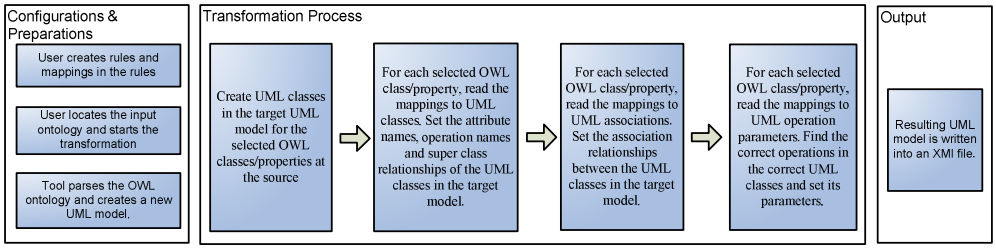



Figure 28 Steps through the production of target UML model from the source ontology

# CHAPTER 5

# CASE STUDY

The tool is applied to build an ontology-based trajectory simulation framework [21]. The trajectory simulation ontology called TSONT [9] serves as a domain model for the trajectory simulation development projects at TUBITAK-SAGE [22]. TSONT has been prepared with Protégé, a popular, free, and open source ontology editor from Standford University [23].

A development methodology is formulated to reuse this ontology-based on Model Driven Engineering. In the reuse scenario, the domain model (i.e. TSONT) is transformed to the Platform Independent Framework Architecture as a step of the abstract design process. This transformation is accomplished with our transformation tool presented in this study. The same transformation was previously carried out by hand.

There are 6 main classes at the top of the trajectory simulation ontology [9]. The most important for the transformation is the Trajectory_Simulation_Class, because each of its subclasses must be represented as a UML class in the resulting model. Operation and attribute names are basically taken from the OWL restriction definitions for each class. Subclasses of Trajectory_Simulation_Function provide the operation parameter information. The hierarchy including some selected example classes in TSONT is shown in Figure 29.

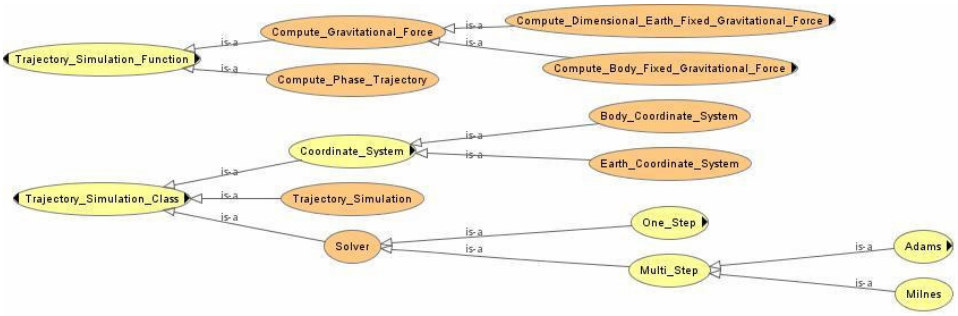



Figure 29 A sample portion of TSONT ontology

We define two rules to fulfill the transformation requirements for the trajectory simulation ontology. In the first rule, we define mappings for UML classes and UML associations using the subclasses of Trajectory_Simulation_Class as our source. The second rule is used for mappings to UML operation parameters, in other words, for setting the parameter names in the target UML operations. Since this information is taken from the subclasses of Trajectory_Simulation_Function, we had to define a separate rule to have a separate set of source OWL classes. The first rule can be described as "the rule for classes and associations", whereas second rule as "the rule for operation parameters". Both rules will read the information only from OWL classes, not from the OWL properties. So these two rules are created for type "OWL class".

***Rule-1: Rule for classes and associations:***

The mappings defined in this rule must be applicable only for a restricted set of OWL classes. The criterion to find the source OWL classes is the subClassOf relationship with the Trajectory_Simulation_Class. So we define a constraint in this rule for subclasses, with condition "equals" and constraint value "Trajectory_Simulation_Class" as shown in Figure 30. As a result, the source

OWL classes are restricted to be the subclasses of Trajectory_Simulation_Class for all mappings in this rule.

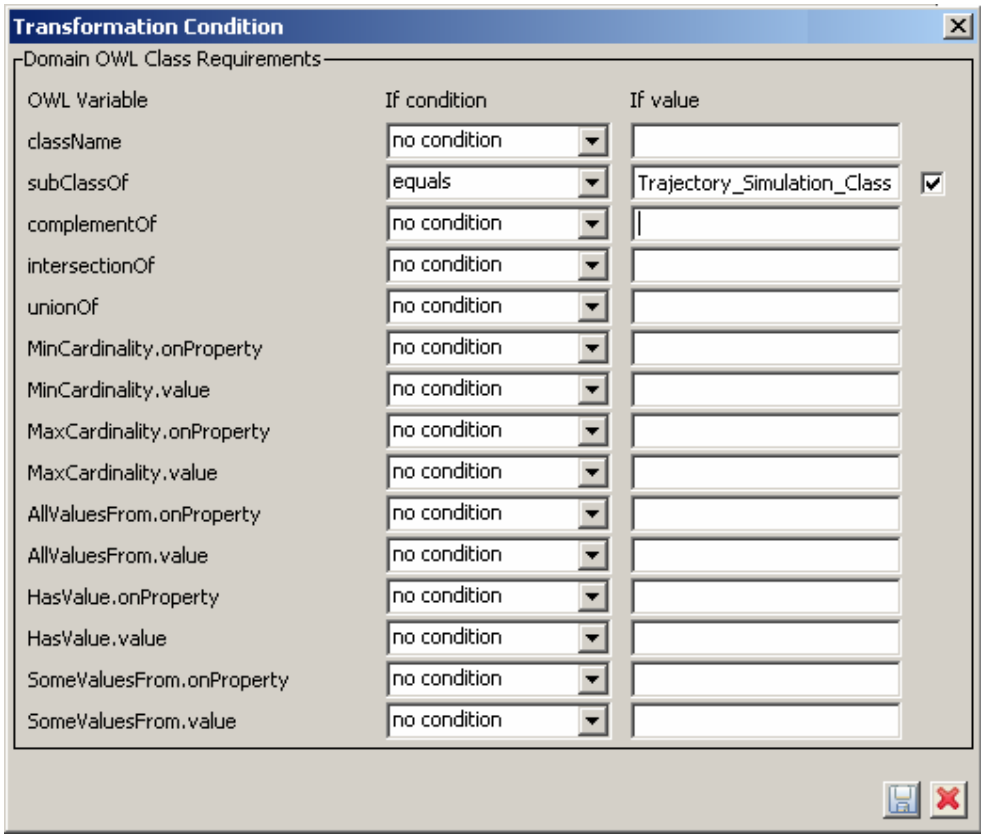



Figure 30 Constraint definition for the first rule

In this rule, we define 5 mappings to create UML classes, super class relationships, operation names for classes, attributes and associations. The rule looks like in Figure 31 after all the 5 mappings are added.

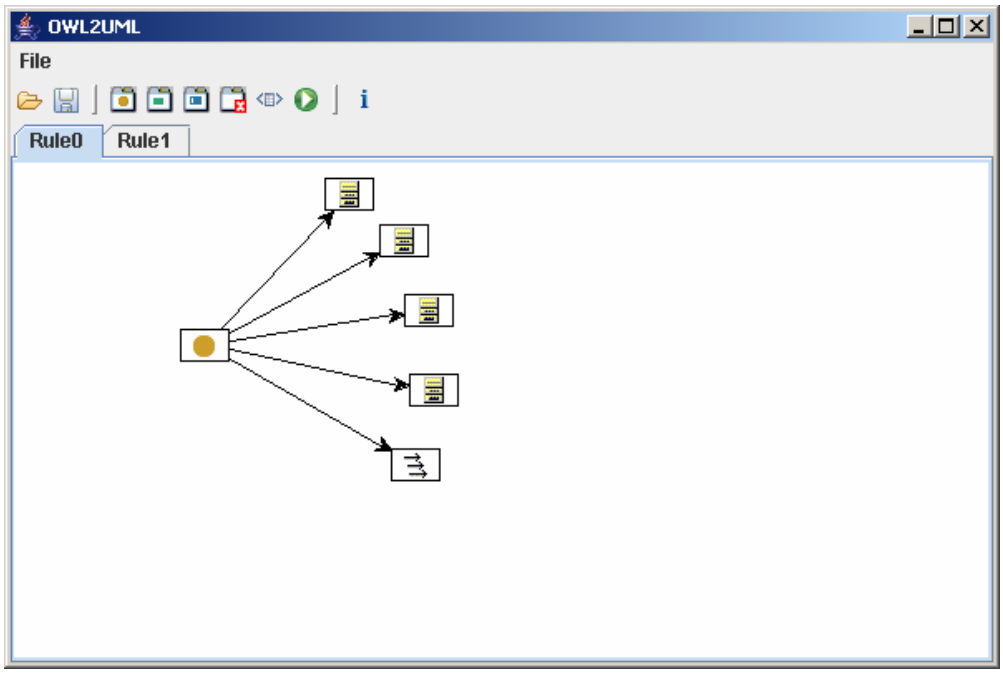



Figure 31 Mappings in the first rule

***Mapping-1: Definition of UML Classes:***

All OWL classes which are subclasses of Trajectory_Simulation_Class must be created as a UML class. This is accomplished by adding a UML class mapping. With this mapping, the tool traverses all OWL classes and if an OWL class is found to be the subclass of Trajectory_Simulation_Class, it adds a new UML class to the target UML model. The name of the new UML class is same as the OWL class name. As a result of this definition, we will have the classes like *Coordinate_System*, *Body_Coordinate_System*, *Earth_Coordinate_System*, *Solver* and all of their subclasses in the resulting UML model.

***Mapping-2: Definition of super classes:***

One other requirement is that, if an OWL class has a subclass relationship, this relationship should also be shown in the UML model. To achieve this, we add a second mapping for UML class with a mapping configuration for super classes. With this configuration, for each subClassOf relationship in OWL, a new super class relationship is added in UML between the corresponding sub/super classes. For example, *Body_Coordinate_System* will be represented as the subclass of *Coordinate_System* in the resulting UML model.

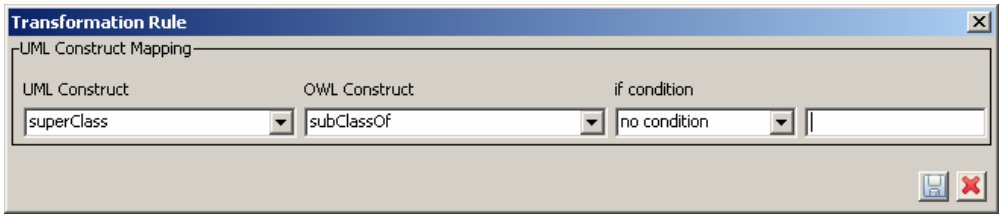



Figure 32 Mapping configuration to set the superclasses

***Mapping-3: Definition of operation names:***

The operation names for each class are to be taken from the AllValuesFrom restrictions if the restriction is defined on a property whose name has the prefix "serves". It is expected that if an OWL class has an AllValuesFrom restriction defined on a property named "servesX", all values at the restriction should be created as an operation for the corresponding UML class. As an example, the OWL class Trajectory_Simulation has an AllValuesFrom restriction on the property servesComputeTrajectory and the value of this restriction is Compute_Trajectory. So an operation named *Compute_Trajectory* will be added to the UML class *Trajectory_Simulation* as a result of this mapping configuration shown in Figure 33.

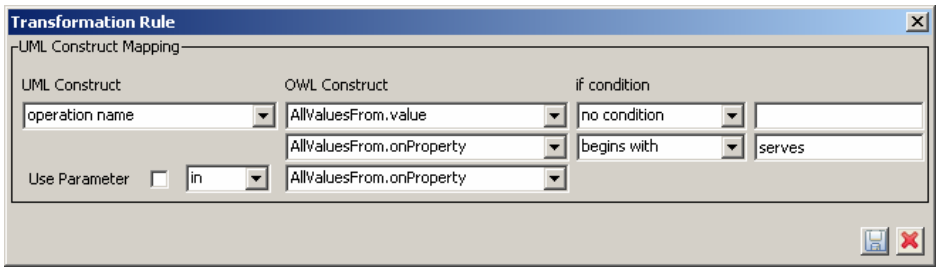



Figure 33 Mapping for the definition of operation names

***Mapping-4: Definition of attributes:***

Attribute information for each class is to be taken from the AllValuesFrom restriction. However in this case, the restriction must be defined on a property whose name has the prefix "has" and the value of the restriction must be used as attribute names of the target UML class. To achieve this, we add a fourth UML class mapping and configure it as in Figure 34. For example, the OWL class Trajectory_Simulation has an AllValuesFrom restriction on the property hasPhase and the value of this restriction is Trajectory_Simulation_Phase. So an attribute named *Trajectory_Simulation_Phase* will be added to the UML class *Trajectory_Simulation* as a result of this mapping configuration.

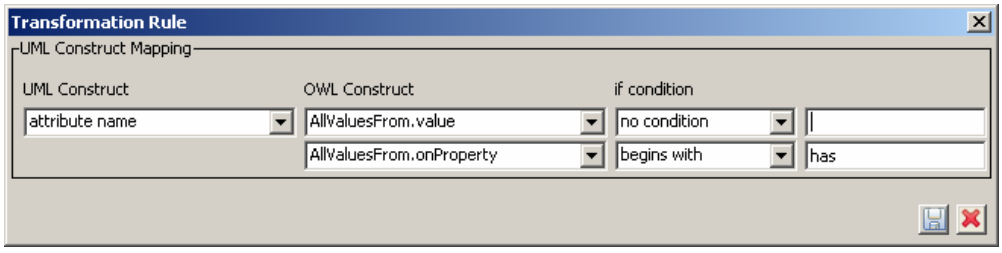



Figure 34 Mapping for the definition of attributes

***Mapping-5: Definition of associations:***

Similar to the attribute definitions, associations must also be taken from the AllValuesFrom restriction values if the restriction is defined on a property whose name has the prefix "has". Type of the association must be "aggregation association". So we add an association mapping to define aggregation associations. Using the same example as the attributes, we will have an aggregation association from *Trajectory_Simulation* class to the *Trajectory_Simulation_Phase* class in the resulting UML model.

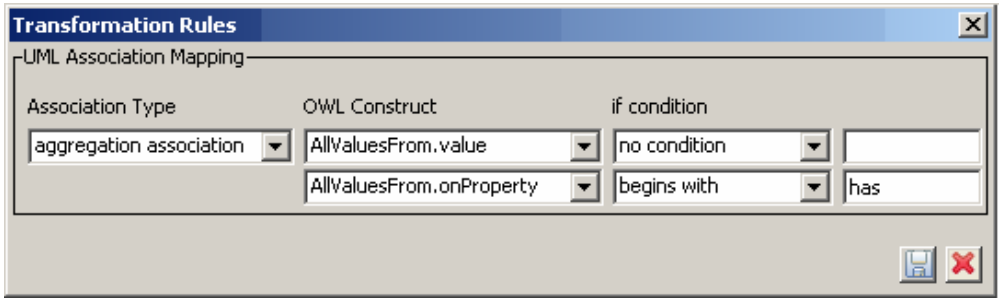



Figure 35 Mapping for the definition of associations

***Rule-2: Rule for operation parameters:***

In Mapping-3 we defined which operations a UML class should have. Now we define mappings to identify from where to get the parameter names for these operations. The source OWL classes for the parameter names are subclasses of Trajectory_Simulation_Function. So we define a constraint in this rule for subclasses, with condition "equals" and constraint value "Trajectory_Simulation_Function".

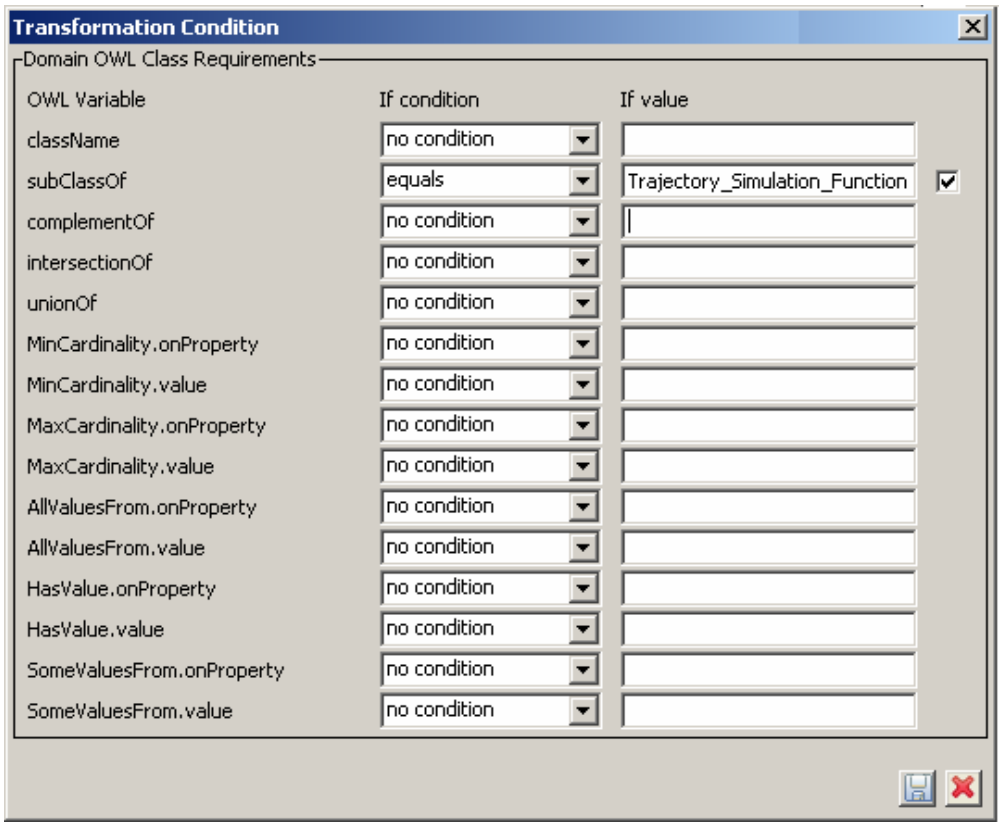



Figure 36 Constraint definition for the second rule

Operation parameters are supposed to be taken from the OWL classes with the same name as the operation name in a UML class. These parameter names must be the values of AllValuesFrom restrictions. If the restriction is defined on a property whose name begins with “in”, the parameter type should be “in”. If the property name begins with “out”, parameter type should be “out”. For these “in” and “out” types, we define two operation parameter mappings in this rule and configure them as in Figure 38 and Figure 39.

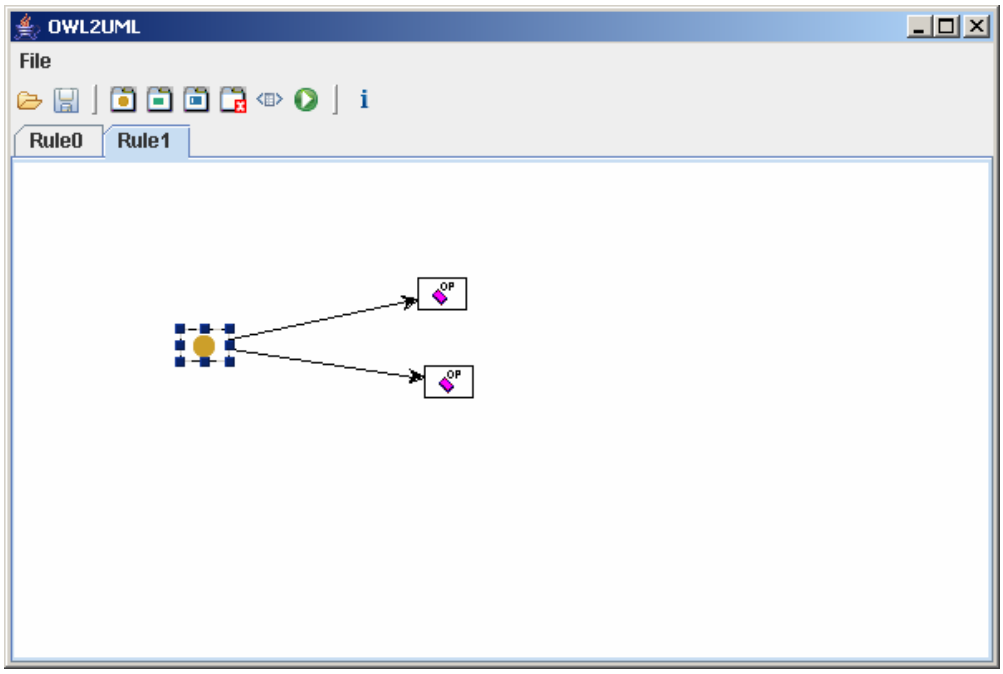



Figure 37 Mappings in the second rule

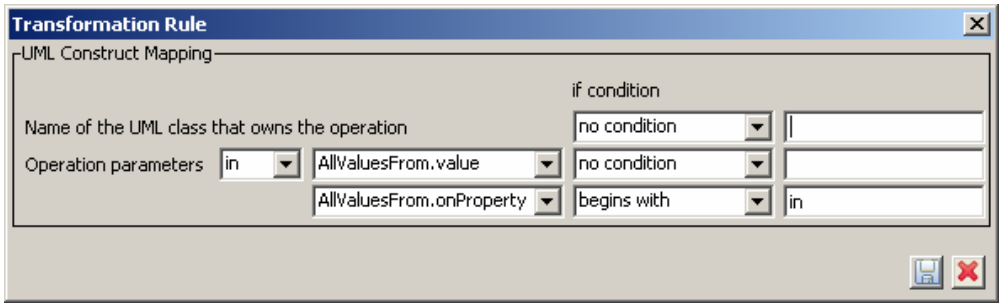



Figure 38 Mapping for the definition of parameters of type “in”

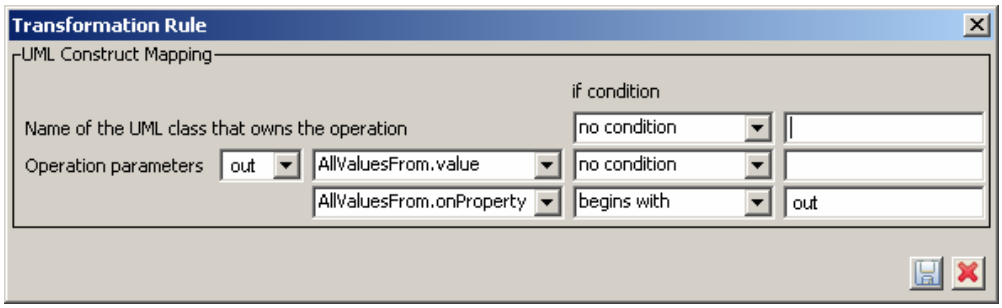



Figure 39 Mapping for the definition of parameters of type "out"

For example *Trajectory_Simulation* class has the operation *Compute_Trajectory* as a result of Mapping-3 in the first rule. During the evaluation of the mappings in this second rule, it is found that the Compute_Trajectory has an AllValuesFrom restriction on the property outTrajectory and the value of this restriction is Trajectory. So the parameter *Trajectory* of type "out" is added to the *Compute_Trajectory* operation in *Trajectory_Simulation* class of the target UML model.

After the definitions of these rules and mappings, we started the transformation process and the UML class diagram represented as an XMI file is produced. We successfully imported this file to Rational Rose XDE for Java [24] and ArgoUML [25]. Figure 40 is a sample from the resulting UML class diagram. Only the important UML structures are shown in this sample, including the examples above.

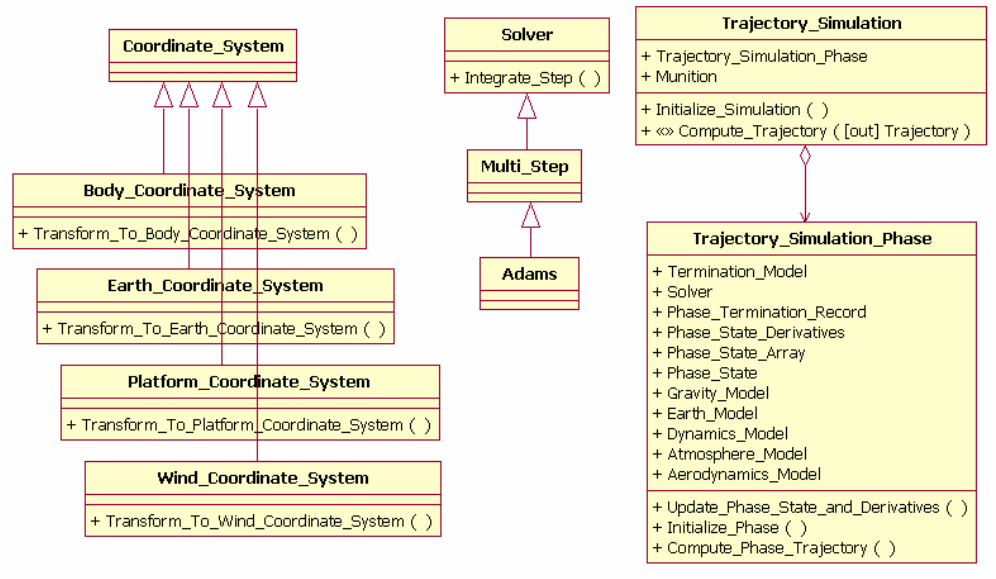



Figure 40 Output UML class diagram, result of the transformation from the input OWL ontology

The outcome of this automated transformation process serves as a part of Platform Independent Framework Architecture, which has been refined into the framework specific to the MATLAB platform [21], [26]. In this refinement step the target problem set is taken into account as well. In addition to class diagrams, Platform Independent Framework Architecture includes manually constructed sequence diagrams to specify the control flow in the framework. An application called Lynx, a surface to surface guided rocket simulation, is implemented by framework completion. The reuse scenario in the development of this application is shown in Figure 41. The dark arrow in the figure indicates the step assisted by our transformation tool. This step was previously performed by hand, which took days of labor and it was prone to trivial errors. With the introduction of our tool, the same step took only a few hours. Additionally, automatic transformation establishes the traceability of the framework architecture to the ontology.

Moreover, we get more uniform results compared to the hand-made transformations.

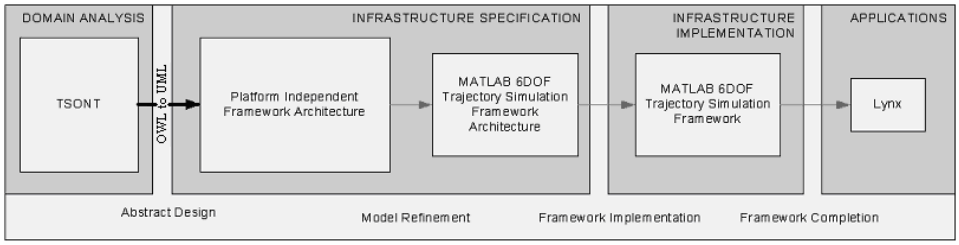



Figure 41 Reuse Scenario in Case of Lynx Development (adapted from [21])

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

## 5.1.Related Work

An early effort on transformation of ontologies into UML class diagrams is DUET [27], [28]. It provides a UML-based environment for visualizing, developing, and manipulating DAML-OIL ontologies. DUET is a plug-in for Rational Rose 2000 and ArgoUML, which can import DAML-OIL ontologies as UML class diagrams into the UML tool, and export a class diagram as an ontology. The tool is not updated in the last few years and it supports only DAML+OIL as the ontology language, not OWL. The most obvious problem with the approach of DUET is that, it does not provide any flexibility to guide the transformation process. In other words, the rules of transformation are hard-coded and can not be modified depending on the user needs. Moreover, it has a tool dependency since it runs only with Rational Rose 2000 and ArgoUML.

OMG's Ontology Definition Metamodel specification [29] includes some formal definitions of how to transform an OWL ontology to a UML model. Just like DUET, this is also a fixed transformation approach. This specification also includes assumptions and ignores some OWL constructs. More importantly, the problem we see in this specification is the high probability of a complex UML model as a result. One reason is that the anonymous classes, including restriction definitions, are created as separate classes in the UML model. The class names for

the anonymous classes are taken from the nodeID of the class in OWL, resulting in classes with meaningless names. The OWL restrictions are attached as comments. Moreover, an OWL property is in some cases transformed to a class attribute and in some cases to an association. All these make the resulting diagram difficult to comprehend.

A transformation from DAML+OIL to XMI using XSLT (Extensible Stylesheet Language for Transformations) is proposed in [30]. Since both of these ontology and UML languages are based on XML, a transformation using XSLT transformation rules is technically possible. However, because of the significant differences between the representation of knowledge in DAML+OIL and XMI, the transformation filters out valuable information such as collections and restrictions.

With the latest releases of IODT (since release 1.1.2), EODM includes a transformation feature from OWL models to EODM Ecore models [31]. This enables the interoperability between OWL models and other EMF supported models. This is because the EODM Ecore model can be used as an intermediate model to support transformation between OWL and other modeling languages. Once the OWL is represented as an EODM Ecore model, it can be transformed to a UML Ecore model through MTF [32] transformations. MTF is a rule-based transformation framework developed by IBM, which provides a language to define mappings between models conforming to the EMF specifications. However, because of the differences in the expressiveness of OWL and Ecore, this approach does not produce the expected results. Some information is lost during this hard-coded, non-configurable transformation from OWL models to EODM Ecore model, such as the restriction definitions or inter-class relationships. The information represented in the resulting EODM Ecore model is very limited, considering all possible OWL constructs. Moreover, even after the production of EODM Ecore model, design of MTF rules to generate the UML Ecore model requires in-depth know-how about MTF and Ecore structures.

Another approach to define mappings between UML and OWL is to define a UML profile for ontologies. A UML profile for ontologies includes some extensions and restrictions on the default UML Metamodel. In [38], some hard-coded mappings between OWL constructs and a UML profile for OWL are introduced. OWL specific constructs are identified mostly using OWL specific stereotypes in UML. For example, a UML class with stereotype "owl::someValuesFrom" indicates that the class in fact represents a restriction. The information like onProperty or restrictionValue of such a restriction is shown with associations in the UML model. Details about this UML profile can be found in [38]. The first problem with this approach is that the resulting UML class diagrams in this approach still include the ontology specific information, especially with the stereotypes. That means, the developer or software engineer who has to understand this UML model must have some ontology knowledge. Another disadvantage is that the resulting model is not very easy to read. For example, a simple restriction definition in OWL can be represented with several stereotyped classes connected with stereotyped associations in UML. The third problem is the "method" in UML classes. According to this approach, since OWL does not have a concept of "method", the resulting UML classes can not have any method.

### 5.2.Contributions and Limitations

The motivation in our work is to provide a user-guided transformation process to bridge the gap between the domain modeling and software modeling realms with minimum loss of information and maximum simplicity. By mapping the OWL and UML constructs on a user interface, the knowledge in an ontology is automatically transformed into a UML class diagram. This class diagram can be understood by a developer, without any need for OWL knowledge.

In addition to the configurable construction of mappings, one other distinguishing feature in our approach is the handling of special semantics in ontologies. Treating the transformation from OWL to UML as an ordinary XML transformation causes

loss of information. Especially the semantics of restrictions and enumerations need special handling and interpretation. Our tool makes it available to use these constructs as the sources of the mappings. It also allows the user to activate some inferences on the ontologies. For example, the subclasses of an OWL class (as well as sub-properties of an OWL property) can be retrieved in a recursive way, whereas it is also possible to process only the direct subclasses of an OWL class.

An OWL class can be defined to be the equivalent or subclass of an anonymous class, where an anonymous class can be a) an OWL restriction, b) the intersection of some OWL classes, c) the union of some OWL classes, d) the intersection of some OWL restrictions, or e) the union of some OWL restrictions. As the anonymous classes do not have class names, our tool does not support the creation of new UML classes for them in the target model. Instead, we allow the constructs included in anonymous classes to be used as source constructs in all three mapping types (see Table 1 and Table 2). For the cases (d) and (e) above, if the user constructs a mapping with OWL restrictions as the source, each OWL restriction definition inside intersectionOf and unionOf is processed to produce the target UML construct.

In OWL, an anonymous class can always include another anonymous class. For example the value of an anonymous restriction definition can be another anonymous restriction definition. However, our tool has a limitation on such recursive anonymous class definitions. Because when an anonymous class includes another anonymous class, it would be hard for the user to define which OWL construct to use as source. Because of this, while processing restrictions, we expect the restriction values to have explicit class names or oneOf definitions (enumeration classes) so that we can transform them to UML constructs. Otherwise, if the restriction value is an anonymous class, which values to use as source in transformation would be a problem.

We strived for simplicity with respect to tool's features. Consider, for example, designation of the visibility (public/protected/private) of attributes. It could be

offered as a part of the user's options in specifying mappings. However, this could complicate the user interface and consistency checking slightly. We have followed a more straightforward approach, which is to set the visibility to "public" as a default for classes and attributes, and to let the user set the visibility on the UML class diagram whenever necessary. Most likely, the private attributes will not show up until implementation considerations are incorporated into the model.

The current version of the tool does not have special handling for specific property types, such as functional properties, transitive properties, or symmetric properties. In addition to this, UML classes are defined using either the OWL class names or OWL property names. In future versions, we are planning to allow the user to define new UML classes using the OWL constructs in a class or property. Depending on the user feedback, we can provide an option to define mappings to Object Constraint Language (OCL) as a target construct [33].

It is also possible to define a fourth mapping type for UML stereotypes. Stereotype is a kind of extension for the classes and associations in UML [11]. Stereotype name is represented by a string in gulliments symbol (<< >>) above the class or association name. It can be used to classify specific constructs in UML to have new modeling elements.

We believe it is worthwhile to try the user-guided approach in other transformation problems as well. In cases where the source and target metamodels are well-structured around comparable basic concepts, and the uses of the target models are clearly defined there seems to be a potential to achieve efficacy of transformations at the expense of negligible effort in constructing them.

# REFERENCES


[1] G. Arango and R. Prieto-Diaz, “Domain analysis concepts and research directions”, in Domain Analysis and Software Systems Modeling, IEEE Computer Society Press, 1991.

[2] OMG, Object Management Group, URL: http://www.omg.org, Last accessed date June 2007.

[3] MDA, OMG Model Driven Architecture, URL: http://www.omg.org/mda, Last accessed date June 2007.

[4] R.Prieto-Diaz, “Domain analysis: An introduction”, in ACM SIGSOFT Software Engineering Notes, ACM Press, 1990.

[5] W. Hesse, “Ontologies in the Software Engineering process”, EAI 2005 - Proceedings of the Workshop on Enterprise Application Integration, Berlin, 2005.

[6] R.A. Falbo, G. Guizzardi, and K.C. Duarte, “An ontological approach to domain engineering”, International Conference on Software Engineering and Knowledge Enginnering, Ischia, Italy, 2002.

[7] OWL Web Ontology Language Overview, URL: http://www.w3.org/TR/2004/REC-owl-features-20040210, Last accessed date June 2007.

[8] M. Horridge, H. Knublauch, A. Rector, R. Stevens, C. Wroe, “A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools”, The University of Manchester, Stanford University, 2004.

[9] U. Durak, H. Oguztuzun, and K. İder, “An ontology for trajectory simulation”, Proceedings of the 2006 Winter Simulation Conference, Monterey, CA, USA, 2006.

[10] Meta-Object Facility, URL: http://www.omg.org/mda/specs.htm#MOF, Last accessed date June 2007.

[11] UML, Unified Modeling Language Infrastructure v2.1.1, URL: http://www.omg.org/cgi-bin/doc?formal/07-02-06, Last accessed date June 2007.

[12] EMF, Eclipse Modeling Framework Project, URL: http://www.eclipse.org/modeling/emf/?project=emf, Last accessed date June 2007.

[13] IODT, Integrated Ontology Development Toolkit, URL: http://www.alphaworks.ibm.com/tech/semanticstk, Last accessed date June 2007.

[14] EODM, EMF Ontology Definition Metamodel, URL: http://www.eclipse.org/modeling/mdt/?project=eodm#eodm, Last accessed date June 2007.

[15] Earth-realm ontology, URL: http://sweet.jpl.nasa.gov/ontology/earthrealm.owl, Last accessed date June 2007.

[16] NASA, Semantic Web for Earth and Environmental Terminology, URL: http://sweet.jpl.nasa.gov, Last accessed date June 2007.

[17] Numerics ontology, URL: http://sweet.jpl.nasa.gov/ontology/numerics.owl, Last accessed date June 2007.

[18] Space ontology, URL: http://sweet.jpl.nasa.gov/ontology/space.owl, Last accessed date June 2007.

[19] Java Regex Library, URL: http://java.sun.com/docs/books/tutorial/essential/regex, Last accessed date June 2007.

[20] XMI, XML Metadata Interchange, URL: http://www.omg.org/technology/documents/formal/xmi.htm, Last accessed date June 2007.

[21] U. Durak, H. Oguztuzun, and K. İder, "An ontology based trajectory simulation framework", Journal of Computing and Information Science in Engineering, accepted for publication.

[22] TUBITAK-SAGE, Scientific and Technical Research Council of Turkey - Defense Industries Research and Development Institute, Ankara, Turkey, http://www.sage.tubitak.gov.tr.

[23] Protégé, URL: http://protege.stanford.edu, Last accessed date June 2007.

[24] Rational Rose XDE Developer for Java, URL: http://www-306.ibm.com/software/awdtools/developer/java/, Last accessed date June 2007.

[25] ArgoUML, URL: http://argouml.tigris.org/, Last accessed date June 2007.

[26] MATLAB Technical Documentation, URL: http://www.mathworks.com, Last accessed date June 2007.

[27] DAML-UML Enhanced Tool, URL: http://codip.grci.com/Tools/index.html, Last accessed date June 2007.

[28] D. Gasevic, D. Djuric, and V. Devedzic, ModelDriven Architecture and Ontology Development, Springer, 2006, pp. 164-165.

[29] ODM, Ontology Definition Metamodel, URL: http://www.omg.org/docs/ad/06-05-01.pdf, Last accessed date June 2007

[30] L. Kuzniarz and M. Staron, "Extracting initial UML domain models from DAML+OIL encoded ontologies", LNCS 2559, pp. 220-231, 2002.

[31] Y. Pan, G. Xie, L. Ma, Y. Yang, and Z. Qui, “IBM research report: An MDA-based system for ontology engineering”, IBM TR RC23795, 2005

[32] MTF, Model Transformation Framework, URL: http://www.alphaworks.ibm.com/tech/mtf, Last accessed date June 2007

[33] OCL, Object Constraint Language, URL: http://www.omg.org/docs/ptc/03-10-14.pdf, Last accessed date June 2007

[34] C. Calero, F. Ruiz, M. Piattini, Ontologies for Software Engineering and Software Technology, Springer, 2006, pp. 249-251

[35] D. Gasevic, D. Djuric, and V. Devedzic, ModelDriven Architecture and Ontology Development, Springer, 2006, pp. 110-111.

[36] Wine ontology, URL: http://www.w3.org/2001/sw/WebOnt/guide-src/wine.owl, Last accessed date June 2007

[37] JGraph, URL: http://www.jgraph.com/, Last accessed date June 2007

[38] S. Brockmans, R. Volz, A. Eberhart, and P. Loeffler, Visual modeling of OWL DL ontologies using UML, Proceedings of the Third International Semantic Web Conference, Springer, 2004, pp. 198-213.

# APPENDIX A

# TOOL USER GUIDE

## A.1. Transformation Tool Window

When the tool is started, en empty window is opened with no transformation configuration.

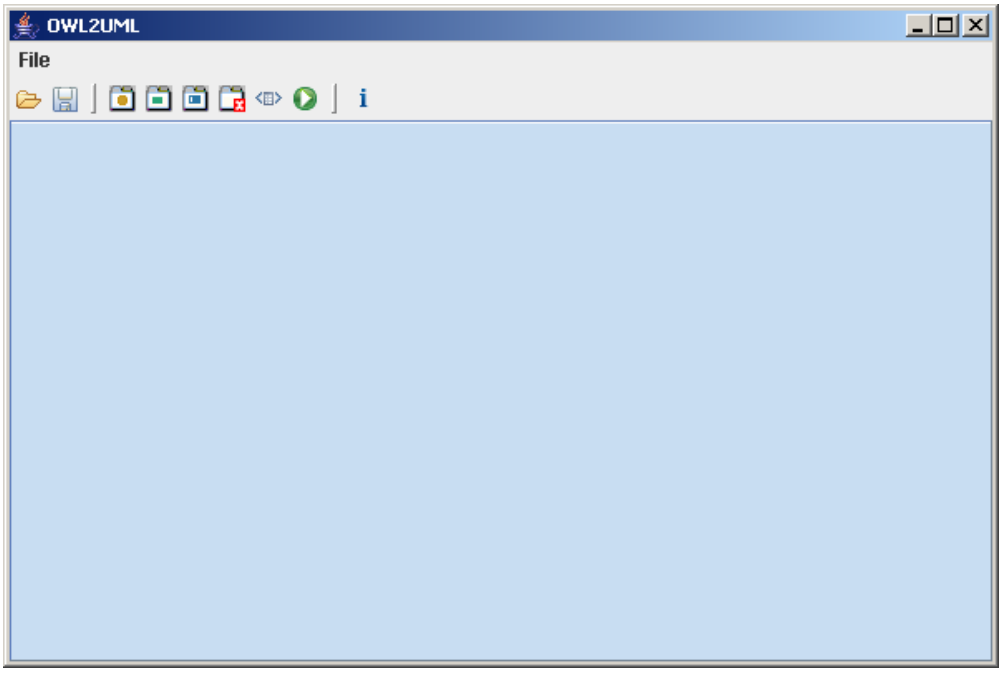



Figure A.1 OWL-to-UML Transformation Window

At the top of the window, there is a menubar which includes the “File” menu. The menu “File” includes the following menu items :

***Save***

Enables the user to save the currently displayed transformation configuration into an XML file.

***Load***

Enables the user to load a previously saved transformation configuration file.

***Close***

Closes the application.

Under the menubar, there is a panel with several buttons. They are used for the following actions:

***Load ( )***

Enables the user to load a previously saved transformation configuration file.

***Save ( )***

Enables the user to save the currently displayed transformation configuration into an XML file.

***Add new rule for OWL classes ( )***

It adds a new tab to the graph panel which represents a rule for OWL classes. The new tab will get a name "ruleX" and all mappings defined in this tab will read only the OWL classes in the ontology as source values. The icon in the graph represents the OWL classes in the source. By right clicking on this icon, constraints can be defined for the rule, i.e. for the source classes.

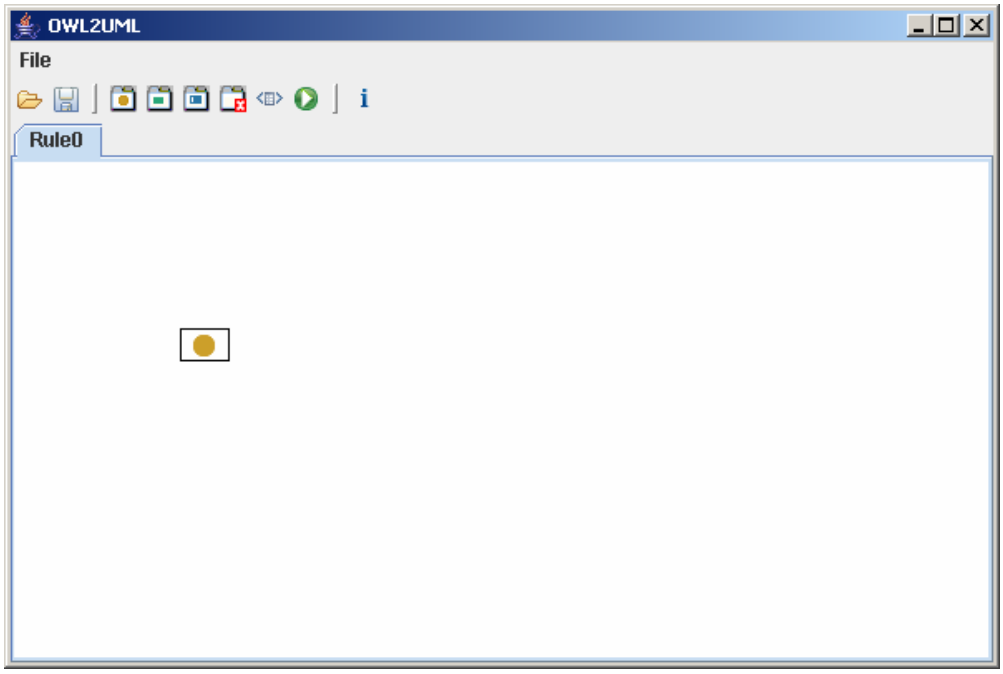



Figure A.2 Rule for OWL class

***Add new rule for OWL datatype properties (***  ***)***

It adds a new tab to the graph panel which represents a rule for OWL datatype properties. The new tab will get a name "ruleX" and all mappings defined in this tab will read only the OWL datatype properties in the ontology as source values. The  icon in the graph represents the OWL datatype properties in the source. By right clicking on this icon, constraints can be defined for the rule, i.e. for the source datatype properties.

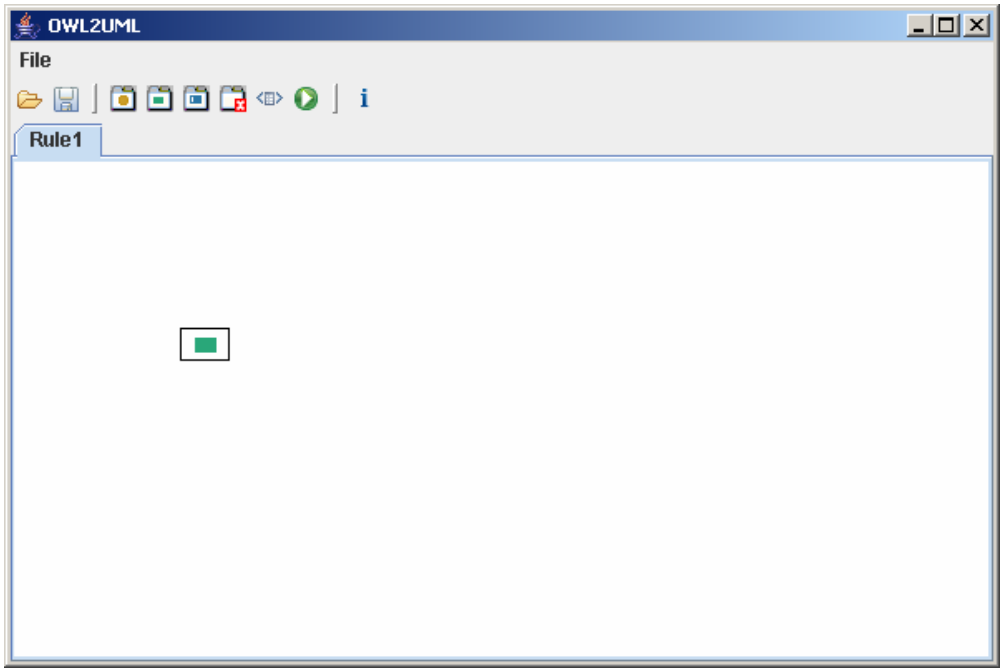



Figure A.3 Rule for OWL datatype properties

***Add new rule for OWL object properties ( )***

It adds a new tab to the graph panel which represents a rule for OWL object properties. The new tab will get a name "ruleX" and all mappings defined in this tab will read only the OWL object properties in the ontology as source values. The icon in the graph represents the OWL object properties in the source. By right clicking on this icon, constraints can be defined for the rule, i.e. for the source object properties.

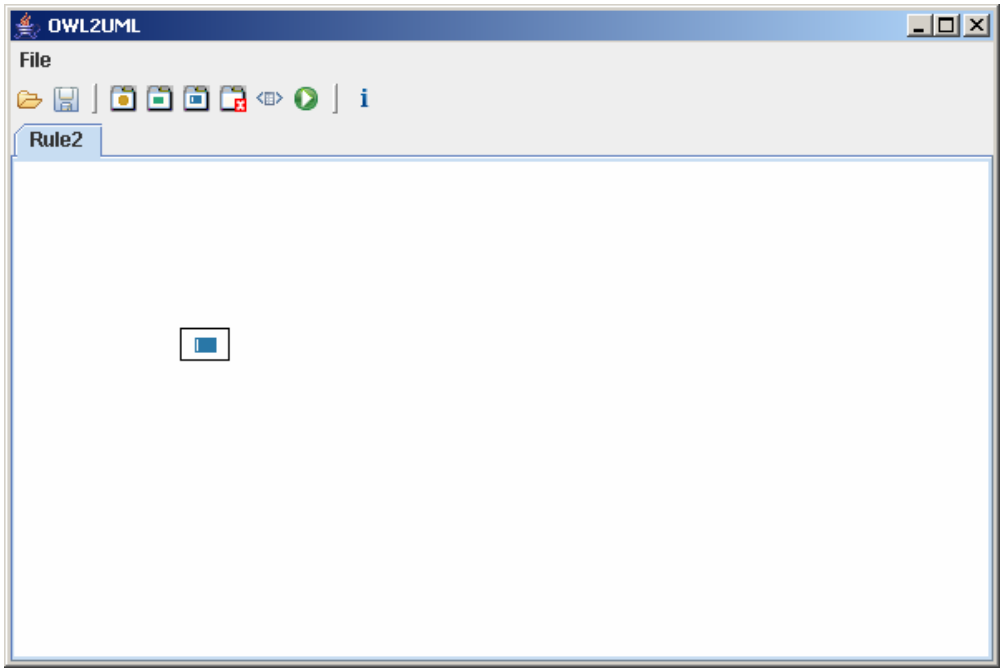



Figure A.4 Rule for OWL object properties

***Remove active rule ( )***

It removes the currently selected rule tab from the window, that is from the transformation configuration. All mappings and configurations in this tab will be removed.

***Set input/output files ( )***

This button opens a panel depicted in Figure A.5. With this panel, the source OWL file and target UML (i.e. XMI) file are set. When the transformation is executed, these files will be used as input and output.

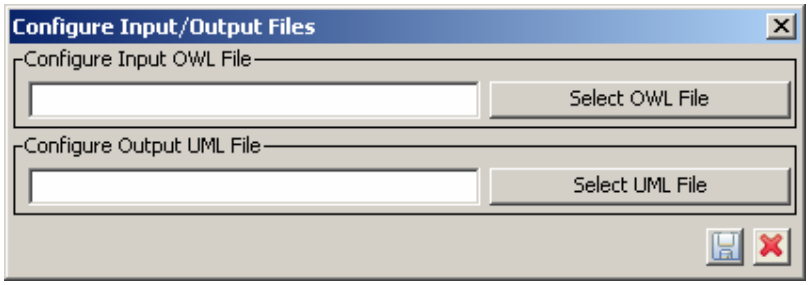



Figure A.5 Set input/output files

***Run transformation ( )***

This button executes the transformation. When the user presses this button, the input ontology is read, rules and mappings are applied to generate the target UML model, and it is written to the output XMI file.

***Help ( )***

Opens the help documents.

## A.2. Configuration of constraints for rules

The constraints for rules restrict the source OWL constructs. Depending on the rule type, different constraints can be defined.

***Constraints for OWL class rules***

When the user right clicks on the  icon, a window is provided to the user to set the rule constraints for OWL classes, as shown in Figure A.6. User can set the "if condition" and the "value" to make the if condition check. For an OWL class to be selected during the execution, all entered criteria must be satisfied by the processed class.

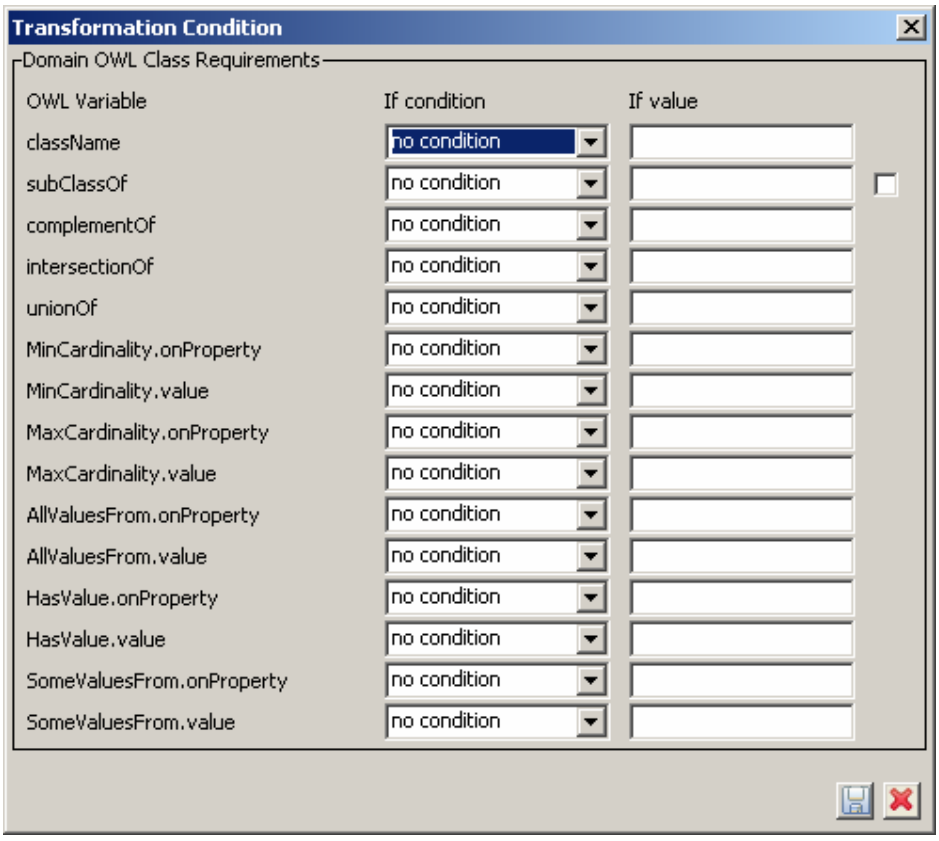



Figure A.6 Constraint configuration window for OWL classes

***Constraints for OWL datatype property rules***

When the user right clicks on the  icon, a window is provided to the user to set the rule constraints for OWL datatype properties, as shown in Figure A.7. User can set the "if condition" and the "value" to make the if condition check. For an OWL datatype property to be selected during the execution, all entered criteria must be satisfied by the processed datatype property.

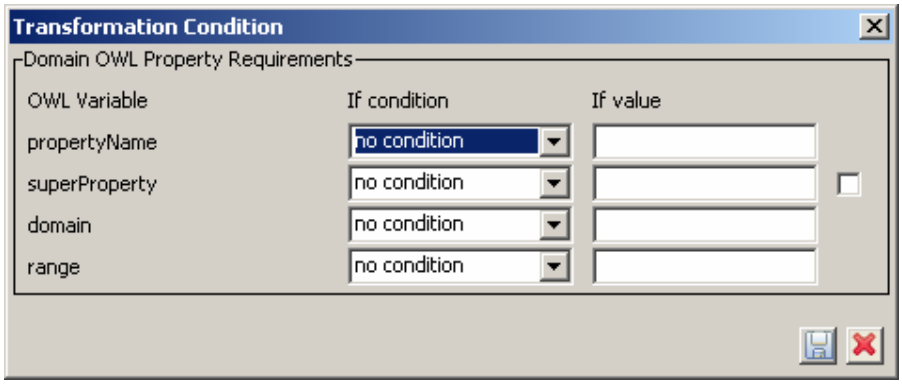



Figure A.7 Constraint configuration window for OWL datatype properties

***Constraints for OWL object property rules***

When the user right clicks on the  icon, a window is provided to the user to set the rule constraints for OWL object properties, as shown in Figure A.8. User can set the "if condition" and the "value" to make the if condition check. For an OWL object property to be selected during the execution, all entered criteria must be satisfied by the processed object property.

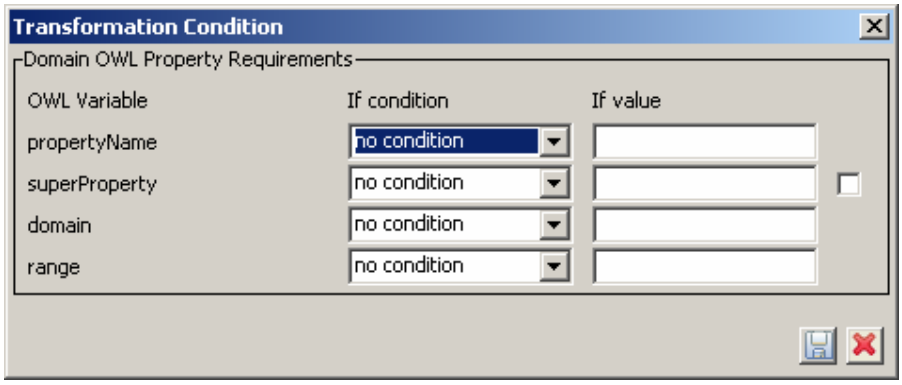



Figure A.8 Constraint configuration window for OWL object properties

## A.3. Adding / Removing / Configuring mappings

### *Mapping to UML class*

If the user clicks on an empty point on a rule graph, a pop-up menu is displayed to the user to add a mapping. If he chooses to add a mapping to UML class, the icon for UML class ( ) is added and an arrow is drawn between the icon representing the source OWL and the new icon representing the target UML. An example is shown in Figure A.9.

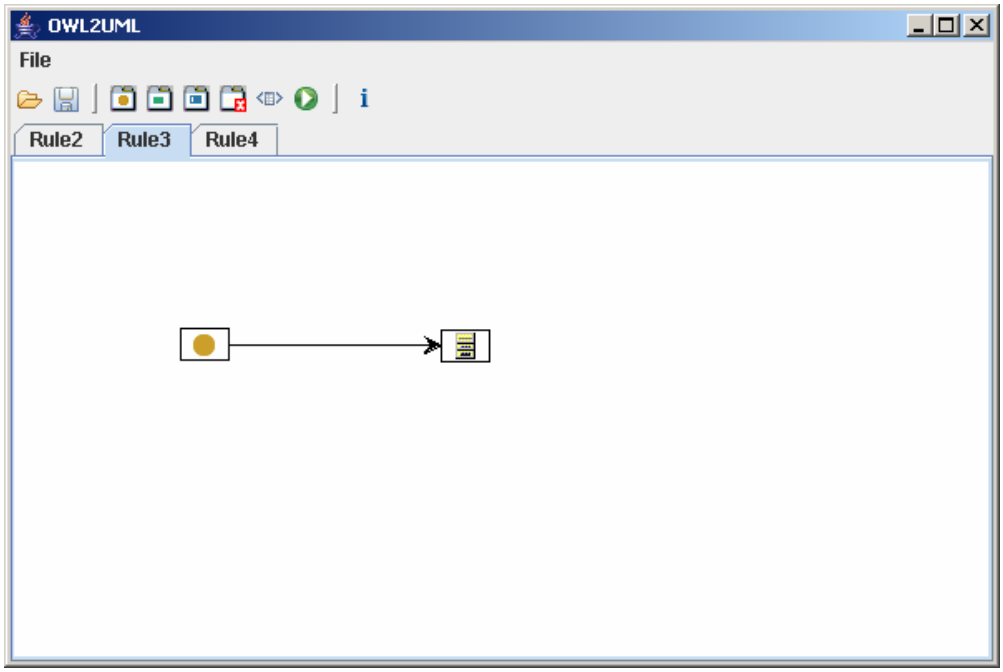



Figure A.9 Mapping between an OWL class and a UML class

Once the mapping is added to the rule, user can remove or configure it. If he right-clicks on the icon, a pop-up menu with 2 items is shown. On the pop-up menu,

“Remove” item is used to remove the mapping from the rule. The “configure” item opens the window in Figure A.10 to configure this mapping.

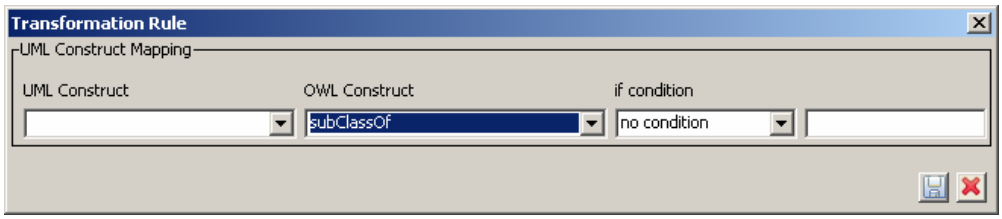



Figure A.10 Mapping configuration between an OWL class and a UML class

The combobox labeled “UML construct” has 3 possibilities, which are “superclass”, “attribute name” and “operation name”. If nothing is entered as the UML construct, the mapping will just create new UML classes in the target. If “superclass” is chosen, the OWL construct in the second combobox will be used to set the super classes of the corresponding class in target. If “attribute name” is chosen, the OWL construct in the second combobox will be used to set the attributes of the corresponding class in target. If “operation name” is chosen, the OWL construct in the second combobox will be used to set the operations of the corresponding class in target. “If condition” and “value” fields are used to apply constraints on this specific mapping.

***Mapping to UML association***

If the user clicks on an empty point on a rule graph, a pop-up menu is displayed to the user to add a mapping. If he chooses to add a mapping to UML association, the icon for UML association (⇉) is added and an arrow is drawn between the icon representing the source OWL and the new icon representing the target UML. An example is shown in Figure A.11.

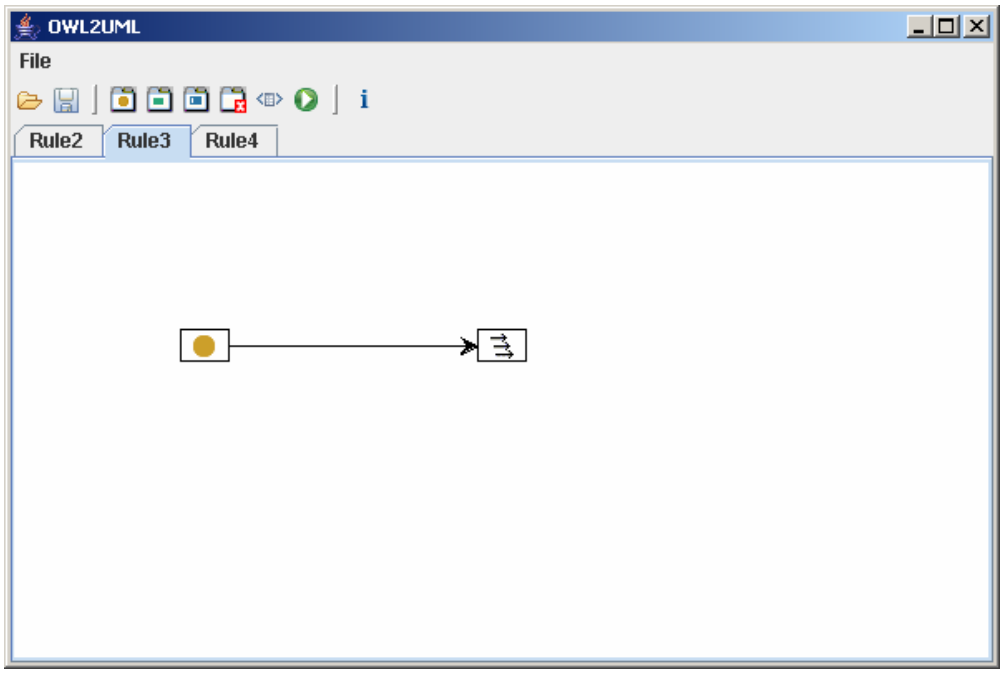



Figure A.11 Mapping between an OWL class and a UML association

Once the mapping is added to the rule, user can remove or configure it. If he right-clicks on the ⇉ icon, a pop-up menu with 2 items is shown. On the pop-up menu, “Remove” item is used to remove the mapping from the rule. The “configure” item opens the window in Figure A.12 to configure this mapping.

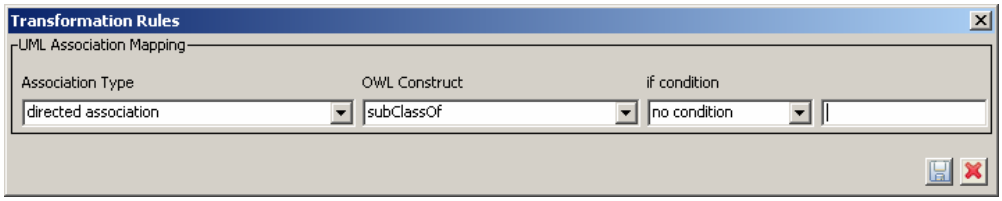



Figure A.12 Mapping configuration between an OWL class and a UML association

The combobox labeled “Association type” has 3 possibilities, which are “directed association”, “aggregation association” and “composite association”. This is used to set the type of the association. The combobox labeled “OWL construct” is used to identify the second end in the association. If the user sets a Min/MaxCardinalityRestriction as the source OWL construct in this field, window will change to the one in Figure A.13. This will enable setting the multiplicity for the association using the restriction value.

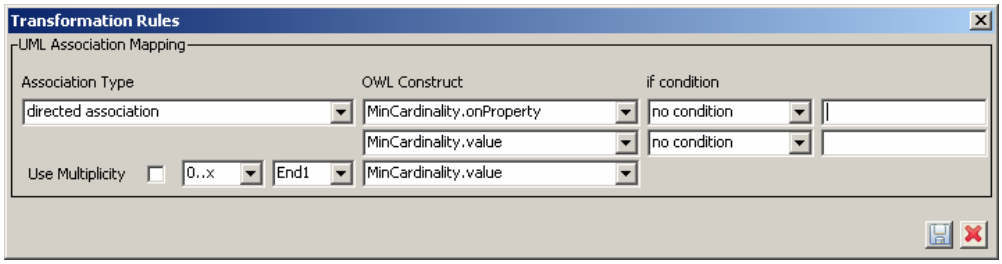



Figure A.13 Mapping configuration for cardinality restrictions

"If condition" and "value" fields are used to apply constraints on this specific mapping. If the OWL construct selected in the mapping does not satisfy the constraint condition, no association is added.

***Mapping to UML operation***

If the user chooses to add a mapping to UML operation, the icon for UML operation () is added and an arrow is drawn between the icon representing the source OWL and the new icon representing the target UML. An example is shown in Figure A.14.

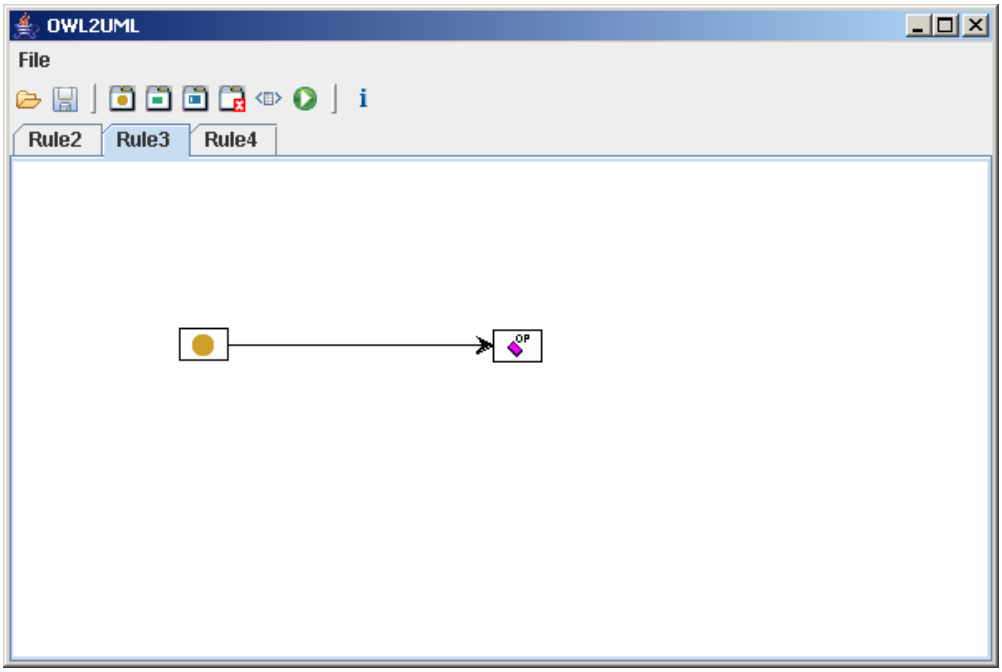



Figure A.14 Mapping between an OWL class and a UML operation

Once the mapping is added to the rule, user can remove or configure it. If he right-clicks on the  icon, a pop-up menu with 2 items is shown. On the pop-up menu,

"Remove" item is used to remove the mapping from the rule. The "configure" item opens the window in Figure A.15 to configure this mapping.

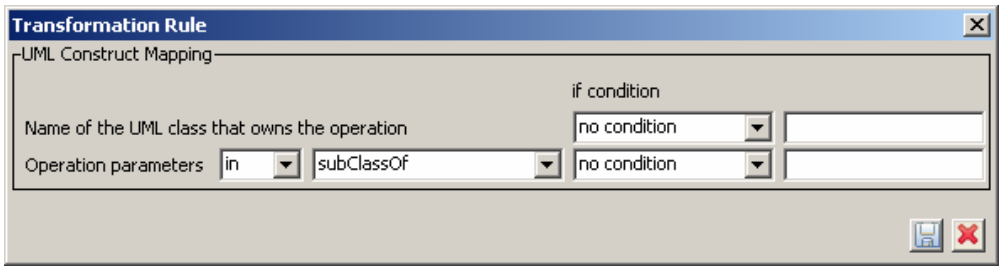



Figure A.15 Mapping configuration between an OWL class and a UML operation

The two comboboxes labeled as "Operation parameters" are used to set the type and source of the operations for this mapping. "If condition" and "value" fields are used to apply constraints on this specific mapping.

## A.4. Running the transformation

After the rules and mappings are configured, user must set the paths for the input and output files. He can save his configuration into an XML file in case he wants to load and use the same configuration later again. Finally he presses the button and output XMI is generated.