Extracting information about domain structure from DAML+OIL encoded ontologies into UML

av
Ludwik Kuzniarz, Miroslaw Staron, Erik Hellman

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Abstract. The report presents and elaborates on the details of knowledge acquisition process from ontologies into domain models. It identifies the knowledge about the domain structure which already exist in form of ontologies, and it also gives the justification why is this knowledge important from domain models perspective. The general idea along with the detailed description and implementation of the process is presented. As the process is based on various XML based technologies, these are shown and described. A small example is introduced for depiction of the practical usage of the method.

Introduction

A vision of using the existing information, and more general, knowledge by different users or agents for specific applications was presented in [1]. The vision was materialized by the introduction of Semantic Web [2,24] where the knowledge is expressed by means of ontologies. An ontology is formal specification of a conceptualization or an abstract model for some phenomenon in the world which identifies relevant concepts of the phenomenon [10,15]. Software development process is an activity where producing an abstract model for a phenomenon in the real world called domain model is also an important issue. The domain model is constructed at a very early stage of the development process and usually is done manually from scratch based on the investigation in the problem domain [4]. But the required knowledge may already and in many cases does exist and is stored as an ontology. So the question is if this knowledge can be automatically acquired to the development process. Formally such acquisition should be a transformation from an ontology specification language to a language used to expressed artefacts produced throughout the development process. During the last several years UML (Unified Modelling Language) [26] has become most widely accepted modelling language used in object-oriented software development process. In the area of ontology specification DAML+OIL language [25] is becoming internationally recognized and widely used. Research towards using UML and its extensions to develop ontologies has been carried out [6,3,7]. The report concerns the reverse process, that is acquisition of the proper knowledge from ontologies to the software development process. More precisely it deals with a basic activity preformed during the analysis phase - understanding the structure of the domain under concern and expressing it in the form of a domain model. The model will express the acquired knowledge of the domain. Within a typical development process the knowledge is acquired via field study and problem investigation. But it seem that it would be desirable to acquire it automatically from the existing ontologies describing the domain under concern. Apart from saving a considerable time such properly done automatic acquisition could guarantee consistency between domain model and existing ontology which otherwise (in case of
manual acquisition) should somehow be proved. The report presents in detail the procedure of acquiring the knowledge from DAML+OIL encoded ontologies into UML domain models. It presents the general idea behind the knowledge acquisition process as well as the details of implementation of the process.

The report elaborates on a detailed procedure for knowledge acquisition process from ontologies into domain models. It begins with a short description of knowledge which is represented by ontologies, which is followed by an outline of domain models, their significance in software development process and the knowledge that is captured within them. The subsequent two paragraphs describe languages for defining ontologies, and language for representing UML models. The paragraphs also present formats of sample documents describing ontology and an UML model. This is followed by the elaboration of knowledge acquisition process, beginning from general presentation and closing with the detailed description. A technology behind the process is also presented and justified. A small example showing how the knowledge acquisition process is performed in practice is presented afterwards. The report is concluded with a general summary of the method and a short discussion of the importance and advantages of the presented method. The two appendixes presented at the very end contain source code for the knowledge acquisition process.

**Knowledge in Ontologies**

According to [10,15] an ontology is formal specification of a conceptualization or an abstract model for some phenomenon in the world which identifies relevant concepts of the phenomenon. Ontologies contain at least one kind of knowledge - knowledge about domain structure (knowledge about the knowledge). This knowledge describes concepts that are significant for the domain. The other kind of knowledge is the knowledge within the domain, which describes concrete instances of concepts and concrete relations between the instances.

The first part of each ontology contains classes, which are concepts from the real world. Each concept has a set of properties describing it. Each concept can also be a specialization (a subclass) of another concept, which means that the specialized concept can be used in every situation when the general concept is required. Each class can also be associated with other classes by means of object properties, which describe relationships between two associated classes. Each end of each object property may have a set of restrictions on it. A subset of the restrictions concerns the cardinalities of instances of a class at the end of the object property, which the restrictions apply to. Each object property can be a subproperty of another object property. In this case, one of the ends is the same as in general object property, and the other end is changed.

The main benefit with ontologies is that it provides a joint terminology between members of a certain domain. This provides a possibility to share and reuse that knowledge. Ontologies structuralize the knowledge and they provide a possibility of encoding the knowledge in machine understandable way. Given the ontology which describes a certain domain, an automated agent can retrieve important information and connect it to another important information. This realizes a vision presented in [1]. What is more, ontologies are a kind of repository containing knowledge about the domains. And this knowledge can be important from the domain modeling perspective. The knowledge can be reused and adapted for creation of new software based on the existing knowledge.

Ontologies are gaining popularity because of their universality. They can be used to describe both the structure of the knowledge and the knowledge itself. They can be extended and they pro-
vide a basic mean of adding semantics to the knowledge they define. The semantics is crucial for the interoperability between the aforementioned autonomous software agents and between separate software systems.

**Knowledge in Domain models**

Software development is a multiple-stage process, which takes as an input a set of requirements, a domain and results in a software product that fulfills the requirements. There is no universal process for development of every possible software. Instead of one, there are many processes, each tailored to the specific company, problem domain or purpose. What is more there are frameworks of software development processes, which allow to be tailored towards specific requirements, thus resulting in a certain software development process. One of such frameworks is Rational Unified Process (RUP) [20], which defines a generic process, which can be customized for a variety of purposes. What is more, it defines certain stages in software development, which should be followed. One of the initial stages is domain modeling (also known as conceptual modelling), which is an analysis of the existing problem domain. It results in producing a domain model. According to [20], the domain model:

... captures the most important types of objects in the context of the domain.

The domain objects represent the entities that exist or events that transpire in the environment in which the system works. [...]

The types of object, mentioned in the definition above are UML classes. They are the same concepts as types of things in ontologies. Therefore, they can be extracted from ontology. Classes are not separate, therefore some relationships must be shown among them. Ontologies also contain the relationships among concepts, the relationships for domain models can be extracted from ontologies. There are two main types of relationships among concepts that are important for domain modelling. The first type is generalization, and the other is association. The latter is a more general relationship, which indicates that instances of one class are connected to the instances of the other class. The foremost is a relationship designating an inheritance between two classes. Classes in domain models also contain attributes - properties of classes. The concepts in ontologies also have properties, which are the same concept as attributes of classes. So the UML class attributes can be extracted from ontologies.

The domain model is crucial within software development process, because it helps to understand the key concepts that are important from the analysed domain perspective. It allows to depict how the concepts are connected, thus providing a way of better understanding the modelled domain.

**Representation of Ontologies**

Ontologies can be defined by mean of a dedicated dialect of Extensible Markup Language (XML) – DAML+OIL [25]. DAML stands for DARPA Agent Markup Language, which is a language for describing contents of web sites for use by mobile agents. OIL [9,14] stands for Ontology Interchange Language, which is a language for defining Ontologies [8]. DAML+OIL is a joint specification of language for defining ontologies that can be used by mobile agents. It is based on and uses another dialect of XML – Resource Description Format (RDF) [11,21,22,23]. RDF is a language for making simple statements about network resources. The structure of statements is defined by RDF Schema - RDFS. RDFS is used to define such concepts as classes, inher-
itance hierarchies and properties of the classes. DAML+OIL provides a set of frame-logic modelling primitives. These are described as a collection of RDF triples. The most common way of expressing DAML+OIL is using an XML based syntax. A DAML+OIL ontology can be expressed in several different syntactic form. The ontology is however equivalent whatever form it is expressed in. DAML+OIL also adds more expression power to RDF/RDFS. It enables to make such statements as for instance that the classes are disjoint. DAML+OIL describes a class in ontology as a set of tags which contain such information as: unique ID of the class, label, comment and one or more superclasses. Properties of classes and the relationships between classes are also represented as sets of tags, which have certain structures. The details of these structures can be found in [25].

Some initial research on Semantic Web was focused on RDF and its extensions as language for defining ontologies [15]. However it has been found that statements, which can be made using RDF/RDFS are insufficient for definitions of ontologies. They are sufficient only for description of a content of web resources. What is more, DAML+OIL allows for definition of knowledge and its structure in the same document.

In DAML+OIL ontologies are represented by a set of tags. Tags which can be used in documents defining ontologies are described in the DAML+OIL specification, which is a W3C recommendation. Although DAML+OIL documents do not have the same structure, there are some elements, which must be present in every DAML+OIL encoded ontology. The common elements are described below. The sections of documents are described on the small excerpt from the ontology in [25].

```
<rdf:RDF>
  <daml:Ontology rdf:about="">
    <daml:versionInfo>
      $Id:daml+oil-ex.daml,v1.9 2001/05/03 mdean Exp $ 
    </daml:versionInfo>
    <rdfs:comment>
      An example ontology, with data types taken from XML Schema
    </rdfs:comment>
  </daml:Ontology>
  <!-- daml classes,instances of these classes,properties, relationships -->
</rdf:RDF>
```

The whole document is enclosed within the <rdf:RDF> tag, which is its root element. There are three main sections of DAML+OIL documents. The first one is a description of the ontology itself, which contains such informations as version, comments and elements imported by the ontology. The second section contains descriptions of classes (concepts in ontology) and relationships among them (ObjectProperties, DatatypeProperties, etc). Each description of a class can be divided into separate parts, the definition of the class, and subsequent, complementary additions to this definition. These additions are sets of tags, which add other elements to the definition they refer to, but cannot substitute the existing elements of the definition. The third part of the document consists of declaration of objects that are instances of concepts defined in this ontology.

Because DAML+OIL is built on top of RDF and RDFS, it uses a lot of constructs from RDF to represent ontologies (elements with prefixes “rdf:” and “rdfs:” respectively).
Representation of UML

Unified Modeling Language specification [26] defines a graphical notation, syntax and semantics of UML model elements but it does not define the internal representation of these model elements. Therefore an XML dialect has been developed to enable tool independent encoding (representation) of UML models, Extensible Metadata Interchange – XMI [27]. Its purpose is to provide a set of tags to describe each UML model element. The full set of XMI tags reflect the syntax specification for UML – its metamodel. XMI specification precisely defines production rules for encoding UML models into XMI. The production rules define a strict mapping between each UML model element and the corresponding set of tags. Every UML model can be encoded in XMI document.

Every XMI document must be structuralized in the same way, which is defined by the DTD provided as part of [26,27]. A simplified structure of XMI documents is shown below. Elements in Italics are sample values, which are not defined by the XMI specification. Commented elements (<!-- -->) indicate sections described farther in this report.

Line 1 contains processing instructions for XML parsers. It is followed by a root element of document. It is the <XMI> tag, with attributes indicating version of XMI standard, its namespace, and creation time. The root element has two sub elements, which are the header and content of the document. The header (lines 4 to 10) describes the documentaion associated with this document, like its creator and version (<XMI.exporter> and <XMI.exporterVersion>), and the metamodel, which instance this document contains (<XMI.metamodel>). There are two attributes of the latter tag. The first one indicates the name of the metamodel (UML in this case) and the second one indicates the version of the metamodel (1.3 in this case). Lines 11 to 21 contain the content part of XMI document. It contains of information about the model itself (lines 12 to 14) and its elements. The information about model consist of its unique identifier (xmi.id), name (name) and other attributes, which are described within the description of classes conversion later in this report.
Lines 15 to 19 contain the main namespace of the model, which contains such elements as classes, associations and generalizations, which are described further in this report.

For instance, in XMI a UML class is defined by a set of tags enclosed between tags `<UML:Class>` and `</UML:Class>`. Tags that are inside class definition represent such model elements as attributes of class `<UML:Attribute>` tag), operations `<UML:Operation>` and a reference to the namespace containing the class. Similarly, the association between classes in UML is also described by a set of tags, enclosed inside `<UML:Association>` and `</UML:Association>`. The detailed description of XMI tags and mapping from UML metamodel to XMI is described in [27].

**Knowledge extraction process**

**Outline**

Acquisition of knowledge is a translation from ontology description language to language used for domain modelling. Figure 1 outlines the translation process. Because there is more knowledge in ontologies than is required in domain models, some of the knowledge must be filtered out. Although it is possible to acquire the knowledge directly from ontology (depicted by a dashed line), it is advisable to introduce additional, intermediate step. It is a filtration of elements of ontologies identified in second and third paragraph of this report. The result of this step is an ontology, which contains only the meaningful concepts, which are transformed into language used for domain modelling. The filtration skips out the elements of ontology which are not meaningful from domain models perspective. If the ontology description language could express only the concepts required (identified in the second paragraph), then the ontology of meaningful concepts and the original ontology would be identical. But since DAML+OIL can express more than its required, then the ontology presented on gray background on figure 1 is a subset of the original ontology, which is also encoded in DAML+OIL. The transition from the ontology to domain modelling language is a translation from DAML+OIL to UML. Since UML can be represented by XMI, which is a dialect of XML, the transformation can be performed automatically with use of technologies dedicated for XML transformations. The details of the transformation and filtration processes are described in the subsequent paragraphs.

![Figure 1: Knowledge acquisition process schema](image-url)
Schema

It is desirable that the acquisition process is automated. A technology used for the automation depends on the language used for ontologies and the language used for domain modeling. If the languages are respectively DAML+OIL and UML, then the process can be performed with use of Extensible Stylesheet Language for Transformations - XSLT [12]. The reason for using XSLT is that dedicated for changing the representation of information between different XML dialects. The process of changing representation is known as transformation. However, to avoid the conflict of names with the transformation which is part of the knowledge acquisition process, in this report it is referred to as XSLT transformation. Both the filtration and the subsequent transformation are both performed by XSLT. The filtration is an XSLT transformation from DAML+OIL encoded ontology into DAML+OIL encoded ontology containing only meaningful concepts. The subsequent transformation is an XSLT transformation, which changes the representation of the same concepts from DAML+OIL encoded ontology to XMI encoded UML domain model. As DAML+OIL is built on top of RDF and RDFS, the transformation is described only for those elements that are expressed with use of DAML+OIL. The relationship and mapping between RDFS constructs and UML are presented in [5].

The process of acquiring the knowledge is a process of finding concepts of interest in an ontology describing the given domain and converting them to UML model elements. The elements of ontology that must be acquired are classes, generalization-specialization relationships, associations and properties of objects. Figure 2 presents the general schema for knowledge acquisition.

Figure 2: A schema of acquiring the knowledge from DAML+OIL to UML with XSLT
(upper box) and transformation (lower box). In the subsequent two paragraphs the details on both the filtration and transformation are presented.

**Filtration**

The filtration process is divided into three steps. Each step deals with one of the three main parts of a DAML+OIL ontology. The first step is to collect all definitions about each class within a single daml:class-node. The definitions we will keep are label, comments, subClassOf and cardinality. This is done by simple specifying which elements we want to copy within our XSLT filter. The second and third step deals with the ObjectProperties and DatatypeProperties. These steps are similar to the first. We copy the elements comment, type, range, domain and subPropertyOf.

This completes the filtration process. We now have a minimized DAML+OIL ontology with only the elements that can be directly represented in UML. This minimization, together with the collection of the definitions of each class within a single daml:class-node, reduces the number of cases we have to handle in the translation.

**Transformation**

The transformation process succeeds the filtration process, so every element from input DAML+OIL ontology is translated into element in the output XMI document. In the subsequent paragraphs the details of the transformation process are described. The layout of the description is the same for every element, starting from the mapping between elements, followed by the justification of the mapping, and closed with the part of XSL stylesheet with transformation rules for this element. Furthermore, in the description of the tag sets, the attributes and values for certain tags are filled with sample values taken from Animal Ontology, which is an example for the walkthrough of DAML+OIL and is described in details in [25].

**Conversion of classes**

Finding UML classes for Domain model is based on finding DAML+OIL classes, which are enclosed between tags <daml:Class> and </daml:Class>. The structure of these tags contains such information as name and description of the class along with the reference to zero or more superclasses of the defined class. A sample set of DAML+OIL tags for a class looks as follows:

```xml
<daml:Class rdf:ID="Male">
    <rdfs:label>Male</rdfs:label>
    <rdfs:comment>Male animal</rdfs:comment>
    <rdfs:SubclassOf rdfs:Resource="#Animal"/>
</daml:Class>
```

After the conversion to the XMI this set of tags has the following result structure:

```xml
<UML:Class xmi.id="Male" name="Male"
    visibility="public" isRoot="true" isLeaf="false" isAbstract="false" isActive="false">
    [...]
</UML:Class>
```
The omitted part (1) is a description of such elements of UML Class as attributes, operations and namespace that the class is enclosed in. Table 1 summarizes the attributes for the XMI tag.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility</td>
<td>public</td>
</tr>
<tr>
<td>isAbstract</td>
<td>false</td>
</tr>
<tr>
<td>isActive</td>
<td>false</td>
</tr>
<tr>
<td>name</td>
<td>Value of <a href="">rdfs:label</a> and &lt;/rdfs:label&gt; tags if they are present. Otherwise value of the rdf:ID attribute</td>
</tr>
<tr>
<td>xmi.id</td>
<td>Value of rdf:ID attribute of <a href="">daml:Class</a> tag</td>
</tr>
<tr>
<td>isRoot</td>
<td>true if the current class does not contain a <a href="">rdfs:subclassof</a> tag, false otherwise</td>
</tr>
<tr>
<td>isLeaf</td>
<td>true if there are no classes that are subclasses of this class, false otherwise</td>
</tr>
</tbody>
</table>

Table 1: Attributes of XMI tag UML Class

Some of the attributes of <UML:Class> tag are derived from DAML+OIL (isLeaf, isRoot, xmi.id, name), while the rest of the attributes have fixed values. The values are set according to the semantics of DAML+OIL elements. Visibility (which in UML designates which elements may have access to this class) is set to public for all classes, because DAML+OIL classes must be visible to all “users” of Ontology. Every DAML+OIL class can have instances, so attribute isAbstract is set to false. The isActive attribute designates a control class [18]. Classes in conceptual (domain) model cannot be active, because they describe elements that exist in real world, not software components, so the value of this attribute is false.

The omitted part (1) contains also the information about the generalizations, of which this class is part of. These are not definitions of such generalization, but references to these definitions. In the document such reference has the following structure:

```xml
<UML:GeneralizableElement.specialization>
  <Foundation.Core.Generalization xmi.idref='AnimalMale' />
</UML:GeneralizableElement.specialization>
```

The attribute xmi.idref of <Foundation.Core.Generalization> tag is the reference to the set of tags representing generalization between two classes. The definition of the generalization is presented in the subsequent paragraph.

Converting generalizations

In DAML+OIL the inheritance, or more precisely the generalization is included as part of the class definition. It is an attribute of <daml:subClassOf/> tag. The generalization in XMI is represented by a set of tags. For the generalization indicated in the previous example (class Male), as a definition of generalization, the following XMI set of tags is produced:

```xml
<UML:Generalization xmi.id="AnimalMale" name=""
  visibility="public" isSpecification="false"
```
Elements of this part of XMI are gathered in table 2. Justification of the attributes that are invented are given below the table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>empty</td>
</tr>
<tr>
<td>visibility</td>
<td>public</td>
</tr>
<tr>
<td>isSpecification</td>
<td>false</td>
</tr>
<tr>
<td>discriminator</td>
<td>empty</td>
</tr>
<tr>
<td>xmi.id</td>
<td>unique value that can be made up by concatenation of names of parent and child classes</td>
</tr>
</tbody>
</table>

**Table 2: Elements of XMI tag for UML Generalization**

Tags `<UML:Generalization.child>` and `<UML:Generalization.parent>` are made of respectively the DAML+OIL class that includes the `<daml:subclassof>` tag and the class that the rdfs:Resource attribute of this tag points to. Each of the XMI tags contains the `<Foundation.Core.GeneralizableElement>` tag, which has an attribute. It is a reference to the element that is either a child and a parent in generalization (respectively to the tag).

The name attribute of `<UML:Generalization>` tag is empty, because the inheritance relationships in DAML+OIL are not identified by names. The generalization relationships that are described by ontologies are always visible, so the visibility attribute is set to public. Because in the course of translation from DAML+OIL to UML, the distinction between specification and realization classes is not made, the isSpecification attribute is false by default. Because there is no such notion in DAML+OIL as dividing the inheritance relationships into partitions, all generalization-specialization relationships have the discriminator attribute set to an empty string.

**Converting associations**

To find associations between classes in UML model, the DAML+OIL must be examined to find ObjectProperty tags. These can be translated into UML associations. The following set of tags must be searched for:

```xml
<daml:ObjectProperty rdf:ID="hasParent">
  <rdfs:domain rdf:resource="#Animal" />
  <rdfs:range rdf:resource="#Animal" />
</daml:ObjectProperty>
```

This pattern should be transformed to the following XMI:
The UML Association itself contains no information about the classes it connects. That information is part of the AssociationEnds, which are the connection points where the UML Association connects the UML Class. The association contains such a general information about connection like its name, visibility, etc. Table 3 explains how the attributes of `<UML:Association>` tag are derived from `<daml:ObjectProperty>` attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the association. It should be the same as the ID attribute.</td>
</tr>
<tr>
<td>xmi.id</td>
<td>value of the rdf:ID attribute</td>
</tr>
<tr>
<td>visibility</td>
<td>public</td>
</tr>
<tr>
<td>isSpecification</td>
<td>false</td>
</tr>
<tr>
<td>isRoot</td>
<td>true</td>
</tr>
<tr>
<td>isLeaf</td>
<td>true</td>
</tr>
<tr>
<td>isAbstract</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3: Elements of XMI tag for UML Association

All values of attributes that are fixed have the same meaning that the corresponding attributes of the `<UML:Class>` tag, and are described there.

For each UML Association there are two AssociationEnds (designated by omitted fragment (2)). Each association end represents a point where the association is connected to UML class. The properties of AssociationEnd describe such concepts as roles, visibilities, etc. As all association ends have the same structure of tags, only one of them is presented below:

```xml
<UML:AssociationEnd xmi.id = "G.2" name = "Animal"
    visibility = "public" isSpecification = "false"
    isNavigable = "true" ordering = "unordered"
    aggregation = "none" targetScope = "instance"
    changeability = "changeable" />
    [...] (3)
</UML:AssociationEnd.type>
    <Foundation.Core.Classifier xmi.idref ="S.1"/>
</UML:AssociationEnd.type>
</UML:AssociationEnd>
```
The explanation of the attributes of XMI part for UML AssociationEnd is in table 4, the tags that are inside the `<UML:AssociationEnd>` and `<UML:AssociationEnd>` tags are discussed below the table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the AssociationEnd, that is the Role at this end of Association. By default it is the name of the class at this end of Association</td>
</tr>
<tr>
<td>xmi.id</td>
<td>Unique ID, which must be generated by the transformation tool</td>
</tr>
<tr>
<td>visibility</td>
<td>public</td>
</tr>
<tr>
<td>isSpecification</td>
<td>false</td>
</tr>
<tr>
<td>isNavigable</td>
<td>true</td>
</tr>
<tr>
<td>ordering</td>
<td>unordered</td>
</tr>
<tr>
<td>aggregation</td>
<td>none</td>
</tr>
<tr>
<td>targetScope</td>
<td>instance</td>
</tr>
<tr>
<td>changeability</td>
<td>changeable</td>
</tr>
</tbody>
</table>

**Table 4: Elements of XMI tag for UML AssociationEnd**

The omitted part of XMI (3) is representing multiplicity at this end of association. It is described in the next paragraph of this report. The tag `<UML:AssociationEnd.type>` contains a tag, which is a reference to the UML Class at this end of association, the actual reference to tag representing the class is the xmi.idref attribute of `<Foundation.Core.Classifier> tag.

The attribute isNavigable indicates that the instances of the class at this association end can be accessed from the instances of the class at the other end of association. As there is no restriction on the navigability between the classes derived from DAML+OIL, the value of this attribute is set to true. As the multiplicities at the end of associations can be more than 1, so the set of instances of classes at this end may be required to be ordered. However, the default value of this attribute is unordered, and during the conversion this value is preserved. The value of the aggregation attribute (none) indicates that the class at this association end is not a part of the class at the other end of the association. The instance value of the targetScope attribute indicates that the instances of the class at this end are associated with instances of class at the other end (as opposed to the classifier scope, which means that the instances of class at this end are associated with class at the other end and not its instances - static). The value of the changeability attribute (changeable) indicates that the instances of the class at this association end can be changed when navigated from the instances of the class at the other end. The rest of attributes for this tag have the same justification that attributes of `<UML:Class>` tag.

The `<daml:ObjectProperty>` tag set is not the only tag that can be translated into UML Associations, the other is `<daml:UniqueProperty>`. The latter has a little different structure, but the only difference in the resulting translation is that the latter has multiplicity equal to 1.
Converting multiplicities for associations

The multiplicities that are part of AssociationEnds are derived from the special kind of restrictions in DAML+OIL. To find the multiplicities, a pattern similar to the following must be searched for:

```
<rdf:subClassOf>
    <daml:Restriction daml:minCardinality="1" daml:maxCardinality="2">
        <daml:onProperty rdf:resource="#hasParent"/>
    </daml:Restriction>
</rdf:subClassOf>
```

The enclosing tag of this pattern is the class that this restriction applies to. This restriction is translated into the following XMI part.

```
<UML:AssociationEnd.multiplicity>
    <UML:Multiplicity >
        <UML:Multiplicity.range>
            <UML:MultiplicityRange lower = "1" upper = "2" />
        </UML:Multiplicity.range>
    </UML:Multiplicity>
</UML:AssociationEnd.multiplicity>
```

The DAML+OIL restriction is mapped into the XMI tag without any attributes added. However, restrictions on cardinalities in DAML+OIL are expressed not only by the daml:Cardinality attribute of the <daml:Restriction> tag. Other possible attributes “daml:Cardinality” translated to both lower and upper attributes of <UML:Multiplicity.range> by the filtration.

Other kinds of restrictions that are present in DAML+OIL ontologies must be translated to OCL constraints of the resulting UML model. The translation is not straightforward and requires significant amount of both code and explanation. Thus the translation is not presented as part of this report.

Implementation

XSLT technology

As mentioned in the previous paragraphs, both the filtration and the transformation are implemented as stylesheets for XSLT, which performs the XSLT transformations on DAML+OIL documents and produces the XMI document.

The XSLT technology is dedicated for transforming between different XML dialects, or transforming an XML document into another format (i.e. RTF). The main benefit is that the XSLT provides a common set of functions for manipulating XML documents. The XSLT processor is a runtime for the XSL transformation stylesheets. Stylesheet realize transformation rules, which are based on pattern matching. Each stylesheet begins with a single pattern to be matched in input documents. The pattern designates that the input document can be transformed into the output document. Certain actions (transformation rules) are taken to perform the transformation. Because the stylesheets realizing the transformations can be very long and incoherent, the technology allows to modularize the stylesheets.

The XSL stylesheet language can be also regarded as a high level language, which is designed to encode not the algorithm of transformation, but rather to define the output of the transformation for a given input. The XSLT technology completely abstracts from any algorithm and therefore it is easy to understand, as the XSL transformation rules are stateless and based on finding (and
matching) patterns in the input document. The role of the stylesheet developer is to define the mapping between constructs in the source document to the constructs in the output documents. The developer does not need to consider any procedure, which enables to extract the desired constructs from the source documents. The extracting is performed by XSLT processor. The XPath (part of XSLT specification) expression defines the constructs extracted from the input document. XPath expressions are in a form of queries performed on the input document and resulting in sequences of references to nodes in the input document.

The implementation of XSLT transformations for filtration and transformation are presented in the appendixes. The code of XSL stylesheets in not described here, because it realizes the extracting and translating of the knowledge. The knowledge was already identified, and the mapping between DAML+OIL and XMI was presented. The dependencies between the modules for transformation are depicted on figure 3. The main module realizing the filtration is filtration.xsl, which

![Figure 3: Source code module dependencies for transformation implementation](image)

is not shown on the figure because it is a single stylesheet, independent from other modules. The main module for the transformation is changeRep.xsl, which imports (thus depends) on three separate modules: classRep.xsl, generalizationRep.xsl and associationRep.xsl. The three modules contain transformation rules for classes, generalizations and associations respectively.

### Using XSLT processor

The XSLT processor which is used in the report to perform the transformations is a publicly available tool “saxon”. It can be obtained from [13]. The processor is a command line tool. To perform the filtration process, the following command should be given at the command line:

```
.saxon.exe ontology.daml filtration.xsl >output.daml
```

Where the ontology.daml is a file with original ontology, filtration.xsl is the file with XSLT transformation rules for the filtration, and output.daml is the resulting ontology containing only the required elements. In order to perform the changing representation:

```
.saxon.exe output.daml changeRep.xsl >output.xml
```

Where the output.daml is the file with the filtered ontology, changeRep.xsl is the file with XSLT transformation rules for the changing representation, and the output.xml is the file with the domain model encoded in XMI. It is ready to import into UML tool.

### Importing XMI encoded domain models into UML tool

An additional step is required to depict the knowledge in domain model into software development process. It is an import the XMI document into the development environment - a UML tool. For the purpose of this report Rational Rose 2001 was used as a sample UML tool. It provides the basic functionality to import and export models in the XMI 1.1 format. The import of the domain
model produces by the XSLT transformation presented in this report can be imported into Rose by choosing the option “Import UML Model” from the “Tools” menu. Please note that some version of Rose may not have the functionality. If this is the case, the add-in for Rational Rose to support XMI can be obtained from the Rational Corporation website [19].

**Example**

The practical benefit of acquiring the knowledge from Semantic Web into Software Development Process is shown on an example in this paragraph. It presents how the ontology describing Animals and people can be used to speed up conceptual analysis phase. The ontology is taken from [25] and is designed for educational purposes. To make it simple and to omit the unnecessary details the ontology has been abridged. The resulting ontology is presented on figure 4. It describes such concepts as Animal, Person, etc. and defines some relationships between them. It also adds some semantics for these classes (like stating that the two concepts are completely separate by relationship disjointUnionOf). Most of the concepts is meaningful for domain analysis, so they are translated to the resulting domain model. The intermediate step - filtering is used here to identify the concepts and also to depicts the difference between the original and filtered ontology. The result of filtering is an excerpt from the original ontology, which another ontology, depicted on figure 5.

The concepts that were filtered out are:

- disjointUnionOf and Collection relationship between classes Man and Woman,
- Restriction on class Person, stating its complementary to class Car. It was a semantic information, which is not included in domain models.
• disjointWith between classes Male and Female, which also was a semantic information not included in domain models.
• Restriction for class Person and ObjectProperty hasFather, because it should be translated into OCL constraint on association hasFather in the resulting model,

The filtered Animal Ontology contains only the elements which will be translated into the UML class diagram, which is a domain model, incorporating concepts significant in the Animal world. The concepts that are:

- Classes: Animal, Male, Female, Person, Man, Woman, HumanBeing and Car,
- ObjectProperties: hasParent and hasFather,
- Restrictions: on class Person and ObjectProperty hasFather, on class Animal and ObjectProperty hasParent,

The next step is the conversion between DAML+OIL and XMI. The converted Ontology is a class diagram representing domain model shown on figure 6 In the course of development process the class diagram can be further simplified and/or extended by some classes that also are important for the domain, but were not included in the ontology. During design of the system, the conceptual class diagram is extended with classes important from programming language perspective. The attributes and operations are added, the whole system is designed. But what is important is that the system is consistent with the knowledge that has and still is used.
The example presented, a simple ontology converted to a simple domain model, a class diagram. For more complex ontologies, the resulting class diagrams will also be more complex and incorporating more concepts. However, the process of acquiring the knowledge is the same.

Conclusions

The report elaborated on the process of knowledge acquisition from DAML+OIL encoded ontologies into UML domain models. The process was described, and the technology behind it was presented. The knowledge required in domain models was identified and the justification that it can be found in ontologies has been shown. The technology behind the acquisition process was described and justified. The example presented in the report shows how the knowledge can be practically extracted with use of the described method. The source code for the transformations is presented in the appendixes.

The process of acquisition knowledge has a very important feature - preserves consistency between the existing knowledge, and the developed software. Without the use of ontologies the consistency checks were done manually by analysts. With the use of ontology and the method presented, the knowledge that is acquired within the process describes the same domain, and as it is derived from the knowledge in ontologies, it is consistent with this knowledge.

The elaborated method and process of knowledge acquisition is a contribution to the research on knowledge sharing and reuse. It proves that the different usages of the same knowledge can be bound together and one can take advantage of the knowledge used by the other. The consistency between knowledge is a contribution to the research on knowledge sharing, and perhaps can be used for development of autonomous agent systems.

References


Appendix B - transformation

Main stylesheet - changeRep.xsl

<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:UML="http://www.omg.org/
  xmlns: RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"高于
<!-- Importing class converting stylesheet-->
<xsl:import href="classRep.xsl" />
<!-- Importing associations converting stylesheet-->
<xsl:import href="associationRep.xsl" />
<!-- Importing generalizations converting stylesheet-->
<xsl:import href="generalizationRep.xsl" />
<xsl:output method="xml" encoding="ISO-8859-1" indent="yes" />
Generalization conversion - generalizationRep.xsl

<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmns:daml="http://www.daml.org/2001/03/daml+oil#"
xmns:UML="http://www.omg.org/
xmns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
<xsl:variable name="part1" select="../@rdf:ID" />
<xsl:variable name="part2" select="substring(@rdf:resource,2)" />
<xsl:if test="$part1 != '' and $part2 != ''">
    <UML:Generalization>
        <xsl:attribute name="xmi.id">
            <xsl:value-of select="concat($part1, $part2)" />
        </xsl:attribute>
        <xsl:attribute name="visibility">public</xsl:attribute>
        <xsl:attribute name="isSpecification">false</xsl:attribute>
        <UML:Generalization.child>
            <Foundation.Core.GeneralizableElement xmi.idref="{$part1}" />
        </UML:Generalization.child>
        <UML:Generalization.parent>
            <Foundation.Core.GeneralizableElement xmi.idref="{$part2}" />
        </UML:Generalization.parent>
    </UML:Generalization>
</xsl:if>
</xsl:template>
</xsl:stylesheet>
Association conversion - associationRep.xsl

<?xml version='1.0' encoding='ISO-8859-1' ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:UML="http://www.omg.org/
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <xsl:template name="associations">
        <xsl:if test="count(rdfs:domain) = 1 and count(rdfs:range) = 1">
            <UML:Association>
                <xsl:attribute name="xmi.id">
                    <xsl:value-of select="@rdf:ID" />
                </xsl:attribute>
                <xsl:attribute name="name">
                    <xsl:value-of select="@rdf:ID" />
                </xsl:attribute>
                <xsl:attribute name="visibility">public</xsl:attribute>
                <xsl:attribute name="isSpecification">false</xsl:attribute>
                <xsl:attribute name="isRoot">false</xsl:attribute>
                <xsl:attribute name="isLeaf">false</xsl:attribute>
                <xsl:attribute name="isAbstract">false</xsl:attribute>
                <UML:Association.connection>
                    <xsl:call-template name="associationEnds" />
                </UML:Association.connection>
            </UML:Association>
        </xsl:if>
    </xsl:template>
    <xsl:template name="associationEnds">
        <UML:AssociationEnd>
            <xsl:attribute name="xmi.id">
                <xsl:value-of select="concat(@rdf:ID, rdfs:domain/@rdf:resource)" />
            </xsl:attribute>
            <xsl:attribute name="name">
                <xsl:value-of select="substring(rdfs:domain/@rdf:resource, 2)" />
            </xsl:attribute>
            <xsl:attribute name="visibility">public</xsl:attribute>
            <xsl:attribute name="isSpecification">false</xsl:attribute>
            <xsl:attribute name="isNavigable">true</xsl:attribute>
            <xsl:attribute name="ordering">unordered</xsl:attribute>
            <xsl:attribute name="aggregation">none</xsl:attribute>
            <xsl:attribute name="targetScope">instance</xsl:attribute>
            <xsl:attribute name="changeability">changeable</xsl:attribute>
            <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity>
                    <UML:Multiplicity.range>
                        <xsl:call-template name="associationMultiplicity" />
                    </UML:Multiplicity.range>
                </UML:Multiplicity>
            </UML:AssociationEnd.multiplicity>
        </UML:AssociationEnd>
    </xsl:template>
</xsl:stylesheet>
<xsl:attribute name="lower">
  <xsl:choose>
    <xsl:when test="count(../rdfs:subClassOf/daml:Restriction/@daml:minCardinality)>0"></xsl:when>
    <xsl:otherwise>1</xsl:otherwise>
  </xsl:choose>
</xsl:attribute>
<xsl:attribute name="upper">
  <xsl:choose>
    <xsl:when test="count(../rdfs:subClassOf/daml:Restriction/@daml:maxCardinality)>0"></xsl:when>
    <xsl:otherwise>1</xsl:otherwise>
  </xsl:choose>
</xsl:attribute>
</xsl:template>
</xsl:stylesheet>
Extracting information about domain structure from DAML+OIL encoded ontologies into UML
av Ludwik Kuzniarz, Miroslaw Staron, Erik Hellman

ISSN 1103-1581
ISRN BTH-RES--02/02--SE

Copyright © 2002 by individual authors
All rights reserved
Printed by Kaserntryckeriet AB, Karlskrona 2002