16 Mapping UML to OWL

16.1 Introduction

Editorial Comment: Issue 10847

This chapter intends to provide an informative comparison between UML and the mandated ontology representation language OWL, in part to motivate the development of the ODM as opposed to a blanket-recommendation that people use UML for ontology representation. It compares the features of OWL Full (as summarized in OWL Web Ontology Language Overview [OWL OV]) with the features of UML 2.0 [UML2]. It first looks at the features the two have in common, although sometimes represented differently, then reviews the features that are prevalent in one but not the other. Little attempt is made to distinguish the features of OWL Lite or OWL DL from those of OWL Full. This overview also ignores secondary features such as headers, comments and version control. In the features in common, a sketch is given of the translation from a model expressed in UML to an OWL expression. In several cases, there are alternative ways to translate UML constructs to OWL constructs. This chapter selects a particular way in each case, but the translation is not intended to be normative. In particular applications, other choices may be preferable.

This Chapter also includes informative formal mappings from UML to OWL and from OWL to UML, both expressed in QVT [MOF QVT].

UML models are organized in a series of metalevels: M3, M2, M1 and M0, as follows:

- M3 is the MOF, the universal modelling language in which modelling systems are specified.
- M2 is the model of a particular modelling system. The UML metamodel is an M2 construct, as it is specified in the M3 MOF.
- M1 is the model of a particular application represented in a particular modelling system. The UML Class diagram model of an order entry system is an M1 construct expressed in the M2 metamodel for the UML Class diagram.
- M0 is the population of a particular application. The population of a particular order entry system at a particular time is an M0 construct.
16.2 Features in Common (More or Less)

16.2.1 UML Kernel

The formal structure of UML is quite different from OWL. What we are trying to do is to understand the relationship between an M1/M0 model in UML and the equivalent model in OWL, so we need to understand how the M1 model is represented in the M2 structure shown. First, a few observations from Figure 71.

- Most of the content of a UML model instance is in the M1 specification. The M0 model can be anything that meets the specification of the M1 model.
- There is no direct linkage between Association and Class. The linkage is mediated by Property.
- A Property is a structural feature (not shown), which is typed. The M1 model is built from structural features.
- Both Class and Association are types.
- A class can have a property which is the structural feature that implements the class.

Editorial Comment: Issue 10852

- A property may or may not be owned by one or more classes a class. A property may be either navigable or not navigable. Associations ends are properties.

It will help if we represent a simple M1 model in this structure (Figure 72).
The properties with their types are:

**Table 10  Properties and Types in Simple Model**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>CourseIdentifier</td>
</tr>
<tr>
<td>description</td>
<td>string</td>
</tr>
<tr>
<td>NumEnrolled</td>
<td>integer</td>
</tr>
<tr>
<td>ID</td>
<td>StudentIdentifier</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
</tr>
</tbody>
</table>

The classes are: Course, Student.

Classes are represented by sets of *ownedAttribute* properties:

**Table 11  Classes and Owned Properties in Simple Model**

<table>
<thead>
<tr>
<th>Class</th>
<th>ownedAttribute Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>code, description, NumEnrolled</td>
</tr>
<tr>
<td>Student</td>
<td>ID, name</td>
</tr>
</tbody>
</table>

Associations are: enrolled

The association can be modeled in a number of different ways, depending on how classes are implemented. If classes are implemented as in Table 11, one way is as the disjoint union of the owned attributes of the two classes.

**Table 12  Implementation of Association in Simple Model**

<table>
<thead>
<tr>
<th>Association</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled</td>
<td>code, description, NumEnrolled, ID, name</td>
</tr>
</tbody>
</table>

But there are other ways to implement a class. If it is known that the property *code* identifies instances of *Course* and that the property *ID* identifies instances of *Student*, then an alternative implementation of *enrolled* is:

**Table 13  Alternative Implementation of Association in Simple Model**

<table>
<thead>
<tr>
<th>Association</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled</td>
<td>code, ID</td>
</tr>
</tbody>
</table>

In this case, the properties *code* and *ID* would be of type *Course* and *Student* respectively.

**16.2.2  Class and Property - Basics**

*Editorial Comment: Issue 10854*

Both OWL and UML are based on classes. A **class** in OWL is a set of *zero or more instances*. A class in UML is a more general construct, but one of its uses is as a set of instances. The set of instances associated at a particular time with a class is called the class’ **extent**. There are subtle differences between OWL classes and UML classes which represent sets.
In UML the extent of a class is an M0 object consisting of instances: a set of zero or more instances of M0 objects. (Instances may be specified at the M1 level in a model library, but they are equivalent to specify possibly several M0 objects.) An instance consists of a set of slots each of which contains a value drawn from the type of the property of the slot. The instance is associated with one or more classifiers. An instance of the class *Course* might be:

**Table 14 Example Course Instance**

<table>
<thead>
<tr>
<th>Classifier</th>
<th>code</th>
<th>title description</th>
<th>NumEnrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>INFS3101</td>
<td>Ontology and the Semantic Web</td>
<td>0</td>
</tr>
</tbody>
</table>

But the M0 implementation representation of a class is not fully constrained. An equally valid instance of *Course* would be the name *INFS3101*, if it were decided that the name would identify an instance of the class. The remainder of the slots could be filled dynamically from other properties of the class.

In OWL, the extent of a class is a set of individuals, which are concretely represented identified by URIs. Individual is defined independently of classes. There is a universal class *Thing* whose extent is all individuals in a given OWL model, and all classes are subclasses of *Thing*. The main difference between UML and OWL in respect of instances is that in OWL an individual may be an instance of *Thing* and not necessarily any other class, so could be outside the system in a UML model. It is of course possible to include a universal class in an M1 model library, but this would be sufficiently unusual to be problematic, whereas the concept is central to OWL.

An OWL class is declared by assigning a name to the relevant type. For example

```xml
<owl:Class rdf:ID="Course"/>
```

An individual is at bottom an RDFS resource, which is essentially a name, so the individual INFS3101 will be declared with something like

```xml
<owl:Thing rdf:ID="INFS3101"/>
```

Relationships among classes in OWL are called *properties*. That the class *course* has the relationship with the class *student* called *enrolled*, which was represented in the UML model as the association *enrolled*, is represented in OWL as a property

```xml
<owl:ObjectProperty rdf:ID = "enrolled"/>
```

Properties are not necessarily tied to classes. By default, a property is a binary relation between *Thing* and *Thing*. 
So, in order to translate the M1 model of Figure 72 to OWL, UML Class goes to owl:Class.

**Table 15 Simple Model Classes Translated to OWL**

<table>
<thead>
<tr>
<th>Class</th>
<th>Owned attributes</th>
<th>OWL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>code, description, NumEnrolled</td>
<td>&lt;owl:Class rdf:ID=&quot;Course&quot;/&gt;</td>
</tr>
<tr>
<td>Student</td>
<td>ID, name</td>
<td>&lt;owl:Class rdf:ID=&quot;Student&quot;/&gt;</td>
</tr>
</tbody>
</table>

The relationships among classes represented in OWL by owl:ObjectProperty and owl:DatatypeProperty come from two different sources in the UML model. One source is the M2 association ownedAttribute between Class and Property, which generates the representation of a class as a bundle of owned attributes as in Table 11. A M1 instance of Class ownedAttribute Property would translate as properties whose domain is Class and whose range is the type of Property. The UML ownedAttribute instance would translate to owl:ObjectProperty if the type of Property were a UML Class, and owl:DatatypeProperty otherwise. The translation of Table 11 is shown in Table 16. Note that UML ownedAttribute M2 associations are distinct, even if ownedAttributes have the same name associated with different classes. The owl property names must therefore be unique. One way to do this is to use a combination of the class name and the owned property name. Note also that since instances of ownedAttribute are always relationships among types, the equivalent OWL properties all have domain and range specified.

An alternative way to give domain and range to OWL properties is to use restriction to allValuesFrom the range class when the property is applied to the domain class. This is probably a more natural OWL specification. However, since all OWL properties arising from a UML model are distinct, the method employed in this chapter is adequate. Should a translation of a UML model be intended as a base for further development in OWL, an appropriate translation can be employed (see Section 16.3).

**Table 16 Simple Model Associations Translated to OWL**

<table>
<thead>
<tr>
<th>Class</th>
<th>Owned property</th>
<th>Type of owned property</th>
<th>OWL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>code</td>
<td>CourseID</td>
<td>&lt;owl:ObjectProperty rdf:ID=&quot;CourseCode&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;rdfs:domain rdf:resource=&quot;Course&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;rdfs:range rdf:resource=&quot;CourseID&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/owl:ObjectProperty&gt;</td>
</tr>
<tr>
<td></td>
<td>description</td>
<td>string</td>
<td>&lt;owl:DatatypeProperty rdf:ID=&quot;CourseDescription&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;rdfs:domain rdf:resource=&quot;Course&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;rdfs:range rdf:resource=&quot;<a href="http://www.w3.org/2001/XMLSchema#string%22/%3E">http://www.w3.org/2001/XMLSchema#string&quot;/&gt;</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/owl:DatatypeProperty&gt;</td>
</tr>
</tbody>
</table>
Note that the translation in Table 16 assumes that a single name is an identifier for instances of the corresponding class. This is not always true. That is there are cases in which a relational database implementation would use a compound key to identify an instance of a class. Since OWL individuals are always unitary names, the translation of the UML class would construct a unitary name from the values of the individual properties. For example, if the association enrolled were treated as a class (UML association class), its representing property might be a concatenation of Course.code and Student.id, so that the link for student 1234 enrolled in course INFS3101 might be translated to an OWL individual with name a globalized equivalent of 1234.INFS3101. Alternatively, a system-defined name could be assigned, linked to each name in the compound key by system-defined properties.

The second source of owl properties in a UML M1 model is the M1 population of the M2 class association. A binary UML association translates directly to an owl:ObjectProperty. The translation of Table 13 is given in Table 17. Note that since associations in UML are always between types, the OWL property always has domain and range specified. If the association name occurs more than once in the same model, it must be disambiguated in the OWL translation, for example by concatenating the member names to the association name.

Table 16 Simple Model Associations Translated to OWL

<table>
<thead>
<tr>
<th>Association</th>
<th>Assn end 1 Property Type</th>
<th>Assn end 2 Property Type</th>
<th>OWL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled</td>
<td>Course</td>
<td>Student</td>
<td>&lt;owl:ObjectProperty rdf:ID=&quot;enrolled&quot;&gt; &lt;rdfs:domain rdf:resource=&quot;Course&quot;/&gt; &lt;rdfs:range rdf:resource=&quot;Student&quot;/&gt; &lt;/owl:ObjectProperty&gt;</td>
</tr>
</tbody>
</table>

Table 17 Sample Associations Translated to OWL

<table>
<thead>
<tr>
<th>Association</th>
<th>Assn end 1 Property Type</th>
<th>Assn end 2 Property Type</th>
<th>OWL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num Enrolled</td>
<td>integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student ID</td>
<td>StudentIdent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 16 Simple Model Associations Translated to OWL

<table>
<thead>
<tr>
<th>Association</th>
<th>Assn end 1 Property Type</th>
<th>Assn end 2 Property Type</th>
<th>OWL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrolled</td>
<td>Course</td>
<td>Student</td>
<td>&lt;owl:ObjectProperty rdf:ID=&quot;enrolled&quot;&gt; &lt;rdfs:domain rdf:resource=&quot;Course&quot;/&gt; &lt;rdfs:range rdf:resource=&quot;Student&quot;/&gt; &lt;/owl:ObjectProperty&gt;</td>
</tr>
</tbody>
</table>

Both languages support the subclass relationship (OWL rdfs:subClassOf, UML generalization). Both also support subproperties (UML generalization of association or meta-associations among properties like subsetting or redefining). UML defines generalization at the supertype classifier, while in OWL subtype and subproperty are separately but identically defined.

Editorial Comment: Issue 10868

The translation from UML to OWL is straightforward. If <S, G> is an M1 instance of the UML M2 metaclass generalization (S is a subclassifier of G), then if both S and G are classes and TS, TG are respectively the types of the identifying owned property of S, G respectively, the OWL equivalent is the addition of the clause <rdfs:subClassOf rdf:resource="TG"/> to the definition of the OWL class TS. Similarly if S and G are both associations, the owl equivalent is the addition of the clause <rdfs:subPropertyOf rdf:resource="G"/> to the definition of the OWL object property S. Note that subassociations can be defined in a number of ways, including by OCL.
Figure 73 M1 Model with Association Class

An association in UML can be N-ary. It can have its own ends (ownedEnd). An association can also be a class (association class), so can participate in further associations. In OWL DL, classes and properties are disjoint, but in OWL Full they are potentially overlapping. However, there is limited syntactic mechanism in the documents so far published to support this overlap. There is an advantage in translating these more complex UML associations to structures supported by OWL DL. In any case, the translations proposed are not normative, so those responsible for a particular application can use more powerful features of OWL if there is an advantage to doing so.

Our proposal takes advantage of the fact that an N-ary relation among types $T_1 \ldots T_N$, or an association class with attributes, is formally equivalent to a set $R$ of identifiers together with $N$ projection functions $P_1, \ldots, P_N$, where $P_i:R \rightarrow T_i$. Thereby N-ary UML associations are translated to OWL classes with bundles of binary functional properties.

Figure 73 extends the model of Figure 72 by making enrolled an association class which owns an attribute grade. The association class enrolled is a member end of an association instructor, whose other member end is staff. Some students enrolled in a given course may be assigned to one staff member as instructor, some as another.
The model of Figure 73 is represented in table form in Table 18. The association class \textit{enrolled} is represented by its two end classes, \textit{Course} and \textit{Student}, the attribute of the association class \textit{Grade}, and by an owned attribute \textit{enrolledR} which implements the association class as a class, in the same way as in Table 12 and Table 13.

The implementation of \textit{enrolled} and \textit{Instructor} in Table 18 is translated into OWL as follows:

\begin{verbatim}
<owl:Class rdf:ID="enrolled"/>
<owl:FunctionalProperty rdf:ID="enrolledCourse">
  <rdfs:domain rdf:resource="enrolled"/>
  <rdfs:range rdf:resource="Course"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="enrolledStudent">
  <rdfs:domain rdf:resource="enrolled"/>
  <rdfs:range rdf:resource="Student"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="enrolledGrade">
  <rdfs:domain rdf:resource="enrolled"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="enrolledInstructor">
  <rdfs:domain rdf:resource="enrolledR"/>
  <rdfs:range rdf:resource="Staff"/>
</owl:FunctionalProperty>
\end{verbatim}

### 16.2.3 More Advanced Concepts

There are a number of more advanced concepts in both UML and OWL. In the cases where the UML concept occurs in OWL, the translation is often quite straightforward, so will not always be shown in this section. See Section 16.3 and Section 16.4 for full details.

Both languages support a module structure, called \textit{package} in UML and \textit{ontology} in OWL. The translation of package to ontology is straightforward. Both languages also support \textit{namespaces}.

Both UML and OWL support a fixed defined extent for a class (OWL \textit{oneOf}, UML \textit{enumeration}). Note that in UML enumeration is a datatype rather than a class.
UML has the option for associations to have distinguished ends which can be navigable or non-navigable. A navigable property is one which is owned by a class or optionally an association, while a non-navigable is not (the end might be of type integer, say). OWL properties always are binary and have distinguished ends called domain and range. A UML binary association with one navigable end and one non-navigable end will be translated into a property whose domain is the navigable end. A UML binary association with two navigable ends will be translated into a pair of OWL properties, where one is inverseOf the other.

A key difference is that in OWL a property is defined by default as having range and domain both Thing. A given property therefore can in principle apply to any class. So a property name has global scope and is the same property wherever it appears. In UML the scope of a property is limited to the subclasses of the class on which it is defined. A UML association name can be duplicated in a given diagram, with each occurrence having a different semantics. It is possible, though not customary, to include a universal superclass in an M1 model library. This is sufficiently unusual that it is not clear what the current toolsets would do with it.

An OWL individual can therefore be difficult to represent in a UML model. UML has a facility dynamic classification which allows an instance of one class to be changed into an instance of another, which captures some of the features of Individual, but an object must always be an instance of some class. UML models rarely include universal classes.

Both languages allow a class to be a subclass of more than one class (multiple inheritance). Both allow subclasses of a class to be declared disjoint. (In OWL, all classes are subclasses of Thing, so any pair of classes can be declared disjoint.) UML allows a collection of subclasses to be declared to cover a superclass, that is to say every instance of the superclass is an instance of at least one of the subclasses. The corresponding OWL construct is the declare the superclass to be the union of the subclasses, using the construct unionOf.

UML has a strict separation of metalevels, so that the population of M1 classes is distinct from the population of M0 instances and also the M1 model libraries. OWL Full permits classes to be instances of other classes. UML only models classes of classes in the context of declaration of disjoint or covering powertypes.

In OWL, a property when applied to a class can be constrained by cardinality restrictions on the domain giving the minimum (minCardinality) and maximum (maxCardinality) number of instances which can participate in the relation. In addition, an OWL property can be globally declared as functional (functionalProperty) or inverse functional (inverseFunctional). A functional property has a maximum cardinality of 1 on its range, while an inverse functional property has a maximum cardinality of 1 on its domain. In UML an association can have minimum and maximum cardinalities (multiplicity) specified for any of its ends. OWL allows individual-valued properties (objectProperty) to be declared in pairs, one the inverse of the other.

So if a binary UML association has a multiplicity on a navigable end, the corresponding OWL property will have the same multiplicity. If a binary UML association has a multiplicity on its both ends, then the corresponding OWL property will be an inverse pair, each having one of the multiplicity declarations.

For an N-ary UML association, multiplicities are more problematic to map to OWL. For example, in Figure 74, the multiplicities show that given instances of event, Olympiad and competitor, there is at most one instance of result; given instances of event, Olympiad and result there is at most one instance of competitor; given instances of Olympiad, competitor and result there may be many instances of event (an athlete may compete at several events in the same Olympiad and finish in the same place in each); and given instances of event, competitor and result there may be many instances of Olympiad (an athlete may compete in the same event at several Olympiads and finish in the same place in each). For an N-ary UML association, any multiplicity associated with one of its end classes will apply to the OWL property translating the corresponding projection from the association class to the translated end class.
The N-ary association in Figure 74 would be translated as a class `competes` whose instances are instances of links in the association, and four properties whose domain is `competes` and whose ranges are the classes attached to the member ends of the association. Since one instance of a link includes only one instance of the class at each member end, all the properties are functional. The multiplicities on the UML diagram do not translate to OWL in a straightforward way.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl   "http://www.w3.org/2002/07/owl#">
  <!ENTITY rdf   "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY rdfs  "http://www.w3.org/2000/01/rdf-schema#">
  <!ENTITY xsd   "http://www.w3.org/2001/XMLSchema#"> ]>
<rdf:RDF xmlns:rdf="&rdf;"
  xmlns:rdfs="&rdfs;"
  xmlns:owl="&owl;"
  xmlns:xsd="&xsd;"/>
<owl:Class rdf:ID="competes">
  <owl:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="competesEvent"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
    </owl:Restriction>
  </owl:subClassOf>
  <owl:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="competesCompetitor"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
    </owl:Restriction>
  </owl:subClassOf>
  <owl:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="competesYear"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
    </owl:Restriction>
  </owl:subClassOf>
  <owl:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="competesPosition"/>
      <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
    </owl:Restriction>
  </owl:subClassOf>
</owl:Class>
```

**Figure 74 Example N-ary Association with Multiplicity**
In UML, multiplicities can be defined on both ends of an association. In OWL, general multiplicities apply to the range instances associated with a given domain instances. In both cases, multiplicities can be strengthened (minimum increased or maximum decreased) when associations/properties are applied to subclasses.

Note that the class might be the domain of a property for which the individual might not have a value. This can happen if the mincardinality of the domain of the property is 0, in which case the property is optional (or partial) for that class. The same can happen in UML. An instance of a class is constrained to participate only in properties which are mandatory, minimum cardinality > 0. So an instance can lack optional properties. (The somewhat strange construct maxCardinality < minCardinality is syntactically correct in OWL and has the semantics that the property has no instances. It can occur where multiple autonomous ontologies are merged, for example.)

However, even if the property is mandatory (mincardinality > 0 and maxcardinality >= mincardinality), there may not be definite values for the property. Consider a class (K) for which a property (P) is mandatory. In this case, the individual (I) must satisfy the predicate

\[\text{[M]}: I \text{ instance of } K \rightarrow \exists X \text{ such that } P(I) = X.\]

It is not required in OWL that there be a constant C such that X = C. All horses have color, but we may not know what color a particular horse has.

In UML, there is a strict separation between the M1 and M0 levels. At the M1 level, that an association is mandatory (minimum cardinality greater than 0) is exactly the predicate \([M]\). Any difference between UML and OWL must come from the treatment of the model of the M1 theory at the M0 level. In practice, M0 models in UML applications tend to be ground Herbrand models implemented by something like an SQL database manager. For these cases, if we know a horse has a color, then we know what color it has. To the extent that UML tools and modelling build this expectation into products, conflict can occur when interoperating with an OWL ontology.
But UML does not mandate M0 models to be Herbrand models. In particular SQL-92 supports the Null value construct, which has multiple interpretations, including “value exists but is not known”. Some years ago, CJ Date proposed a zoo of nulls with specific meanings, including “value exists but is not known”, and there have been proposals by Ray Reiter and others for databases with either existentially quantified variables in the data or which reason with the M1 theory for existentially quantified queries. It is possible for a particular application to introduce a special constant “unknown” into a class, which is treated specially by the programs. UML does not forbid an implementation of a class model in one of these ways. So there is no difference in principle between UML and OWL for properties which are declared to have minCardinality greater than 0 (and maxCardinality >= minCardinality) for a class.

Note that a consequence of this possible indeterminacy, it may not be possible to compute a transitive closure for a property across several ontologies, even if they share individuals.

An OWL property can have its range restricted when applied to a particular class, either that the range is limited to a class (subclass of range if declared) (allValuesFrom) or that the range must intersect a class (someValuesFrom). UML permits these and other restrictions using the facilities specializes or refines.

OWL allows properties to be declared symmetric (SymmetricProperty) or transitive (TransitiveProperty). In both cases, if the domain and range are not type compatible, the property is empty. UML uses OCL for this purpose.

OWL permits declaration of a property whose value is the same for all instances of a class, so the property value is in effect attached to the class (OWL DL property declared as allValuesFrom a singleton set for that class). OWL full allows properties to be directly assigned to classes without special machinery. In OWL, if class A is an instance of class B, then a property P whose domain includes B will designate a value P(A) which can apply to the class A so can be derived for all instances of A.

UML allows a property to be derived from other model constructs, for example a composition of associations or from a generalization. That a property is derived can be represented as an annotation in OWL. The actual derivation rule can not in general be represented in OWL (OWL does not support arithmetic, for example). Derivation rules in UML are expressed in OCL, and there is no general translation of OCL to OWL.

A classifier in UML can be declared abstract. An abstract classifier typically cannot be instantiated, but may be a superclass of concrete classifiers. There is no OWL equivalent for this.

Two different objects modeled in UML may have dependencies which are not represented by UML named (model) elements, so that a change in one (the supplier) requiring a change in the other (the client) will not be signaled by for example association links. Two such objects may be declared dependent. There are a number of subclasses of dependency, including abstraction, usage, permission, realization and substitution. OWL does not have a comparable feature except as annotations, but RDF, the parent of OWL, permits an RDF:property relation between very general elements classified by RDFS:Class. Therefore, a dependency relationship between a supplier and client UML model element will be translated to a reserved name RDF:Property relation whose domain and range are both RDF:Class. Population of the property will include the individuals which are the target of the translation of the supplier and client named elements.
16.2.4 Summary of More-or-Less Common Features

This section has described features of UML and OWL which are in most respects similar. Table 19 summarizes the features of UML in this feature space, giving the equivalent OWL features. UML features are grouped in clusters which translate to a single OWL feature or a cluster of related OWL features. The column Package shows the section of the UML Superstructure document [UML2] where the relevant features are documented.

Table 19 Common Features of UML and OWL

<table>
<thead>
<tr>
<th>UML elements</th>
<th>Package</th>
<th>OWL elements</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>class, property ownedAttribute, type(^a)</td>
<td>7.3.7 Classes</td>
<td>class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.8 Classifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.32 Multiplicities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instance</td>
<td>7.3.22 Instances</td>
<td>individual</td>
<td>OWL individual independent of class</td>
</tr>
<tr>
<td>ownedAttribute, binary association</td>
<td>7.3.7 Classes</td>
<td>property</td>
<td>OWL property can be global</td>
</tr>
<tr>
<td>subclass, generalization</td>
<td>7.3.7 Classes</td>
<td>subclass</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.8 Classifiers</td>
<td>subproperty</td>
<td></td>
</tr>
<tr>
<td>N-ary association, association class</td>
<td>7.3.7 Classes</td>
<td>class, property</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.4 Association Classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enumeration</td>
<td>7.3.11 Datatypes</td>
<td>oneOf</td>
<td></td>
</tr>
<tr>
<td>disjoint, cover</td>
<td>7.3.21 Generalization sets</td>
<td>disjointWith, unionOf</td>
<td></td>
</tr>
<tr>
<td>multiplicity</td>
<td>7.3.32 Multiplicities</td>
<td>minCardinality</td>
<td>OWL cardinality declared only for range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>maxCardinality</td>
<td></td>
</tr>
<tr>
<td>package</td>
<td>7.3.37 Packages</td>
<td>ontology</td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>7.3.12 Dependencies</td>
<td>reserved name RDF:property</td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) This cell summarizes the relationship between UML class and OWL class mediated by property, ownedAttribute and type. It does not signify that the latter three are themselves translated to OWL class.

All of the UML features considered in the scope of the ODM have more-or-less satisfactory OWL equivalents. Some UML features have no OWL equivalents, as summarized in Section Table 20. Some OWL features in this feature space have no UML equivalent, so are omitted from Table 19. They are summarized in Table 21. Besides the small differences in the features in the feature space common to UML and OWL, there are some more general differences described in sections 16.5 and 16.6.

Table 20 UML features with no OWL equivalent

<table>
<thead>
<tr>
<th>UML elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>navigable, non-navigable</td>
</tr>
<tr>
<td>derived</td>
</tr>
<tr>
<td>abstract classifier</td>
</tr>
<tr>
<td>Classes as instances</td>
</tr>
</tbody>
</table>
This section describes mappings from [UML2] models to ODM OWL models. The UML2 metamodel is based on ptc/04-10-02. The mapping is limited to OWL DL, which means only OWL-DL constructs will be used in mapping definitions. There are many abstract metaclasses in UML2 kernel package, so only important concrete classes are mapped to OWL constructs.

Mappings are expressed in QVT [QVT]. A brief tutorial is presented in ***Appendix H***.

Mappings are shown for all constructs in Table 19 for which there is an OWL equivalent, except for Instance. The Instance model is not part of the classes model, and is intended to show partial specifications of instances rather than concrete instances. The profiles represent OWL individuals as singleton classes rather than UML instances. So the mappings do not include Instance.

### 16.3 UML to OWL

This section describes mappings from [UML2] models to ODM OWL models. The UML2 metamodel is based on ptc/04-10-02. The mapping is limited to OWL DL, which means only OWL-DL constructs will be used in mapping definitions. There are many abstract metaclasses in UML2 kernel package, so only important concrete classes are mapped to OWL constructs.

Mappings are expressed in QVT [QVT]. A brief tutorial is presented in ***Appendix H***.

Mappings are shown for all constructs in Table 19 for which there is an OWL equivalent, except for Instance. The Instance model is not part of the classes model, and is intended to show partial specifications of instances rather than concrete instances. The profiles represent OWL individuals as singleton classes rather than UML instances. So the mappings do not include Instance.

#### 16.3.1 Naming Issues

In OWL, all objects are identified either by uniform resource identifiers (uri) or by an arbitrarily assigned identifier unique within the ontology (blank nodes). A typical method is for objects within an ontology to be identified by uri which is a fragment on a base uri which identifies the ontology. It is also possible for an object to have a uri independent of that of the ontology. Blank node identifiers can be treated as fragments in this way during the course of the mapping, even though the identifiers do not persist. A uri is conceptually global. It universally identifies the same object no matter where it appears.

In UML, objects are identified by name within a minimally disambiguating context. If there are several packages involved in a mapping, they have different names. But other packages may exist elsewhere which have the same name. Within a package, classes, associations, and some other objects are identified by names unique to the package. Lower level kinds of objects like properties are identified by names unique within their parent object. For example several different classes may have attributes with the same name.

Ontologies in OWL are free-standing objects which can import one another. Packages in UML can also import one another, but in addition there is a standard procedure by which several packages may be merged into one.

A critical problem in mapping between UML and OWL is the generation of appropriate identifiers for objects in the target model instance given the identifiers of the relevant objects found in the relevant pattern in the source model instance. Since the mappings proceed from the packaging constructs to their components, the first problem is generation of an identifier for the target packaging construct given the identifier of the source packaging construct. If the source is an OWL ontology, one possibility is to identify the target package with the same uri as the ontology. However, this method violates the spirit of the uri, since the same uri now identifies two different objects which could evolve independently. If the source is a package, a base uri must be constructed for the target ontology. There is not enough information available in the UML model instance to generate a globally unique uri.

<table>
<thead>
<tr>
<th>Table 21 OWL features with no UML equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thing, global properties, autonomous individual</td>
</tr>
<tr>
<td>allValuesFrom, someValuesFrom</td>
</tr>
<tr>
<td>SymmetricProperty, TransitiveProperty</td>
</tr>
<tr>
<td>Classes as instances</td>
</tr>
<tr>
<td>disjointWith, complementOf</td>
</tr>
</tbody>
</table>

214 Ontology Definition Metamodel
Because of these incompatibilities, we have made use of two only partly specified relations, PackageNameToUriBase and URIRefToName. PackageNameToUriBase takes a package name and creates a uri suitable to be extended by fragment identifiers. URIRefToName takes a uri reference, possibly a fragment on a base uri, and generates a name unique to the uri reference. (The relation takes distinct uri references to distinct names.)

Further, in mapping from UML to OWL, the target of objects whose names are unique within packages are identified by uri references which are fragments on the uri base of the corresponding ontology. Targets of objects whose names are unique only within a narrower context are identified by fragment identifiers generated by concatenating the name of the source object with the names of its context objects starting with the object whose name is unique to a package. Thus an attribute bar of a class foo would map to an object with fragment identifier foobar.

### 16.3.2 Package To Ontology

Each object in both the source and target model instance must have an identification scheme. Both UML and OWL support the concept of a namespace represented as a packaging construct, called Package in UML and ontology in OWL. Individual objects are contained in packaging constructs, and identified with respect to the identifier of their packaging construct.

A package is a namespace for its members, and may contain other packages. Only packageable elements can be owned members of a package. By virtue of being a namespace, a package can import either individual members of other packages, or all the members of other packages.

![Diagram of UML Package to OWL Ontology]

**Figure 75 Map UML Package to OWL Ontology**

transformation UMLToOWL (uml:UML, owl:OWL)

// transform UML model to OWL model
16.3.3 Class To Class

The mapping from Class to OWLClass includes the transformation of generalization relationships between Classes.

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. It has the same semantics of RDFSubClassOf in RDF Schema, and the two ends of the generalization relationships can be accessed by the source and target that are defined in DirectedRelationship.
Figure 76 Map UML Class to OWL Class [1]

Figure 77 Map UML Class to OWL Class [2]
top relation UClassToOClass
//map UML Class to OWL Class
{  
cn: String;
checkonly domain uml uc: Class{name=cn, owningPackage=p: Package};
enforce domain owl oc: OWLClass{uriRef=: URIReference {uri = ref : UniformResourceIdentifier, fragmentIdentifier=:LocalName {name=cn}}, ontology=o};
// no need to name the larger constructs unless they are used or generated elsewhere
when{
    PackageToOntology(p, o);
    ref = o.uri; // Provides a base uri for the fragment identifier
}
}//UClassToOClass

top relation GeneralizationToSubClassOf
//map generalization hierarchy to rdfs:subClassof
{  
checkonly domain uml uc: Class{superClass=gen: Class};
enforce domain owl oc: OWLClass{RDFSsubClassOf=super: OWLClass};
when{
    UClassToOClass(uc, oc);
    UClassToOClass(gen, super);
}
}// GeneralizationToSubClassOf

16.3.4 Attribute to Property

The ownedAttribute defines the attributes owned by the class. It is an ordered set of Properties, which can be mapping to either OWLDatatypeProperty or OWLObjectProperty. If a property is part of the memberEnds of an Association, the mapping of it will be discussed in Association mapping section [defined in Section 16.3.5 and Section 16.3.6].

If the type of the property is a PrimitiveType, the property is mapped to the OWLDatatypeProperty. If the type of the property is an Enumeration, and the ownedLiteral of the Enumeration has specification as ValueSpecification, then the property is OWLDatatypeProperty.

If the type of the property is Class, or the ownedLiteral of an Enumeration type has at least one classifier, the property can be mapped to OWLObjectProperty.

top relation AttributeToObjectProperty
// maps ownedAttribute where property's type is a class to an object property
// notice that the enforce creates the more specific object OWLObjectProperty, then the more general structure Property is created in the where clause. The when clause both forces the relation to wait until the classes have been mapped, and gives the mappings for the classes.
{
    checkonly domain uml prop: Property{class = cl : Class, type = tp : Class};
enforce domain owl op: OWLObjectProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran : OWLClass};
when {
    UClassToOClass(cl, dom);
    UClassToOClass(tp, ran);
}
where {
    PropertyToProperty(prop, op);
}
}// AttributeToObjectProperty
relation PropertyToProperty
// not a top relation. Intended to be called in a where clause of the relation mapping one of the more specific
metaclasses. It fills in the structures relevant to the superclass Property. Assumes the naming conventions for a
property used as an attribute.
{
    cn, pn : string;
    checkonly domain uml prop : Property(name = pn, class = :Class{
        name = cn, owningPackage=p:Package});
    enforce domain owl op : Property(uriRef=:URIReference {
        uri = ref : UniformResourceIdentifier, fragmentIdentifier=:LocalName(name=cnpn)), ontology=o);
    when{
        PackageToOntology(p, o);
        ref = o.uri;  // Provides a base uri for the fragment identifier
    }
    where {
        cnpn = cn + pn; // Follows naming conventions to disambiguate property name
    }
} // PropertyToProperty

top relation AttributeToDatatypeProperty
// maps ownedAttribute where property's type is a primitive type to datatype property
{
    checkonly domain uml prop : Property(class = cl : Class, type = tp : PrimitiveType);
    enforce domain owl op : OWLDatatypeProperty(RDFSdomain = dom : OWLClass, RDFSrange = ran : Literal);
    when {
        UClassToOClass(cl, dom);
        UMLPrimTypeToLiteral(tp, ran);
    }
    where {
        PropertyToProperty(prop, op);
    }
} // AttributeToDatatypeProperty

top relation UMLPrimTypeToLiteral
// maps UML primitive type names to OWL literal type names
// To be implemented
{} // UMLPrimTypeToLiteral

16.3.5 Binary Association To Object Property

An association specifies a semantic relationship that can occur between typed instances. It has at least two ends
represented by properties, each of which is connected to the type of the end. More than one end of the association
may have the same type. In this section, only binary association is discussed. In Section 16.3.4, instances of
OWLObjectProperty have been created. However, the possible OWLinverseOf relationship between two
navigableOwnedEnd of an association has not been created. AssociationToObjectProperty relation is used to set
OWLinverseOf relationships among related properties.

Further, associations both of whose ends are properties with the same type will be mapped to symmetric properties in
OWL.

Note that in this strategy the UML association name is not mapped, so even though the OWL to UML mapping
recognizes inverse pairs and maps them to associations, the association name is not recoverable.
top relation AssociationToPropertyPair
// Association whose ends are of different types goes to pair of inverse properties
{
  checkonly domain uml assn : Association{
    memberEnd = ps : Sequence(Property){
      prop1 : Property (type = tp1:Class),
      prop2 : Property(type = tp2 : Class)};
  // Note checkonly clause succeeds only when there are exactly two memberEnds
  enforce domain owl oprop1 : OWLObjectProperty{
    RDFSdomain = cl2:OWLClass, RDFSrange = cl1 : Class, OWLInverseOf = oprop2 : OWLObjectProperty{
      RDFSdomain = cl1:OWLClass, RDFSrange = cl2 : Class}};
  when {
    not prop1.type = prop2.type;  // ends are of different type
    UClassToOClass(tp1, cl1);
    UClassToOClass(tp2, cl2);
  }
  where {
    EndPropertyToProperty(prop1, oprop1);
    EndPropertyToProperty(prop2, oprop2);
  }
}
relation EndPropertyToProperty
// not a top relation. Intended to be called in a where clause of the relation mapping one of the more specific metaclasses. It fills in the structures relevant to the superclass Property. Assumes the naming conventions for a property used as an association end.
{
  an, pn : string;
  checkonly domain uml prop : Property(name = pn, association = :Association{
    name = an, owningPackage=p:Package});
  enforce domain owl op : Property(uriRef=:URIReference{
    uri = ref : UniformResourceIdentifier, fragmentIdentifier=:LocalName{name=anpn}}, ontology=o);
  when{
    PackageToOntology(p, o);
    ref = o.uri;  // Provides a base uri for the fragment identifier
  }
  where {
    anpn = an + pn; // Follows naming conventions to disambiguate property name
  }
} // EndPropertyToProperty

top relation SymAssociationToSymProperty
// Association where both properties are of the same type to symmetric property
{
  checkonly domain uml assn : Association{
    memberEnd = ps : Sequence(Property) {
      prop1 : Property{type = tp :Class},
      prop2 : Property{type = tp :Class}}};
  // checkonly clause succeeds only if there are exactly two member ends.
  enforce domain owl oprop1:SymmetricProperty(RDFSdomain = cl:OWLClass, RDFSrange = cl:OWLClass);
  when {
    UClassToOClass(tp, cl);
  }
  where {
    EndPropertyToProperty(prop1, oprop1);
  }
} // SymAssociationToSymProperty

16.3.6 Association Classes and N-ary Associations

An AssociationClass can be seen as an association that also has class properties, or as a class that also has association properties. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not to any of the classifiers.

Both association classes and N-ary associations are mapped to a class which is the domain of properties derived from each of its ends.

16.3.6.1 Association Class
top relation AssociationClassToOWLClass
{
  checkonly domain uml asc : AssociationClass{
    enforce domain owl cl : OWLClass{
      where {
        UClassToOClass(asc, cl);
      }
  }
} // AssociationClassToOWLClass
top relation AssociationClassToOWLProps
// Generates the properties coming from the member ends.
{
    checkonly domain uml asc : AssociationClass{memberEnd = uprop : Property{type = tp}};
    enforce domain owl oprop : OWLObjectProperty{RDFSdomain = cl : OWLClass,
        RDFSrange = rancl : OWLClass};
    when {
        AssociationClassToOWLClass(asc, cl);
        UClassToOClass(tp, rancl);
        EndPropertyToProperty(uprop, oprop);
    }
} // AssociationClassToOWLProps

top relation OwnedEndAttributeToObjectProperty
// maps ownedEnd property where property's type is a class to an object property
{
    checkonly domain uml asc : AssociationClass{ownedEnd = uprop : Property{type = tp}};
    enforce domain owl op : OWLObjectProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran : OWLClass};
    when {
        AssociationClassToOWLClass(asc, dom);
        UClassToOClass(tp, ran);
    }
    where {
        PropertyToProperty(uprop, op);
    }
} // OwnedEndAttributeToObjectProperty

top relation OwnedEndAttributeToDatatypeProperty
// maps ownedAttribute where property's type is a primitive type to datatype property
{
    checkonly domain uml asc : AssociationClass{ownedEnd = uprop : Property{type = tp : PrimitiveType}};
    enforce domain owl op : OWLDatatypeProperty{RDFSdomain = dom : OWLClass, RDFSrange = ran : Literal};
    when {
        AssociationClassToOWLClass(asc, dom);
        UMLPrimTypeToLiteral(tp, ran);
    }
    where {
        PropertyToProperty(prop, op);
    }
} // OwnedEndAttributeToDatatypeProperty

16.3.6.2 N-ary Association

top relation NaryAssociationToOWLClass
// Maps n-ary association where n > 2 to an OWL class
{
    checkonly domain uml asc : Association{};
    enforce domain owl cl : OWLClass{};
    when {
        asc.memberEnd->size() > 2; // only associations with more than two ends
    }
    where {
        UClassToOClass(asc, cl);
    }
} // NaryAssociationToOWLClass

top relation NaryAssociationToOWLProps
// Generates the properties coming from the member ends.
{
    checkonly domain uml asc : Association{memberEnd = uprop : Property{type = tp}};
    enforce domain owl oprop : OWLObjectProperty{RDFSdomain = cl : OWLClass,
        RDFSrange = rancl : OWLClass};
    when {
        NaryAssociationToOWLClass(asc, cl);
        UClassToOClass(tp, rancl);
        EndPropertyToProperty(uprop, oprop);
    }
} // NaryAssociationToOWLProps

16.3.7 Multiplicity

In UML, property is a MultiplicityElement, which defines upperValue and lowerValue to express cardinality. However, OWL uses Restrictions to represent Cardinality. So in addition to map Class to OWLClass, some OWLRestrictions will be generated based on multiplicity definitions of the ownedProperties and corresponding RDFSsubClassOf relationships between OWLClass and OWLRestriction will also be created.
The relation mapping multiplicity can be a top relation, since having PropertyToProperty in the when clause delays its execution until the RDFSdomain = structure in the enforce clause already exists. It is possible that there are several domains for the property. This relation will force them all to be subclasses of the restriction class.

```
// Upper multiplicity
{
    m, n : string;
    checkonly domain uml p : Property{upperValue = :ValueSpecification{value = m}};
    enforce domain owl owlp : Property{propertyRestriction = rest:MaxCardinalityRestriction{
        OWLmaxCardinality = :TypedLiteral{lexicalForm = m, datatypeURI = :URIRef{
            uri = :UniformResourceIdentifier{name = "xsd:integer"}}},
        RDFSdomain = :OWLClass{RDFSSubClassOf = rest}};
    when {
        not m = "*";  // Excludes infinity, which is no restriction
        PropertyToProperty(p, owlp);
    }
    where {
        BlankNodeID(rest);
    }
} // UpperMultToMaxCard
```

```
// Make sure Blank node has a nodeID
{
    n : string;
    checkonly domain owl rest : OWLRestriction{};
    enforce domain owl rest : OWLRestriction{nodeID = n};
    where {
        if rest.nodeID->isEmpty n = genAnonNodeID();  // Generates a unique blank node identifier
    }
} // BlankNodeID
```

```
// Lower multiplicity
{
    m, n : string;
    checkonly domain uml p : Property{lowerValue = :ValueSpecification{value = m}};
    enforce domain owl owlp : Property{propertyRestriction = rest:MinCardinalityRestriction{
        OWLminCardinality = :TypedLiteral{lexicalForm = m, datatypeURI = :URIRef{
            uri = :UniformResourceIdentifier{name = "xsd:integer"}}},
        RDFSdomain = :OWLClass{RDFSSubClassOf = rest}};
    when {
        not m = 0;  // Excludes zero, which is no restriction
        PropertyToProperty(p, owlp);
    }
    where {
        BlankNodeID(rest);
    }
} // LowerMultToMinCard
```

### 16.3.8 Association Generalization

Several kinds of generalizations of properties and associations in UML are mapped to subproperty and subclass relationships in OWL:

- subsetted properties to subPropertyOf
- properties at member ends of a generalized association to subPropertyOf
- generalized n-ary associations or association classes to subClassOf

// Map a subsetted property to subPropertyOf
{
  checkonly domain uml prop:Property{subSettedProperty = superprop : Property};
  enforce domain owl oprop : Property{RDFSsubPropertyOf = osuperprop : OWLProperty};
  when {
    PropertyToProperty(prop, oprop);
    PropertyToProperty(superprop, osuperprop);
  }
} // SubsetsPropertyToSubproperty

// Maps each member end of a generalized association to a subproperty
// Steps through the member ends by successive instantiations of the set comprehension pattern
{
  p1, p2 : OWL::....::OWLBase::Property;
  checkonly domain uml assn : Association{general = superassn : Association{
    memberEnd = supSeq : Sequence(Property){usuper | true}},
    memberEnd = subSeq : Sequence(Property){uprop | true}}
  {supSeq->indexOf(usuper) = subSeq ->indexOf(uprop)}; // steps through both sequences in tandem
  enforce domain owl oprop:Property{RDFSsubPropertyOf = osuper : Property};
  when {
    (AssociationToPropertyPair(assn, p1);  AssociationToPropertyPair(superassn, p2))OR
    (SymAssociationToSymProperty(assn, p1); SymAssociationToSymProperty(superassn, p2))OR
    (NaryAssociationToOWLProps(assn, p1); NaryAssociationToOWLProps(superassn, p2))OR
    (AssociationClassToOWLProps(assn, p1); AssociationClassToOWLProps(superassn, p2));
    // Makes sure associations are both mapped
    EndPropertyToProperty(uprop, oprop); // Corresponding ends in super and sub
    EndPropertyToProperty(usuper, osuper); // Extracts mapping of end properties
  }
} // AssocGeneralToSubProp

// Creates subclass relationship for mapped n-ary associations
{
  checkonly domain uml assn : Association{general = superassn : Association};
  enforce domain owl oclass : OWLClass{RDFSsubClassOf = osuper : OWLClass};
  when {
    NaryAssociationToOWLClass(assn, oclass);
    NaryAssociationToOWLClass(superassn, osuper);
  }
} // GeneralizesNaryToSubclass

// Creates subclass relationship for mapped association class generalization
{
  checkonly domain uml assn : AssociationClass{general = superassn : AssociationClass};
  enforce domain owl oclass : OWLClass{RDFSsubClassOf = osuper : OWLClass};
  when {
    AssociationClassToOWLClass(assn, oclass);
    AssociationClassToOWLClass(superassn, osuper);
  }
} // GeneralizesAssocClassToSubclass
16.3.9 Enumeration

An enumeration in UML is a designated collection of literals, so corresponds to an enumerated datatype in OWL.

top relation EnumerationToEnumeratedDatatype
//Created an enumerated datatype from an enumeration
{
  checkonly domain uml enum:Enumeration(ownedLiteral = ul :EnumerationLiteral);
  enforce domain owl edt:OWLDataRange{OWLoneOf = ol:RDFSLiteral};
  where {
    UMLLiteralToOWLLiteral(ul, ol); // not supplied
    UClassToOClass(enum, edt);
  }
} // EnumerationToEnumeratedDatatype

16.3.10 Powertypes

If a generalization set is covering, the general classifier is the union of the specific classifiers. If a generalization set is disjoint, then the specific classifiers are pairwise disjoint. OWL does not support the equivalent for properties, so generalization sets involving Associations are not mapped.

top relation IsCoveringToUnion
// Covering generalization set of classes goes to union
{
  checkonly domain uml genset :GeneralizationSet{isCovering = true,
    powertype = super :Class,
    generalization = :Generalization{specific = speccl :Class}};
  enforce domain owl ucl:UnionClass{OWLunionOf = osc :OWLClass};
  when {
    UClassToOClass(super, ucl);
    UClassToOClass(speccl, osc);
  }
} // IsCoveringToUnion

top relation IsDisjointToDisjoint
// Disjoint generalization to disjoint subclasses. Will generate sufficient pairs to make the mappings all pairwise disjoint
{
  checkonly domain uml genset :GeneralizationSet{isDisjoint = true,
    powertype = super :Class,
    generalization = :Generalization{specific = scl1 :Class},
    generalization = :Generalization{specific = scl2 :Class}};
  // Succeeds if there are more than one specific class
  enforce domain owl cl1 :OWLClass{OWLdisjointWith = cl2 :OWLClass};
  when {
    scl1.name < scl2.name;
    UClassToOClass(scl1, cl1);
    UClassToOClass(scl2, cl2);
  }
} // IsDisjointToDisjoint

} // transformation UMLToOWL
16.4 OWL to UML

16.4.1 Problematic Features of OWL

16.4.1.1 Mapping for Individuals

In the profile, Individual is represented as a singleton class. This works well in a profile, because a profile is a tool of the OWL ontology creator. The ontology creator would only model individuals in a profile if there were some special reason to. They could, and probably would, choose not to represent the vast bulk of individuals in this way.

The mappings on the other hand have to treat all individuals uniformly. It would be possible to map individuals to singleton classes, but then the mapping would have to deal with the RDFtype association. If an object is modeled as a singleton class, then the subclass relationship is equivalent to the instance relationship, so it would be necessary to map the RDFtype associations to subclass relationships in UML. This is probably not at all what one would generally want. In terms of Gruber's ontology quality measures, this is enormous encoding bias.

In these informative mappings, mapping Individual is not specified.

16.4.1.2 Mapping for Enumerated Classes

Enumerated classes are represented in OWL as owl:oneOf class restrictions, where the enumeration is either a data range (literals) or a set of individuals. Since individuals are not mapped, only the data range version of oneOf class restriction is mapped.

16.4.1.3 Mapping for complementOf and disjointWith

UML has constructions corresponding to the OWL complementOf and disjointWith constructions in the PowerSets package. These have been mapped in the UML to OWL mappings. However, in OWL these constructs are pairwise rather than applying to an entire generalization set. Mapping pairwise restrictions like this is complicated, leading to a difficult-to-read UML model. In these informative mappings, mappings for OWLcomplementOf and OWLdisjointWith are not specified.

16.4.1.4 Multiple Domains or Ranges for Properties

It is possible for a property to have multiple classes specified as either the domain or range of a property. In this case, OWL specifies that the domain or range is the intersection of all. The mappings specified here take this into account.

A problem is that multiple instances of domain or range specifications can come from a number of sources.

- Domain of the inverse of a property is range of the property, the range of the inverse is domain of a property
- Domain and range can be specified on superproperties of a property
- Domain and range can be derived from other equivalent properties
- Combinations of these

The mappings assume that the OWL model instance mapped includes the deductive closure of all domain and range specifications.

16.4.2 Transformation Header

transformation OntoUMLSource (owl:OWLMetamodel, uml:UMLMetamodel) {

key OWLOntology(uriRef);
key OWLUniverse (uriRef, ontology);
key OWLUniverse (nodeID, ontology); // anonymous classes
// All objects in an OWL model instance are instances of OWLUniverse. Figure 80.
// Objects in UML have names relative to other constructs, ultimately to Package
key Package(name);
key Class(name, owningPackage);
key Association(name, owningPackage);
key UML::...::Kernel::Property(name, class);
key UML::...::Kernel::Property(name, association);

16.4.3 Packaging Construct: OWLOntology

16.4.3.1 Ontology to Package
top relation OntoToPackage //Map ontology to Package.
{
    checkonly domain owl ont:OWLOntology { };
    enforce domain uml pack:Package { };  
    where {
        TopOntoToPackage(ont, pack);
        IntOntoToPackage(ont, pack);
    }
} // OntoToPackage

relation TopOntoToPackage  //Map top ontology to Package (Figure 75 ).
{
    n, un : string;
    checkonly domain owl ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = n}};
    enforce domain uml pack:Package {name = un};
    when {
        ont.importingOntology->isEmpty;
    }
    where {
        URIRefToName(n, un);
    }
} //TopOntoToPackage

relation IntOntoToPackage  //Map imported ontology to Package (Figure 75 ).
{
    n, un : string;
    checkonly domain owl ont:OWLOntology {uriRef = :UniformResourceIdentifier{name = n},
          importingOntology = onta : OWLOntology};
    enforce domain uml pack:Package {name = un,
          _packageImport = :PackageImport{importingNamespace = packa:Package}};
16.4.3.2 Ontology Properties to Comments

The Package construct in UML can not be the source of properties. The only related structure is the facility to attach comments inherited from NamedElement. Other than the mapping between importingOntology and packageImport, ontology properties are therefore mapped to comments. Note that OWLversionInfo is not an Ontology property, but an annotation property.

\[
\text{Top relation PriorVersionInfoToComment}
\]

\[
\text{checkonly domain owl ont:OWLOntology(OWLpriorVersion = :RDFS literal\{lexicalForm = v\});}
\]

\[
\text{enforce domain uml pack:Package\{ownedComment = :Comment\{body = (\"Prior Version \" + v)\};}
\]

\[
\text{when \{} 
\text{OntoToPackage(ont, pack);}
\]

\[
\text{\}} // PriorVersionInfoToComment
\]

\[
\text{Top relation IncompatibleWithToComment}
\]

\[
\text{checkonly domain owl ont:OWLOntology(OWLincompatibleWith = :OWLOntology\{uriRef = :UniformResourceIdentifier\{name = n\}\});}
\]

\[
\text{enforce domain uml pack:Package\{ownedComment = :Comment\{body = (\"Incompatible With \" + n)\};}
\]

\[
\text{when \{} 
\text{OntoToPackage(ont, pack);}
\]

\[
\text{\}} // IncompatibleWithToComment
\]

\[
\text{Top relation BackwardsCompatibleWithToComment}
\]

\[
\text{checkonly domain owl ont:OWLOntology(OWLbackwardsCompatibleWith = :OWLOntology\{uriRef = :UniformResourceIdentifier\{name = n\}\});}
\]

\[
\text{enforce domain uml pack:Package\{ownedComment = :Comment\{body = (\"Backwards Compatible With \" + n)\};}
\]

\[
\text{when \{} 
\text{OntoToPackage(ont, pack);}
\]

\[
\text{\}} // BackwardsCompatibleWithToComment
\]

16.4.4 Classes

16.4.4.1 OWL Class to UML Class

Classes in OWL are identified by uri, except for restriction classes which are blank nodes. A class in UML is identified by name within package.
16.4.4.2 Class Identified by URI
relation UClassToClass
// map an OWL class identified by uri to a topic.
{
    identifier, un : string;
    checkonly domain owl cl:OWLclass {uriRef = :UniformResourceIdentifier(name = identifier)};
    // Note that an instance of an anonymous class fails to have a uri, so is excluded
    enforce domain uml ucl : Class {name = un};
    where {
        URIRefToName(identifier, un);
    }
} // UClassToClass

16.4.4.3 Anonymous Class to Class
An anonymous class in OWL is a blank node is identified by a nodeID. This is unique within an ontology, if not persistent. This identifier will serve as a name for the corresponding UML class, where the name is persistent.

relation AnonClassToClass
// map an anonymous OWL class to a UML class.
{
    ID : string;
    checkonly domain owl aclass:OWLClass {nodeID = ID};
    enforce domain uml ucl : Class {name = ID};
    when{
        aclass.uriRef->isEmpty; // Some classes have also uri refs. These classes are not anonymous.
    }
} // AnonClassToClass

16.4.5 Hierarchy

16.4.5.1 Subclass, Equivalent Class
top relation SubclassToGeneralization
// map the RDFSsubclassOf meta-association to a UML subclass/superclass relationship (Figure 76).
{
    checkonly domain owl subcl:OWLClass {RDFSsubClassOf = supercl : OWLClass};
    enforce domain uml usubcl : Class {superClass = usuper : Class};
    when{
        OClassToUClass(subcl, usubcl);
        OClassToUClass(supercl, usuper);
    }
} // SubclassToGeneralization

top relation EquivalentClassToMutualGeneralization
// map equivalent classes to a pair of UML subclass/superclass relationships (Figure 76).
{
    checkonly domain owl class1:OWLClass {OWLEquivalentClass = class2 : OWLClass};
    enforce domain uml uclass1 : Class {superClass = class2 : Class{superClass = class1}};
}
16.4.5.2 Universal Superclass

OWL has a universal superclass called owl:Thing acting as a default domain and range for properties. Need to create a comparable UML class, called Thing *** See 14.2.5 ***

```
top relation UniversalSuperclass
{
    owcl : OWLClass;
    checkonly domain owl ont : OWLOntology{};
    enforce domain uml thing : Class(name = "Thing", owningPackage = pack : Package,
        _class = ucl : Class); // _class is opposite metaproperty to superClass
    when {
        OntoToPackage(ont, pack);
        owcl = ont.owlUniverse; // will instantiate for each OWL object which is an OWLClass
        ucl = pack.ownedMember; // will instantiate for each ownedMember which is a Class
        SubclassToGeneralization(owcl, ucl); // forces wait until subclasses structure has been generated
        ucl.superClass->isEmpty; // selects only those classes without superclasses
    }
} // UniversalSuperclass
```

16.4.6 Constructed Classes

OWL allows classes to be constructed by union, intersection, and difference. OWL also allows classes to be declared disjoint. Union and intersection can be mapped to subclass relationships, while disjoint and difference are not mapped.

```
top relation IntersectionToUML
// OWL intersection to subclass relationships
// Will generate an instance of superClass for each instance of OWLintersectionOf
{
    checkonly domain owl interclass : IntersectionClass(OWLintersectionOf = oclass : OWLClass,
        ontology = ont : OWLOntology);
    enforce domain uml usub:Class(superClass = uclass : Class, owningPackage = pack : Package);
    when {
        OntoToPackage(ont, pack);
        owcl = ont.owlUniverse; // will instantiate for each OWL object which is an OWLClass
        ucl = pack.ownedMember; // will instantiate for each ownedMember which is a Class
        SubclassToGeneralization(owcl, ucl); // forces wait until subclasses structure has been generated
        ucl.superClass->isEmpty; // selects only those classes without superclasses
    }
} // IntersectionToUML
```

```
top relation UnionToUML
// OWL union to subclass relationships
// Will generate an instance of _Class for each instance of OWLunionOf
{
    checkonly domain owl unclass : UnionClass(OWLunionOf = oclass : OWLClass,
        ontology = ont : OWLOntology);
    enforce domain uml usuper:Class(_Class = uclass : Class, owningPackage = pack : Package);
    when {
        OntoToPackage(ont, pack);
        OClassToUClass(unclass, usuper);
        OClassToUClass(oclass, uclass);
    }
```
16.4.7 Data Range

A data range in OWL is either a literal type or an enumeration of literals. The mapping of literal types is application specific, but the enumeration corresponds to the enumeration in UML.

```
top relation EnumerationToEnumeration
{
    checkonly domain owl edt:OWLDataRange{OWLoneOf = ol:RDFSLiteral{lexicalForm = v}};
    enforce domain uml enum:Enumeration{ownedLiteral = ul : EnumerationLiteral};
    where {
        OWLLiteralToUMLLiteral(ol, ul); // Not supplied
        OClassToUClass(edt, enum);
    }
} // EnumerationToEnumeration
```

16.4.8 Range Restriction Restriction Classes

The restriction classes allValuesFrom, someValuesFrom and HasValue all define subclasses of the domain on which a specified property behaves in a specified way. UML does not have the machinery to represent the specified behavior. So the mapping in each case is to an anonymous class, declared as a subclass of the domain of the property (if any), with the restriction indicated in an attached comment. The mapping for hasValue only includes the case where the value is a literal, since there is not a good general representation of Individuals in UML.

```
top relation AllValuesFromToClass
{
    cn, pn : string;
    up : UML::Kernel::Property;
    checkonly domain owl avr : AllValuesFromRestriction{OWLallValuesFrom = oc : OWLClass,
        OWLonProperty = op : RDFProperty};
    enforce domain uml rcl : Class{comment = ("AllValuesFrom " + cn + " on " + pn)};
    when {
        OClassToUclass(oc, rcl);
        OWLPropToUMLProp(op, up);
    }
    where {
        cn = rcl.name;
        pn = up.name;
        SubclassOfPropDomain(op, rcl);
    }
} // AllValuesFromToClass
```

```
relation SubclassOfPropDomain
// Makes UML mapping of restriction class subclas of mapping of property domain, if any
{
    checkonly domain owl op:Property{RDFSdomain = oc : OWLClass};
    enforce domain uml uc:Class{superClass = uc};
    when {
        OClassToUclass(oc, uc);
    }
} // SubclassOfPropDomain
```

```
top relation SomeValuesFromToClass
{
    cn, pn : string;
    up : UML::Kernel::Property;
    checkonly domain owl svr : SomeValuesFromRestriction{OWLsomeValuesFrom = oc : OWLClass,
```
Properties in OWL are similar in concept to properties and associations in UML, but quite different in detail. The mappings follow as closely as possible the profiles given in 14.2.6.

An OWL property will be mapped to a UML property which is an ownedAttribute of a Class in case:

- it is a datatype property
- it is an object property with no inverse and is not inverse functional.

An OWL property will be mapped to a UML binary association in case

- it is an object property with an inverse (including symmetric property). In this case the property and its inverse will be mapped to the two ends of the association.
- It is inverse functional. An inverse functional property generates a partition of its range. Even if it has no inverse, it must be mapped to an association because the corresponding multiplicity must apply to the end opposite the end corresponding to the inverse functional property.

Cardinality constraints will be mapped to multiplicities. This includes functional and inverse functional properties.

A property can have possibly several classes declared as its domain, and possibly several declared as its range. In either case the result is:

- no classes declared - owl:Thing
- one class declared - the class
more than one class declared - the intersection of the declared classes

16.4.9.1 Property to Owned Attribute

top relation DTPropToAttribute
{
    identifier, un : string;
    checkonly domain owl dtp:OWLDatatypeProperty {uriRef = :UniformResourceIdentifier(name = identifier),
        RDFSRange = :TypedLiteral {datatypeURI = dt},
        ontology = ont:OWLOntology};
    enforce domain uml prop : Property{name =un, owningPackage = pack : Package, type = tp : PrimitiveType};
    when {
        OntoToPackage(ont, pack);
        LiteralToPrimitiveType(dt ,tp); // relates RDF literal types to UML primitive types
    }
    where {
        URIRefToName(identifier, un);
    }
} //DTPropToAttribute

top relation ObjPropToAttribute
{
    identifier, un : string;
    checkonly domain owl op:OWLObjectProperty {uriRef = :UniformResourceIdentifier(name = identifier),
        ontology = ont:OWLOntology};
    enforce domain uml prop : Property{name =un, owningPackage = pack : Package};
    when {
        OntoToPackage(ont, pack);
        op.inverseProperty->isEmpty; // no inverse
        op.OWLinverseOf->isEmpty;
        not op.oclIsTypeOf(SymmetricProperty); // not its own inverse
        not op.oclIsTypeOf(InverseFunctional); // not inverse functional
    }
    where {
        URIRefToName(identifier, un);
    }
} //ObjPropToAttribute

top relation AddAttributeClass
// Property and domain already mapped. Add Class.
{
    checkonly domain owl prop:Property();
    enforce domain uml upr:Property{class = cl : Class};
    when {
        ObjPropToAttribute(prop, upr) OR DTPropToAttribute(prop, upr);
        PropDomain(prop, cl);
    }
} //AddAttributeClass

top relation AddAttributeType;
// property and range already mapped. Add type.
{
    checkonly domain owl prop:Property();
    enforce domain uml upr:Property{type = cl};
    when {
        ObjPropToAttribute(prop, upr);
        PropRange(prop, cl);
    }
}
16.4.9.2 Property to Association

top relation PropertyPairToAssociation

// Property and its inverse go to an association whose name is the concatenation of the property names.
{
    assocID : string;
    pack1 : Package;
    checkonly domain owl prop:Property{ontology = ont:OWLOntology,
        OWLInverseOf = invp : Property{ontology = invont : OWLOntology}};
    enforce domain uml assn : Association{memberEnd = ps : Sequence(Property) {p1, p2},
        name = assocID, owningPackage = pack : Package};
    when {
        OntoToPackage(ont, pack); // association will be in this package
        OntoToPackage(invont, pack1); // even though the inverse might be in another ontology
        invp.equivalentProperty->isEmpty; // no equivalent properties
        invp.OWLEquivalentProperty->isEmpty;
    }
    where {
        PropertyToAProperty(prop, p1);
        PropertyToAProperty(invp, p2);
        assocID = p1.name + p2.name; // Association name is concatenation of property names
    }
}

} // PropertyPairToAssociation

top relation SymmetricPropToAssociation

// Symmetric property goes to an association both of whose member ends are the same property
{
    assocID : string;
    checkonly domain owl prop:SymmetricProperty{ontology = ont:OWLOntology};
    enforce domain uml assn : Association{memberEnd = ps : Sequence(Property) {p1, p1},
        name = assocID, owningPackage = pack : Package};
    when {
        OntoToPackage(ont, pack); // association will be in this package
    }
    where {
        PropertyToAProperty(prop, p1);
        assocID = p1.name;
    }
}

} // SymmetricPropToAssociation

top relation InverseFunctToAssociation

// Inverse functional property with no inverse go to an association
{
    checkonly domain owl prop:InverseFunctionalProperty{ontology = ont:OWLOntology};
    enforce domain uml assn : Association{memberEnd = ps : Sequence(Property){p1, p2},
        name = assocID, owningPackage = pack : Package};
    when {
        prop.OWLInverseOf->isEmpty; // The inverse functional property has no inverse declared
        prop.inverseProperty->isEmpty;
        OntoToPackage(ont, pack); // association will be in this package
    }
    where {
        PropertyToAProperty(prop, p1);
        PropertyToOppProperty(prop, p2);
        assocID = p1.name;
    }
}
16.4.10 Domains, Ranges and Property Types

16.4.10.1 Domains

top relation PropDomain
{

}
checkonly domain owl prop:Property {}; enfore domain uml cl : Class{};
where {
    DefaultDomain(prop, cl);
    SingleDomain(prop, cl);
    MultDomain(prop, cl);
}
} // PropDomain

relation DefaultDomain
// Create default domain for property
{
    checkonly domain owl prop:Property(ontology = ont : OWLOntology);
    enforce domain uml usuper : Class{};
    when {
        prop.RDFSdomain->isEmpty; // no domains declared, so default
        UniversalSuperclass(ont, usuper); // usuper has already been created for this ontology
    }
} // DefaultDomain

relation SingleDomain
// Find single domain for property
{
    checkonly domain owl prop:Property{RDFSdomain = dom: OWLClass, ontology = ont : OWLOntology};
    enforce domain uml domcl : Class{};
    when {
        prop.RDFSdomain->size() = 1; // only one domain declared
        OClassToUClass(dom, domcl);
    }
} // SingleDomain

relation MultDomain
// Find intersection of multiple domains for property as subclass called "DomainIntersection_" + name of property
// Assumed that the collection of domains of a property is the deductive closure of all sources of domains, in particular
// a range of an inverseOf property
{
    din : string;
    obj : NamedElement;
    checkonly domain owl prop:Property{RDFSdomain = dom: OWLClass, ontology = ont : OWLOntology};
    enforce domain uml domintcl : Class{name = rin, superClass = domcl: Class, owningPackage = pack :Package};
    when {
        prop.RDFSdomain->size() > 1; // if more than one domain declared
        OClassToUClass(dom, domcl); // will be instantiated once for each class
        OWLPropToUMLObj(prop, obj); // Need the property to be created so can get its name
        OntoToPackage(ont, pack);
    }
    where {
        din = "DomainIntersection_" + obj.name;
    }
} // MultDomain

16.4.10.2Ranges

top relation PropRange
{
    checkonly domain owl prop:Property {};
    enforce domain uml cl : Class{};
}

Ontology Definition Metamodel
where {
    DefaultRange(prop, cl);
    SingleRange(prop, cl);
    MultRange(prop, cl);
}
} // PropRange

relation DefaultRange
// Create default range for property
{
    checkonly domain owl prop:Property{ontology = ont : OWLOntology};
    enforce domain uml usuper : Class{};
    when {
        prop.RDFSrange->isEmpty; // no ranges declared, so default
        UniversalSuperclass(ont, usuper); // usuper has already been created for this ontology
    }
} // DefaultRange

relation SingleRange
// Find single range for property
{
    checkonly domain owl prop:Property{RDFSrange = ran: OWLClass, ontology = ont : OWLOntology};
    enforce domain uml rancl : Class{};
    when {
        prop.RDFSrange->size() = 1; // only one range declared
        OClassToUClass(ran, rancl);
    }
} // SingleRange

relation MultRange
// Find intersection of multiple ranges for property as subclass called “RangeIntersection_” + name of property
// Assumed that the collection of ranges of a property is the deductive closure of all sources of ranges, in particular a
domain of an inverseOf property
{
    rin : string;
    obj : NamedElement;
    checkonly domain owl prop:Property{RDFSrange = ran: OWLClass, ontology = ont : OWLOntology};
    enforce domain uml ranintcl : Class{ name = rin, superClass = rancl: Class,
        owningPackage = pack : Package};
    when {
        prop.RDFSrange->size() > 1; // if more than one range declared
        OClassToUClass(ran, rancl); // will be instantiated once for each class
        OWLPropToUMLObj(prop, obj); // Need the property to be created so can get its name
        OntoToPackage(ont, pack);
    }
    where {
        rin = “RangeIntersection_” + obj.name;
    }
} // MultRange

16.4.11 Cardinalities and Multiplicities

Top relation CardinalityToMultiplicity
{
    checkonly domain owl oprop : Property{};
    enforce domain uml uprop : Property{};
    when {
        OWLPropToUMLProp(oprop, uprop); // Delays until properties have been created
}
where {
    if oprop.oclIsTypeOf(FunctionalProperty) {
        FunctionalToUMult(oprop, uprop);
        FunctionalInverseFunToULMult(oprop, uprop);
    } else if oprop.oclIsTypeOf(InverseFunctionalProperty) {
        InvFunctionalToUMult(oprop, uprop);
    } else {
        CardToULMult(oprop, uprop);
        MaxCardToUMult(oprop, uprop);
        MinCardToLMult(oprop, uprop);
    }
    AddUnlimitedMax(uprop);
    AddZeroMin(uprop);
} // CardinalityToMultiplicity

relation CardToULMult
// max and min cardinality are the same
{
    c : string;
    checkonly domain owl oprop : Property{
        propertyRestriction = :CardinalityRestriction{OWLCardinality = :TypedLiteral{lexicalForm = c}};
        enforce domain uml uprop : Property{upperValue = c, lowerValue = c};
    } //MaxMinCardToULMult

relation MaxCardToUMult
// max cardinality
{
    u : string;
    checkonly domain owl oprop : Property{
        propertyRestriction = :MaxCardinalityRestriction{OWLmaxCardinality = :TypedLiteral{lexicalForm = u}};
        enforce domain uml uprop : Property{upperValue = u};
    } //MaxCardToUMult

relation MinCardToLMult
// min cardinality
{
    l : string;
    checkonly domain owl oprop : Property{
        propertyRestriction = :MinCardinalityRestriction{OWLminCardinality = :TypedLiteral{lexicalForm = l}};
        enforce domain uml uprop : Property{lowerValue = l};
    } //MinCardToLMult

relation FunctionalToUMult
// Functional property
{
    checkonly domain owl oprop : FunctionalProperty{};
    enforce domain uml uprop : Property{upperValue = "1"};
} //FunctionalToUMult

relation InvFunctionalToUMult
// If a inverse functional property has an inverse, the multiplicity goes on the inverse
{
    checkonly domain owl ifprop : InverseFunctionalProperty{};
    enforce domain uml opprop : Property{upperValue = '1'};
    when {
        ifprop.OWLinverseOf->exists(invprop : Property | PropertyToAProperty(invprop, opprop))
    }
}
if prop.inverseProperty->exists(invprop : Property | PropertyToAProperty(invprop, opprop))

} // InvFunctionalToUMult

relation AddUnlimitedMax
// Add unlimited max cardinalities to UML properties with no max cardinality
{
    enforce domain uml prop:Property{upperValue = "*"};
    when {
        prop.upperValue->isEmpty;
    }
} // AddUnlimitedMax

relation AddZeroMin
// Add zero min cardinalities to UML properties with no min cardinality
{
    enforce domain uml prop:Property{lowerValue = "0"};
    when {
        prop.lowerValue->isEmpty;
    }
} // AddZeroMin

16.4.12 Subproperty, Equivalent Property

top relation SubpropertyToGeneralization
// a property generalizes by subsetting
{
    checkonly domain owl prop: Property(RDFSsubPropertyOf = superprop : Property);
    enforce domain uml uprop{subscribedProperty = superuprop : Property};
    when {
        OWLPropToUMLObj(prop, uprop);
        OWLPropToUMLObj(superprop, superuprop);
    }
} //SubpropertyToGeneralization

top relation EquivPropertyToGeneralizations
// a Equivalent properties by mutual subsetting
{
    checkonly domain owl prop: Property(OWLEquivalentProperty = equivprop : Property);
    enforce domain uml uprop : Property{subscribedProperty = equivuprop : Property{
        subscribedProperty = uprop});
    when {
        OWLPropToUMLObj(prop, uprop);
        OWLPropToUMLObj(equivprop, equivuprop);
    }
} //EquivPropertyToGeneralizations

16.4.13 Annotation Properties to Comments

OWL has an annotation property facility, including several built-in annotation properties, whose domain is OWLUniverse, while UML has only comments on NamedElement. Note that OWLversionInfo is an annotation property, not an ontology property. Note also that since the mapping doesn’t map Individuals to UML, annotation properties on individuals are not mapped.

The built-in annotation properties are modeled as meta-associations, while the user-defined annotation properties are instances of OWL.AnnotationProperty.

top relation VersionInfoToComment
{  
  v : string;
  checkonly domain owl res :OWLUniverse(OWLversionInfo = :RDFSLiteral(lexicalForm = v));
  enforce domain uml ne :NamedElement(ownedComment = :Comment(body = ("Version " + v)));
  when {
    UniverseToNamedElement(res, ne);
  }
} // VersionInfoToComment

relation UniverseToNamedElement
{
  checkonly domain owl res :OWLUniverse();
  enforce domain uml ne :NamedElement();
  when {
    OntoToPackate(res, ne) OR
    OClassToUClass(res, ne) OR
    OWLPropToUMLProp(res, ne);
  }
} // UniverseToNamedElement

function StringFromURIRef(uriref : URIReference) : string
{
  uriref.fragmentIdentifier.name->isEmpty then uriref.name
  else uriref.name + "#" + uriref.fragmentIdentifier.name;
} // StringFromURIRef

top relation RDFSCommentToComment
{
  com : string;
  checkonly domain owl res:OWLUniverse(RDFScomment = :RDFSLiteral(lexicalForm = com));
  enforce domain uml ne:NamedElement(comment = com);
  when {
    UniverseToNamedElement(res, ne);
  }
} // RDFSCommentToComment

top relation RDFSLabelToComment
{
  com : string;
  checkonly domain owl res:OWLUniverse(RDFSlabel = :RDFSLiteral(lexicalForm = com));
  enforce domain uml ne:NamedElement(comment = com);
  when {
    UniverseToNamedElement(res, ne);
  }
} // RDFSLabelToComment

top relation SeeAlsoToComment
{
  com : string;
  checkonly domain owl res:OWLUniverse(RDFSseeAlso = sr : RDFSResource);
  enforce domain uml ne:NamedElement(comment = com);
  when {
    UniverseToNamedElement(res, ne);
  }
  where {
    if sr->oclIsTypeOf(RDFSLiteral) then com = sr.lexicalForm
    else com = StringFromURIRef(sr.uriRef);
  }
} // SeeAlsoToComment

Ontology Definition Metamodel
16.5  OWL but not UML

16.5.1  Predicate Definition Language

OWL permits a subclass to be declared using subClassOf or to be inferred from the definition of a class in terms of other classes. It also permits a class to be defined as the set of individuals which satisfy a restriction expression. These expressions can be a boolean combination of other classes (intersectionOf, unionOf, complementOf), or property value restriction on properties (requirement that a given property have a certain value – hasValue). The property equivalentClass applied to restriction expressions can be used to define classes based on property restrictions.

For example, the class definition

```xml
<owl:Class rdf:ID="TexasThings">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#locatedIn" />
      <owl:allValuesFrom rdf:resource="#TexasRegion" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Defines the class TexasThings as a subclass of the domain of the property locatedIn. These individuals are precisely those for which the range of locatedIn is in the class TexasRegion. Given that we know an individual to be an instance of TexasThings, we can infer that it has the property locatedIn, and all of the values of locatedIn associated with it are instances of TexasRegion. Conversely, if we have an individual which has the property locatedIn and all of the values of locatedIn associated with that individual are in TexasRegion, we can infer that the individual is an instance of TexasThings.

Because it is possible to infer from the properties of an individual that it is a member of a given class, we can think of the complex classes and property restrictions as a sort of predicate definition language.

UML provides but does not mandate the predicate definition language OCL. Note that a subsumption reasoner could be built for UML. But because UML is strongly typed, it could work in the way mandated for OWL only if there were a universal superclass provided in the model library, which is rarely provided in practice.

OCL and CL (Common Logic) are two predicate definition languages which are relevant to the ODM. Both are more expressive than the complex class and property restriction expressions of OWL Full. There are also other predicate definition languages of varying expressive powers which particular applications might wish to use.

The ODM does not mandate any particular predicate definition language, but will provide a place for a package enabling the predicate definition language of choice for an application. In particular, the ODM includes a metamodel for CL.

16.5.2 Names

A common assumption in computing applications is that within a namespace the same name always refers to the same object, and that different names always refer to different objects (the unique name assumption). As a consequence, given a set of names, one can count the names and infer that the names refer to that number of objects.

Names in OWL do not by default satisfy the unique name assumption. The same name always refers to the same object, but a given object may be referred to by several different names. Therefore counting a set of names does not warrant the inference that the set refers to that number of objects. Names, however, are conceptually constants, not variables.

OWL provides features to discipline names. The unique name assumption can be declared to apply to a set of names (allDifferent). One name can be declared to refer to the same object as another (sameAs). One name can be declared to refer to something different from that referred to by any of a set of names (differentFrom).

Two classes can be stated to be equivalent (equivalentClass) and two properties can be stated to be equivalent (equivalentProperty). Equivalent classes have the same extents, equivalent properties link the same pairs.

UML supports named elements with namespaces only at the M1 level. Although a UML class may be defined to contain a definite collection of names, names at the M0 level are not prescribed. Applications modeled in UML are frequently implemented using systems like SQL which default the unique name assumption, but this is not mandated. UML places no constraints on names at the M0 level.

In particular, it is permitted for applications modeled in UML to be implemented at the M0 level using names which are existentially quantified variables. Note that the UML constraint language OCL uses variables. OWL does not support variables at all.
16.5.3 Other OWL Developments

There are a number of developments related to OWL which are not yet finalized, including SWRL Semantic Web Rule Language and OWL services. These are considered out of scope for the ODM. A translation of an out-of-scope model element will be to a comment in the OWL target.

16.6 In UML But Not OWL

16.6.1 Behavioral and Related Features

UML allows the specification of behavioral features, which declare capabilities or resources. One use of behavioral features is to calculate property values. Behavioral features can be used in the OCL that derives properties. Facilities of UML supporting programs include operations, which describe the parameters of methods; static operations, which are operations attached to a class like static attributes; interface classes, which specify among other things operation features; qualified associations, which are a special kind of ternary relation; and active classes, which are classes each instance of which controls its own thread of execution control.

ODM omits these features of UML.

16.6.2 Complex Objects

UML supports various flavors of the part-of relationship between classes. In general, a class (of parts) can have a part-of relationship with more than one class (of wholes). One flavor (composition) specifies that every instance of a given class (of parts) can be a part of at most one whole. Another (aggregation) specifies that instances of parts can be shared among instances of wholes.

Composite structures defined in classes specify runtime instances of classes collaborating according to connectors. They are used to hierarchically decompose a class into its internal structure which allows a complex objects to be broken down into parts. These diagrams extend the capabilities of class diagrams, which do not specify how internal parts are organized within a containing class and have no direct means of specifying how interfaces of internal parts interact with its environment.

Ports model how internal instances are to be organized. Ports define an interaction point between a class and its environment or a class and its contents. They allow you to group the required and provided interfaces into logical interactions that a component has with the outside world. Collaboration provides constructs for modelling roles played by connectors.

Although not strictly part of the complex object feature set, the feature template (parameterized class) is most useful where the parameterized class is complex. One could for example define a multimedia object class for movies, and use it as a template for a collection of classes of genres of movie, or a complex object giving the results of the instrumentation on a fusion reactor which would be a template for classes containing the results of experiments with different objectives.

Although it is recognized that there is a need for facilities to model mereotopological relationships in ontologies, and UML provides a capability in this space, there does not seem to be sufficient agreement on the scope and semantics of existing models for inclusion of specific mereotopological modelling features into the ODM at this stage.

16.6.3 Access Control

UML permits a property to be designated read-only. It also allows classes to have public and private elements.

ODM omits access control features.
### 16.6.4 Keywords

UML has **keywords** which are used to extend the functionality of the basic diagrams. They also reduce the amount of symbols to remember by replacing them with standard arrows and boxes and attaching a `<<keyword>>` between guillements. A common feature that uses this is `<<interfaces>>`

ODM omits this feature.

### 16.6.5 Profiles

UML has a facility called **Profile**, whereby a specialist developer can make lightweight extensions to the modeling language by defining **stereotypes**, which define subclasses of metaclasses. This enables the developer to either articulate the metaclass into a number of kinds or to rename the metaclass.

OWL DL does not have a facility like this. One can achieve the same effect in OWL Full by defining subclasses of `owl:Class` or `rdf:Property`, since OWL is its own metalanguage.

Profiling in UML is necessary because of the strict separation of metalevels, and is useful partly because it allows reuse of the UML graphical rendering conventions, and also the UML graphical editors and other tools. OWL does not at present have a standard graphical representation. Because OWL DL does not support an equivalent of stereotypes, and because the functional equivalent of stereotypes in OWL Full is a user capability rather than a metamodeler capability as in UML, the mappings from UML to OWL and from OWL to UML disregard this feature.