Conceptual Modelling in UML and OWL-2

Jesper Zedlitz and Norbert Luttenberger
Christian-Albrechts-Universität
Kiel, Germany
Email: {j.zedlitz, n.luttenberger}@email.uni-kiel.de

***Abstract*—Both OWL-2 and UML static class diagrams lend themselves very well for conceptual modelling of complex information systems. Both languages have their advantages. In order to benefit from the advantages and software tools of both languages, it is usually necessary to repeat the modelling process for each language. We have investigated whether and how conceptual models written in one language can be automatically transformed into models written in the other language. For this purpose we investigated differences and similarities of various model elements (such as element type, data types, relationship types) in static UML data models and OWL-2 ontologies. We provide a transformation for similar elements.**

***Keywords*—UML; OWL; conceptual modelling; model transformation; meta modelling**

I. Introduction

The Web Ontology Language (OWL) is mostly considered as a language for knowledge representation. However, it can also be used as a language for conceptual modelling of complex information systems. It can be used as a language to represent the entities of a certain domain, to express the meaning of various, usually ambiguous terms and to identify the relationships between them.

In this respect, OWL can be seen as a direct "competitor" to static class diagrams from Unified Modeling Language (UML). This kind of diagrams is often used for conceptual modelling, for example in the ISO 191xx series of standards. Both languages have their benefits. UML's visual syntax is easy to understand and there is a variety of software tools to choose from. OWL is backed up by formal logic and logical conclusions can be drawn on models using inference software (a.k.a. *reasoner*). In order to be benefited from the advantages and software tools of both languages, it is usually necessary to repeat the modelling process for each language. We have investigated whether and how conceptual models written in one language can be automatically transformed into models written in the other language.

In contrast to existing approaches, our transformation abstracts from the concrete syntax—in most cases a serialisation based on the Extensible Markup Language (XML)—and operates on the level of UML's and OWL's meta-models. This allows to show which model elements can be transformed and which cannot—independent of individual examples.

This article is the extended version of the paper published at SEMAPRO 2013 [1]. It is organized as follows. We start with a general overview of our approach. The following sections deal with certain kind of constituents: element types (Section III), data types (Section IV), relationship types (Section V), and constraints (Section VI). Each section describes how the particular kinds of model element is represented in UML and OWL and how it can be transformed from one language into the other. Section VII deals with the question of whether the transformation rules presented in the previous sections are correct and—within their previously specified limits—complete. Section VIII gives an overview of existing approaches for transformations between UML and OWL. In Section IX, we summarize our work and point out fields of future work.

II. Our Approach

In this section, we present the main ideas of a transformation of conceptual models on the meta model level by using a standardized declarative model transformation language.

In order to transform between OWL and UML models, it is necessary to find a "common third". This common third might be a transfer format, i.e., a common syntax. It might also be a common meta model, which permits a syntax independent transformation of model elements of one language into corresponding model elements of the other language.

A fundamental difference between OWL-2 and its predecessors is the fact that OWL-2 has a MOF-compliant meta model. OWL-2 ontologies can be processed not only as serialized XML documents, but also as MOF-based models. Therefore, all tools that are available for model transformations in the context of MOF can also be used to process ontologies.

A *model-based* transformation only operates on a syntactical level. In contrast, a *model* transformation offers access to both the models as well as the meta models [2, p. 28]. The transformation rules refer to the meta models only—hence the term "processing on meta model level". When writing the transformation rules, it is not necessary to know which models are going to be transformed later. Every model that is compliant with the (input) meta model can be processed using these transformation rules.

As a transformation language for MOF-based models, the Object Management Group (OMG) introduced the "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)". It is particularly well suited for our purposes. The fact that the name of the language contains "MOF" makes the close connection obvious. QVT includes two different language versions. For our purposes, the declarative QVT Relations (QVT-R) is more suitable:

- A declarative notation leads to compact transformation rules that require less code duplication.
- The developed rules, their interaction with each other- as well as the connection between the source and

the target model, may subsequently be analysed more effectively.

- During the execution of a QVT-R transformation so called *trace classes* are generated. These are useful for later analysis.
- QVT-R has a graphical syntax allowing a clear and easy to understand presentation of the transformation rules.

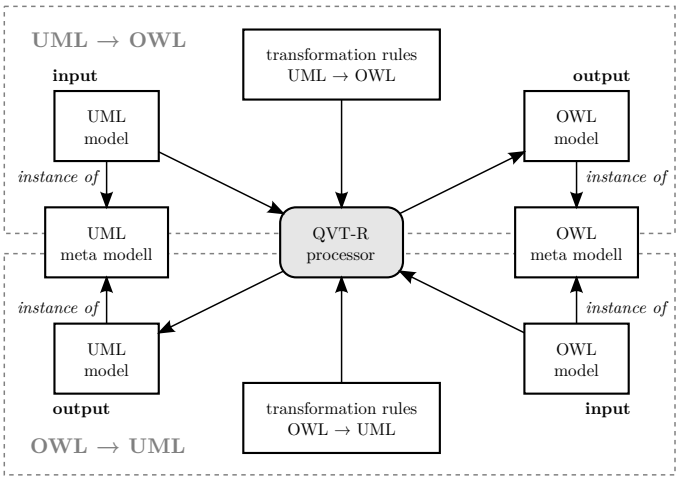


Fig. 1. Sketch of the transformation UML↔OWL. The upper part of the figure is read from left to right. The lower part is read from left to right.

Figure 1 sketches the complete transformation process for UML and OWL used in the context of this work. The upper half shows the UML → OWL transformation, the lower half the OWL → UML transformation. Common for both directions of transformation is the use of UML and OWL meta models. Before and after the actual model-to-model transformations, certain pre-transformations need to be done, especially when dealing with OWL ontologies. These pre-transformations map between concrete and abstract syntax. This can easily be done using the OWL API [3] or an XSLT script.

III. Element Types

Element types are among the most important components of a conceptual model: "Defining the entity types [...] is a crucial task in conceptual modeling" [4, p. 41]. Element types are also known as concepts and are eponymous for the term "conceptual model". Element types are used to group individual objects with the same or similar characteristics.

Both UML and OWL make an equal distinction between classes on the one side and "instances" resp. "individuals" on the other side. Because of this similarity, a transformation from UML classes into OWL classes is straight forward.

For the transformation from OWL to UML, one must distinguish between named—i.e, declared—classes on the one hand and unnamed class expressions (CE) on the other hand. A named class is the simplest case. In this case, a UML class is created with its name derived from the identifier of the class. Transformations of unnamed CE will be covered in later sections about generalization (ObjectUnionOf) and intersection (ObjectIntersectionOf).

A. Names

For an unambiguous identification, each element type must have a unique name within a model. The UML specification demands that every model element must have a unique name within its package. This ensures that each model element can be uniquely identified by specifying its name and the position in the package hierarchy. OWL uses Internationalized Resource Identifier (IRI) conforming to RFC 3987 to name element types as well as other elements of an ontology—instances of the OWL meta class `Entity`. All assigned names have global scope, regardless of the context in which they are used.

Since—unlike in UML—an OWL model element must have a unique name not only within a package but globally, such a globally unique name must be assigned during the transformation of any model element. Therefore, corresponding globally unique IRIs are generated during the transformation. For the transformation OWL → UML it is sensible to use the remaining characters of an IRI written in short form as the name of UML elements—assuming the prefix IRI is associated with the package. As a result the generated UML models will be easier to read. This is possible, because names in the UML must be unique only within a package.

B. Inheritance

A common situation in conceptual modelling is that the population of one element type is neccessarily also the population of a another element type. This is referred to as an inheritance relationship between these two element types.

Due to the very similar structure and semantics—especially the transitivity—of *Generalization* elements on the one hand and `SubClassOf` axioms on the other hand, a transformation from UML to OWL is easily possible. If both of the elements that are connected via an instance of the UML meta class *Generalization* can be transformed to OWL, the transformation of the *Generalization* instance is simple. An instance of the OWL meta class *SubClassOf* must be created during the transformation process. Since the newly created `SubClassOf` element is an axiom, it is necessary to connect it to the containing ontology.

For the transformation of an axiom of the form `SubClassOf(`C_c C_p`)` in the direction OWL → UML the following cases must be distinguished:

(a) Both sub-class C_c and super class C_p can be transformed into a UML class.

(b) At least one of the element types is a CE and the membership to that CE is described by necessary and sufficient conditions.

(c) Al least one of the element types cannot be transformed into a UML class due to other reasons (e.g. complementing).

In case (a) in which both element types can be mapped, a transformation is simple. Figure 2 shows an example for the transformation of a `SubClassOf` axiom in this simplest case. An instance of the UML meta class *Generalization* is used to create an inheritance relationship between the two transformed UML classes.

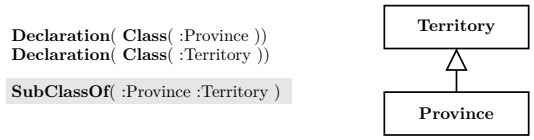


Fig. 2. Example for the transformation of a `SubClassOf` axiom.

If in case (b) one of the two element types is a CE defined by necessary and sufficient conditions, one has to distinguish whether it is the sub-class C_c or super-class C_p. If it is the sub-class C_c it is a case as shown in this example:

```
SubClassOf(
   DataMinCardinality( 1 :hasISBN )
   :Book
)
```

The element type C_c is defined by a formula ϕ. Every individual e is an instance of this element type if, and only if, it satisfies the formula:

$$C_c(e) \leftrightarrow \phi(e)$$

The fact that an individual e is an instance of C_p is denoted by $C_p(e)$. From the inheritance relationship

$$C_c(e) \rightarrow C_p(e)$$

immediately follow (by substituting $C_c(e)$)

$$\phi(e) \rightarrow C_p(e)$$

Meeting the formula $\phi(e)$ is therefore a sufficient condition that an object is an instance of the element type C_p. Since UML does not support automatic classification on the basis of sufficient conditions, this case is not transformable.

The situation in case (b) is different, if the element type defined by necessary and sufficient conditions appears as super-type C_p. Here is an example for this case:

```
SubClassOf(
   :Book
   DataMinCardinality( 1 :hasISBN )
)
```

Taken alone, the element type C_p could not be transformed into a UML class. However, the fact that the element type C_p occurs within in `SubClassOf` axiom as super-type makes the necessary condition of the CE a necessary condition of the sub element type C_c. This is shown in the following:

The element type C_p is defined by a formula ϕ. Every individual e is instance of this element type if, and only if, it satisfies the formula:

$$C_p(e) \leftrightarrow \phi(e)$$

The fact that an individual e is an instance of C_c is denoted by $C_c(e)$. From the inheritance relationship

$$C_c(e) \rightarrow C_p(e)$$

immediately follow (by substituting $C_p(e)$)

$$C_c(e) \rightarrow \phi(e)$$

To meet the formula $\phi(e)$, it is therefore a necessary condition that an object is an instance of the element type C_c. The example shown in the listing above can be read as: "Every book must have at least one ISBN." Such necessary conditions are in turn easily transformable into UML.

C. Abstract Elements

An element type is called abstract if it cannot be instantiated. Thus, one might assume, models containing abstract classes are by definition inconsistent. Wahler et al. provide a solution to this apparent problem in [5]. UML allows the definition of abstract classes that must not have direct instances. Instantiation is only allowed for subclasses. OWL knows no language construct to define that a class must not contain any individual directly, and only one of its subclasses is allowed to contain individuals.

Usually, abstract classes appear in connection to generalizations. In that context, it is possible to treat them as "normal" classes. However, the limitation remains that after the transformation from UML to OWL, it cannot be assured that an abstract class does not contain any direct instances.

During the transformation of instances of the OWL meta class `ObjectUnionOf` in connection with inheritance relationships, abstract element types play a role. This is discussed in the section about generalizations below.

D. Element types with fixed population

An element type with a fixed population consists of a predefined set of objects that make up the population of this element type. It is not possible to classify more objects as instances of the element type. UML has no way to specify that the population of an element type may only consist of a fixed set of objects. For data types, it is possible to define element types with fixed population by using an enumeration.

In OWL, elements with a fixed population can be defined for both individuals as well as for values of data types. For a fixed set of individuals, this is a CE. A predefined set of values is a data range. It is not necessary that the listed individuals are instances of the same element type. Similarly, the values listed do not have to belong to the same data type. Since, in UML one can only defined data types with fixed population, a transformation of the `ObjectOneOf`-CE is not possible. Both transformation directions for data types are described in the corresponding sections about enumerations.

E. Generalization

A generalization is an extension of the inheritance concept discussed above. Usually, more than two element types are put into relation. It is also possible to specify whether it is a complete and/or non-overlapping generalization.

In UML, the generalization of element types is represented similarly to the the inheritance of element types. Also, information about whether a generalization is complete and/or non-overlapping (disjoint) can be expressed in UML. Four

characteristics can be used for this: complete, incomplete, disjoint, or overlapping.

In addition to simple inheritance with the help of a `SubClassOf` axiom, OWL knows another axiom that can be used to define that the element types $E_2 \dots E_n$ constitute a complete, non-overlapping partition of E_1: `DisjointUnion(`$E_1, E_2, \dots, E_n$`)`.

Because of the different possibilities in UML to specify completeness or non-overlapping of generalization relationships, different cases for the transformation UML $\rightarrow$ OWL can be identified:

- general case
- non-overlapping generalization
- non-overlapping and complete generalization
- complete (but not non-overlapping) generalization
- generalization of data types—treated in the section on data types

General case: The concepts of specialization and generalization in UML and OWL are very similar. If E' is a specialized sub-type of an element type E and i is an instance resp. individual, $E'(i) \rightarrow E(i)$ holds true in both cases. Therefore, the transformation is quite simple. For every generalization relationship "E is generalization of E' (resp. E' is specialization of E)", the axiom `SubClassOf(` E' E `)` is added to the ontology.

Non-overlapping generalization: A generalization is non-overlapping (disjunct), if an instance of a sub-type must not be an instance of another sub-type of the same generalization at the same time. This restriction can be enforced by adding a `DisjointClasses` axiom (instance of the OWL meta class `DisjointClasses`), which contains all sub element types of the generalization.

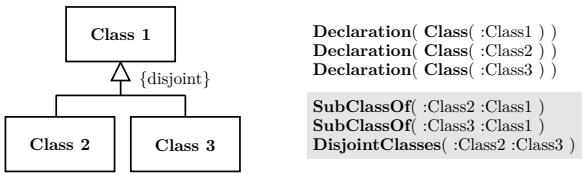


Fig. 3. Example for the transformation of non-overlapping (but not necessarily complete) generalization.

Non-overlapping and complete generalization: If a generalization is non-overlapping and complete, a stronger axiom can be used. A `DisjointUnion(`E E'_1 ... E'_n`)` axioms– which is actually only a shorthand notation—states that every individual, which is an instance of element type E, is an instance of exactly one element type E_i and every individual, which is an instance of element type E_i, automatically belongs to the population of E.

Complete generalization: The situation is similar when the generalization is complete, but not overlapping. In this case, the `DisjointClasses` axiom contained in the axioms combined in the shorthand notation of a `DisjointUnion` axiom is not necessary. This results in the solution shown in Figure 5, which uses an `ObjectUnionOf`-CE and a `EquivalentClasses` axiom.

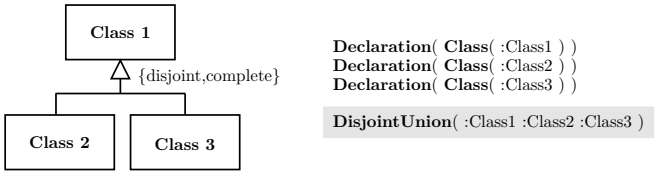


Fig. 4. Example for the transformation of a non-overlapping and complete generalization.

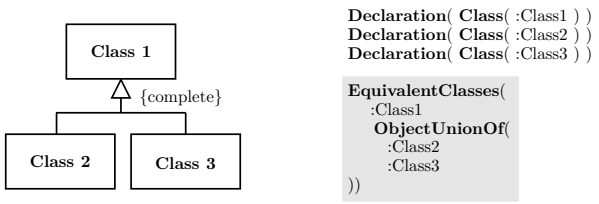


Fig. 5. Example for the transformation of a complete but not non-overlapping generalization.

1) ObjectUnionOf: An `ObjectUnionOf(`$E_1 \dots E_n$`)` defines an element type with a population consisting of those individuals that are instance of one or more element types $E_1 \dots E_n$. If $E_1 \dots E_n$ can be transformed into UML classes $C_1 \dots C_n$, the `ObjectUnionOf` can be represented as abstract class.

Between the abstract class and each member of the union $C_1 \dots C_n$ generalization relations (instances of the UML meta class *Generalization*) must be created during the transformation process. The generalizations are combined into an instance of the UML meta class *GeneralizationSet*. It is marked as *complete*. This is possible, because in UML the definition of a union of element types $E_1 \dots E_n$ is semantically equivalent to an abstract element type and the specification of subtypes.

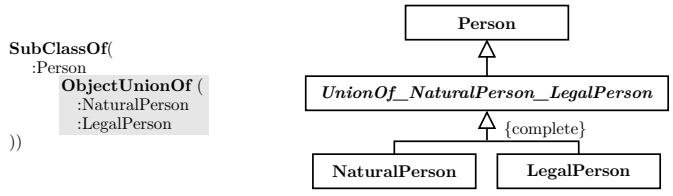


Fig. 6. Example for the transformation of an instance of the OWL meta class `ObjectUnionOf` into an abstract class (instance of the UML meta class *Class*) and inheritance relationships with *GeneralizationSet*.

DisjointUnion: The `DisjointUnion` axiom is a syntactical shortcut. The axiom `DisjointUnion(`A B_1 ... B_n`)` is semantically equivalent to the three axioms

`SubClassOf(A ObjectUnionOf(`B_1 ... B_n`))`
`SubClassOf(ObjectUnionOf(`B_1 ... B_n`) A)`
`DisjointClasses(`B_1 ... B_n`)`

In this particular case, the above mentioned problems of `SubClassOf` axioms and sufficient conditions for class membership can be circumvented and the semantics of the expression can be transformed. The long notation of a `DisjointUnion` axiom contains a `ObjectUnionOf(`B_1 ... B_n`)` CE. As described above, this CE is transformed into $n + 1$ UML classes, n generalization relations and an instance of

the UML meta class *GeneralizationSet* which is marked as *complete*. However, in this case the superclass is not abstract. A name is assigned to this class that corresponds with the IRI of the OWL class. Additionally, the instance of the UML meta class *GeneralizationSet* is marked *disjoint* to reflect the disjointness of the classes $B_1 \ldots B_n$.

F. Intersection

An element type can be defined by the intersection of other element types. An object is an instance of that element type if it is an instance of a set of other element types.

Since UML has no model element similar to *GeneralizationSet* for specializations, the semantic of this construct cannot be expressed completely. If it is possible to transform the element types $E_1 \ldots E_n$ that are part of the `ObjectIntersectionOf`-CE into UML classes $C_1 \ldots C_n$, it is only possible to define an abstract class, which is a subclass of the classes $C_1 \ldots C_n$. Since UML does not allow an automatic classification by sufficient conditions, it is not possible to enforce that an object, which is a instance of all classes $C_1 \ldots C_n$, is also an instance of the abstract subclass.

IV. Data Types

In general, a datatype consists of three components: the value space, the lexical space, and a well-defined mapping from the lexical into the value space. The value space is the—possible infinite—set of values that can be represented by the datatype. The lexical space describes the syntax of the datatype's values. The mapping is used to map syntactically correct values to elements of the value spaces. It is possible that many, even infinite many, syntactically different values are mapped to the same element of the values space.

Primitive datatypes do not have an internal structure. Examples of primitive types are character strings, logical values, and numbers.

Enumerations are a special kind of datatypes with no internal structure. In contrast to general primitive types the lexical space and the value space of an enumeration are equal-sized, well-defined finite sets. The mapping from lexical to value space is a one-to-one mapping. An example for an enumeration datatype are the English names of the days of the week which consist of seven possible values.

In contrast to primitive data types, **complex data types** have an internal structure. The following are examples for complex data types:

- a person's name consisting of a given name and a family name
- a physical measurement consisting of value and unit of measurement
- an address consisting of street name, house number, postal code and city name

Generalization of datatypes can be defined similarly to the generalization of element types. If a datatype A generalizes a datatype B each date that is instance of B (i.e., its lexical representation belongs to the lexical space of B and its value belongs to the value space of B) is also instance of datatype A. For example the integers generalize natural numbers. Each natural number is also an integer.

A. Representation in UML

Apart from a few pre-defined primitive types UML allows the definition of additional datatypes in class diagrams. These can be primitive types, complex datatypes, and enumerations. In UML, datatypes—similar to classes—can have owned attributes (as well as operations which are not discussed here). Therefore, they can be used to describe structures. Figure 7 shows examples for the three kind of datatypes.

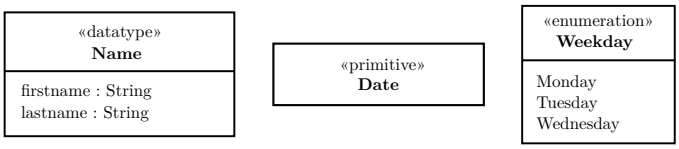


Fig. 7. Examples for datatypes in UML. Left: user-defined datatype with two components. Center: user-defined primitive datatype. Right: Enumeration with three allowed values.

In contrast to instances of classes, "any instances of that data type with the same value are considered to be equal instances."[6, p. 63] Although the graphical representations of datatypes in general (instances of *DataType*) as well as primitive types (instances of *PrimitiveType* and enumerations (instances of *Enumeration*) in particular look similar to the representation of classes (instances of *Class*), they are different elements of the meta model as shown in Figure 8.

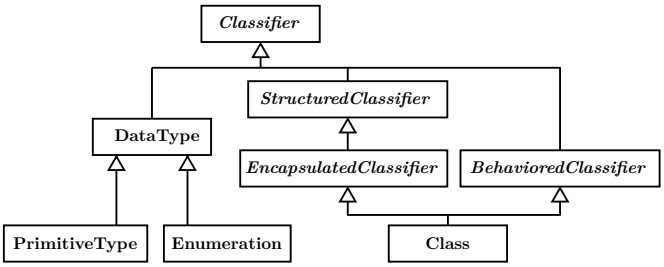


Fig. 8. Extract from the UML meta model, showing the difference between classes and datatypes.

In UML, generalizations are defined for *Classifier* and therefore also for *DataType*. Thus, inheritance/generalization relations between datatypes can be defined in a UML class diagram.

B. Representation in OWL-2

In OWL-2, three different kinds of datatypes can be distinguished:

1) `rdfs:Literal` as base datatype
2) datatypes of the OWL-2 datatype map, which is basically a subset of the XML Schema datatypes [7].
3) datatypes that have been defined within an ontology using `DatatypeDefinition`

The value space of the base datatype `rdfs:Literal` is the union of the value spaces of all other datatypes. The OWL-2 datatype map adopts the value space, lexical space, and the restrictions for user-defined datatypes from

the XML Schema specification. Sets of values (instances of datatypes)—so called *Data Ranges*—can be defined by combining datatypes via common set-theoretic operations. A set of values consisting exactly of one pre-defined list can be described by using `DataOneOf`. A `DatatypeRestriction` allows to define a set of values by restricting the value space of a datatype with *constraining facets*. The OWL-2 datatype map defines which restrictions are allowed. For example a number datatype can be restricted by: less equal, greater equal, equal, and greater.

An OWL-2 datatype is defined by assigning an Internationalized Resource Identifier (IRI) to a `DataRange` using a `DatatypeDefinition` axiom. According to the OWL-2 DL specification this IRI must have been declared to be the name of a datatype.

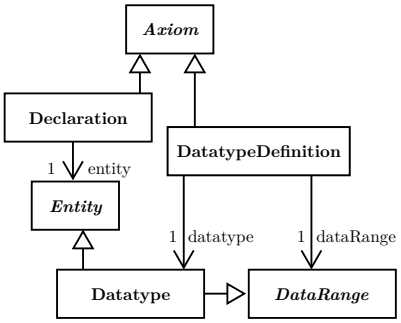


Fig. 9. Extract relevant for datatypes from the OWL-2 meta model.

The abstract syntax (see Figure 9) shows that a datatype is linked indirectly (via an instance of `DatatypeDefinition`) to its value space (an instance of a subclass of `DataRange`. Therefore, it is possible to use a datatype with no assigned value space. By definition this datatype has the value space of `rdfs:Literal`.

Subclasses of `DataRange` (e.g., `DataUnionOf`), which are used for the definition of value sets (and therefore datatypes), have references to `DataRange`. `Datatype` is a subclass of `DataRange`, too. Thus, arbitrarily nested constructions of datatype-defining elements are possible.

C. Primitive Types

Three cases have to be considered for the UML → OWL-2 transformation of primitive types:

1) The datatype is one of the four pre-defined datatypes "Boolean", "Integer", "String", or "UnlimitedNatural".
2) The datatype is one of the XML Schema datatypes.
3) The definition of the (user-defined) datatype is part of the UML-model.

Since OWL-2 uses the datatype-definitions from XML Schema, a datatype in case (1) can be transformed into its corresponding datatype from XML Schema. Primitive types can be recognized by the fact that they are contained in a package "UMLPrimitiveTypes".

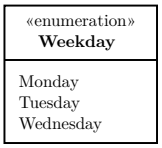
The transformation in case (2) is even more obvious because a datatype is used that is also present in OWL-2. The name of the package containing the primitive types depends on the UML type library used. A common package name is "XMLPrimitiveTypes". This name can be used to recognize primitive types that fall under case (2). The XML Schema datatype can be referenced in the ontology by adding the XSD namespace to the type's name.

For user-defined datatypes in case (3), a new datatype is defined in the ontology by using a `Datatype` axiom. OWL-2 datatypes—like all OWL-2 model elements—are identified by unique IRIs. Therefore, an appropriate IRI must be generated during the transformation. In UML, elements (including datatypes) are uniquely identified by their name and package hierarchy. Therefore, a combination of package and datatype name can be used for the IRI.

For the transformation OWL-2 → UML, primitive types are difficult. OWL-2 offers a variety of possibilities to define new datatypes. However, some primitive types—and probably the most common ones—can be transformed. The primitive types of OWL-2 derive from the XML Schema datatypes. There are established UML-libraries for the XML datatypes. Therefore, it is sufficient to include such a library into the transformation process. An instance of a primitive type contained in the library can be looked up by the IRI of the OWL-2-datatype and references as necessary.

D. Enumerations

As mentioned in Section VIII, several authors have already discussed how to transform enumerations. In OWL-2 the data range `DataOneOf` is suitable to define a datatype with a fixed pre-defined value space. Each lexical value of the `DataOneOf` data range is transformed into an *EnumerationLiteral* instance and vice-versa. OWL-2 as well as UML support the specification of datatypes for the elements of an enumeration: An OWL-2 *Literal* instance has a *datatype* attribute, an UML *EnumerationLiteral* instance has a *classifier* attribute referencing the datatype.




```
Declaration( DataType( :Weekday ) )

DatatypeDefinition(
  :Weekday
  DataOneOf( "Monday" "Tuesday" "Wednesday" )
)
```

Fig. 10. Example for the transformation of an enumeration.

For the transformation OWL-2 → UML one has to consider the fact that in OWL-2 the data range `DataOneOf` can be used without a `DatatypeDefinition` which assigns a name to it. Since an UML *Enumeration* necessarily needs a name, it can be generated based on the literals contained in the data range.

E. Complex Data Types

OWL-2 datatypes consist of exactly one literal and are therefore not further structured. Since OWL-2 is built upon the Resource Description Framework (RDF), there is the theoretical possibility to use a blank node and the RDF-instruction `parseType="Resource"` to implement complex data as shown in this listing:

```
<rdf:RDF xml:base="http://example.com/persons/"
```

```
   xmlns="http://example.com/persons/"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

   <owl:Ontology rdf:about="http://example.com/persons/"/>

   <owl:Class rdf:about="Person" />

   <owl:NamedIndividual rdf:about="Timmi">
      <rdf:type rdf:resource="Person"/>
      <hasName rdf:parseType="Resource">
         <first>Timmi</first>
         <last>Tester</last>
      </hasName>
   </owl:NamedIndividual>
</rdf:RDF>
```

However, neither the OWL-1 nor the OWL-2 specification mention `parseType="Resource"`. Therefore, it is probably not a valid construct for OWL-2. Even if this notation was valid for OWL-2 and an element type could be assigned to such an anonymous individual, the definition of the element type would be indistinguishable from the definition of a "normal" element type.

The UML $\rightarrow$ OWL-2 transformation of complex datatypes, i.e., datatypes with owned attributes, is similar to the transformation of UML classes with owned attributes into OWL-2 classes and properties. There are two characteristics of UML datatypes that have to be considered:

1) Values do not have an identity.
2) Every value exists only once.

Since the transformation is similar to the transformation of classes the instances of the resulting element in the ontology will be individuals. In OWL-2, every (typed) individual must have a name. Therefore, the semantics for characteristic (1) is changed: in UML, the instance of the datatype does not have an identity. The corresponding individual in OWL-2 is assigned with an IRI by which it can be referenced (and also identified).

Characteristic (2) requiring that every value must exist not more than once, can be ensured by using `HasKey` axioms. For every UML datatype D with owned attributes $a_1 \dots a_n$ that is transformed into a OWL-2 class C with data property $dp_1 \dots dp_n$, the following axiom is added to the ontology:

`HasKey(` C `()` `(` $dp_1 \dots dp_n$ `)` `)`

This axiom ensures that every occurrence of an individual with the same values for $dp_1 \dots dp_n$ is one and the same individual.

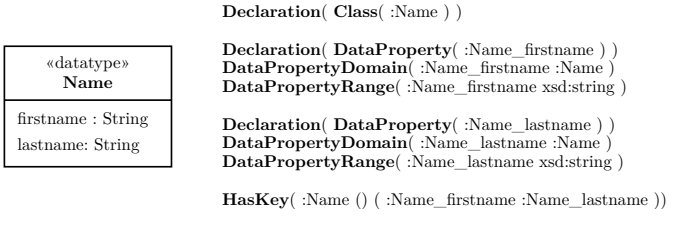


Fig. 11. Example for the transformation of a complex datatype.

F. Generalization of datatypes

In general, the transformation of a datatype generalization in a UML class diagram is not possible, since OWL-2 has no support for inheritance/generalization of datatypes. In the special case of a complete generalization of datatypes with no internal strcuture (e.g., enumerations), a transformation is possible: While the generalization of UML classes can be transformed into an OWL-2 `ObjectUnionOf` class expression, this is not possible for datatypes. As the name suggest, an `ObjectUnionOf` can only be used for classes. Instead, an instance of `DataUnionOf` is used. The sub-datatypes combined in the `DataUnionOf` constitute a new data range. By using a `DatatypeDefinition` axiom a name is assigned to this set of datatypes. This name is the name of the super-datatype from UML. Figure 12 shows an example for such a transformation.

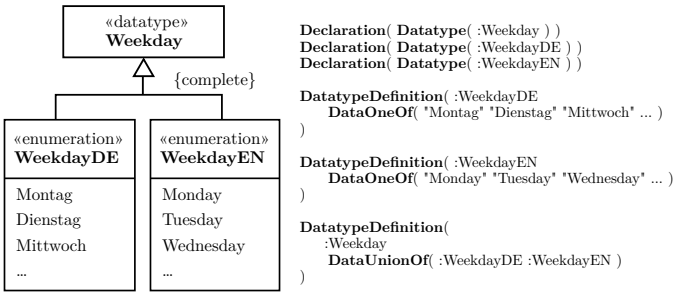


Fig. 12. Example for the transformation of a generalization relation between datatypes.

V. Relationship Types

In a software system one can typically find a variety of relations between instances of the element types. The characteristics of these relations are described by means of relationship types. Instead of the term "relationship type" the term "relation" is often used. This is not quite correct, as a relation refers to a concrete instance of a relationship type between instances of element types. A relationship type describes which characteristics apply to all of its relations in general.

A relation includes participants (=participating instances of an element type) that play a certain role. Applied to relationship types, these participants are element types that play a certain role within the relationship type. It is possible to omit the indication of roles for a relation and the relationship type, respectively.

A relationship type with two members is called binary relationship type. Corresponding relations are called binary relations, accordingly. Since arbitrary n-ary relations can be transformed into binary relations [4, Chap. 6][8], only binary relationship types are considered in the following.

In UML, binary relationship types can be presented in two different ways—as associations or as class-dependent attributes. Associations are depicted by lines between element types. The name of the relationship type can be written close to the line, maybe with an arrow indicating the direction. Class-dependent attributes are listed in the middle section of an element type. Although the concrete graphical syntax of associations and attributes differs significantly from each other, both are represented in the abstract syntax by the same UML meta class *Property*. Because of this similarity, it is

useful to consider the transformation of associations and class-dependent attributes together, since they differ only within a few aspects.

OWL knows two different constructs to connect elements:

- Object Properties – for relations between instances of classes
- DataProperties – for relations between an instance of a class and an instance of a datatype.

At the declaration of an OWL property—i.e., an indication that a property with this name exists—initially no information on the related element types E_1 and E_2 is made. Therefore, an instance of such a relationship type can be used for the connection of arbitrary objects. Only by the use of further axioms statements about the domain and the range of the property are made. Thus, the element types E_1 and E_2 are determined.

Since the UML meta class *Association* that represents an association is a sub-type of *Classifier*, all associations are direct components of a UML package. The OWL concept most similar to an association is an object property. Via its declaration it is also a component of the ontology. The transformation of class-dependent attributes is a bit more complicated because there is no obvious corresponding concept to them in OWL. The main issue is that in OWL classes do not contain any other model elements—as it is the case in UML. But again, the connection can be mapped to an object property or a data property. In both cases, the decision on whether an instance of the UML meta class *Property* is mapped to an object property or a data property depends on the kind of element type the *Property* instance points to. If it is an instance of the UML meta class *Class* or a complex datatype—i.e., an instance of the UML meta class *DataType* with dependent attributes—an object property will be used. If however, it is an instance of a simple data type—i.e., an instance of the UML meta classes *PrimitiveDataType* or *Enumeration*—, a data property will be used.

In contrast to UML, where the types of an association (domain and range) are always specified, it is optional to specify domain and range for properties in OWL. Per definition, the domain and range of such properties is `owl:Thing` (respectively `owl:Literal` for the range of a data property). Such properties can be used to connect instances of arbitrary element types. In order to restrict the properties like in an UML model, it is necessary to add appropriate axioms that specify the allowed classes and datatypes for domain and range of the property. The range of an instance of the UML meta class *Property* can be determined as follows: it is the element type that is connected to the *Property* instance via an instance of the UML meta class *Type* appearing in the role *type*. To determine the domain one has to distinguish between associations and class-dependent attributes.

- In case of a class-dependent attribute a connection exists between the *Property* instance and a instance of *Class* in role *class*. The element type represented by this *Class* instance is the wanted type.
- If the *Property* instance is part of an association (i.e., a connection to an instance of *Association* where the role *association* exists), the type of the other member-end of the association has to be chosen.

Figure 13 illustrates this selection of domain and range.

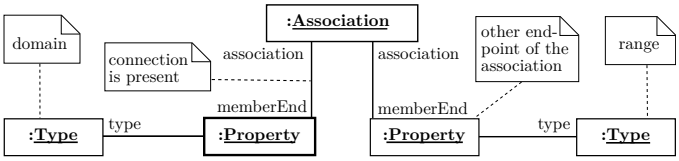


Fig. 13. Selection of the domain and range in case of an association. The focus is on the thicker bordered *Property* instance.

In order to avoid that two OWL properties that are the transformation result of different instance of the UML meta class *Property* are interpreted as a single property, all those properties that are not in an inheritance relationship will be marked as disjoint. For this purpose, `DisjointObjectProperties` and `DisjointDataProperties` axioms are added to the ontology.

As described above, OWL distinguishes between data properties connecting classes with datatypes, and object properties connecting classes with classes. Data properties are mapped onto class-dependent attributes in UML. With a few exceptions object properties are also transformed into class-dependent attributes. Object properties to which there is an inverse relation within the ontology are treated seperately. Such an inverse relation may be specified in several ways: by explicit specification using an `InverseObjectProperties` axiom, by using an anonymous inverse `InverseObjectProperty` or by marking an object property as symmetric or inverse-functional.

It must be expected that more than one class is specified as domain or range of a property. UML does not allow this notation. In such cases, a helper class is added, which inherits from all classes in the domain or range. Figure 14 illustrates such a construction. At those places where the affected object property is used, the new auxiliary class will be used in the UML model, accordingly.

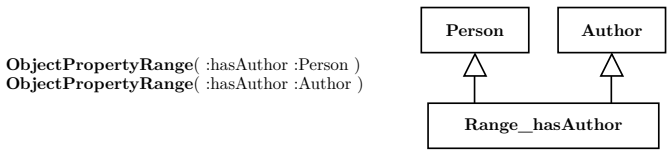


Fig. 14. Example for the transformation of multiple axioms indicating the range of an object property into UML classes with inheritance relations.

One possibility to transform object properties with `owl:Thing` as domain and/or range is to define a single base class C_{super} within the UML model. C_{super} is defined as super-class of all other classes in the model and thus corresponds to `owl:Thing`. In this particular case of a single base class, an object property without definition of domain and range can be mapped to an instance of the UML meta type *Association* with two member ends of type C_{super}.

A. Inheritance

Sometimes it is necessary that from one relationship between two objects, another relationship follows automatically.

In other words, the population of a relationship type R_2 includes the population of another relationship type R_1. All instances of R_2 are automatically also instances of R_1.

In UML, the inheritance of relationship types is realized similarly to the inheritances between element types. If both relationship types are represented by an association, a generalization between these two associations can be created. In the case of class-dependent attributes, a subsetted attribute can be specified. Also, OWL allows to express that two relationship types are in an inheritance relationship. For this purpose, the axioms `SubObjectPropertyOf` and `SubDataPropertyOf` exist. An example of such a transformation is shown in Figure 15.

An inheritance relationship (instance of the UML meta class *Generalization*) between two instances of the UML meta class *Association* can be transformed into OWL by using instances of the OWL meta classes `SubPropertyOf`. Since a bidirectional association is transformed into two object properties, the generalization between two bidirectional associations must be transformed into two instances of the OWL meta class `SubPropertyOf` as well.

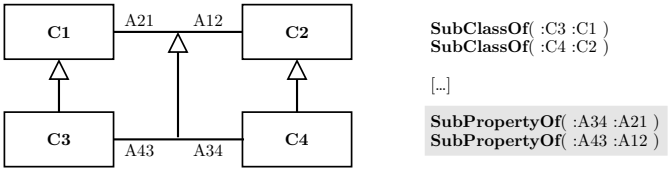


Fig. 15. Example for the transformation of generalized associations.

For the transformation of a `SubObjectPropertyOf` axiom the class-dependent attribute *subsettedProperty* of the UML meta class *Property* can be used. It specifies the parent *Property* instance.

B. Cardinality Constraints

Cardinality constraints are one of the most important types of restrictions for conceptual modelling. They allow a technical limitation of the quantity of relations that an object can have with other objects. In UML, cardinality constraints are called multiplicities and can be specified for both associations as well as class-dependent attributes.

OWL also knows constructs to describe cardinalities. Class expressions (CE) are used to describe certain sets of individuals. In the following, a binary relation $R(x, y)$ between two individuals x and y is considered.

The CE `ObjectMinCardinality( n R )` describes the set of individuals that are in a relation R to more than n individuals: $\{x : |\{y : R(x, y)\}| \geq n\}$. Corresponding CEs exist for sets of individuals that are linked with less than n individuals (`ObjectMaxCardinality`) or exactly n individuals (`ObjectExactCardinality`). If y is a datatype, the CEs `DataMinCardinality`, `DataMaxCardinatlity`, or `DataExactCardinatliy` are used.

`ExactCardinality` is only a shortcut for a pair of `MinCardinality` and `MaxCardinality` elements. However, using this shortcut improves the readability of the ontology. Therefore, a UML cardinality constraint with the value for upper and lower bound is tranformed into an instance of the UML meta class `ExactCardinality` with a equivalent value.

If the value of the upper bound is 1, an additional instance of the OWL meta class `FunctionalObjectProperty` or `FunctionalDataProperty` is created. Again, although this is only a abbreviating notation, the additional axiom improves the comprehensiveness of the ontology, because the type of property is easier to recognize.

However, it is not possible and sufficient to add the corresponding CE for the cardinality constraints directly to the ontology. On the one hand, CEs are not axioms and can therefore not be added to the ontology directly. On the other hand, in UML, cardinality constraints of class-dependent attributes and associations always refer to a specific class. In OWL properties are not directly contained in classes (see above). Therefore, cardinality constraints defined for properties as CE do not affect classes.

These difficulties can be solved by adding an instance of the OWL meta class `SubClassOf` to the ontology for every cardinality constraint. The sub-type is defined as the OWL class that corresponds to the UML class for which the relationship type and its cardinality constraint were defined. The super-type is the CE that represents the cardinality constraint. Thus, the cardinality constraints of the CE are inherited by the class that is related to the association or the class-dependent attribute.

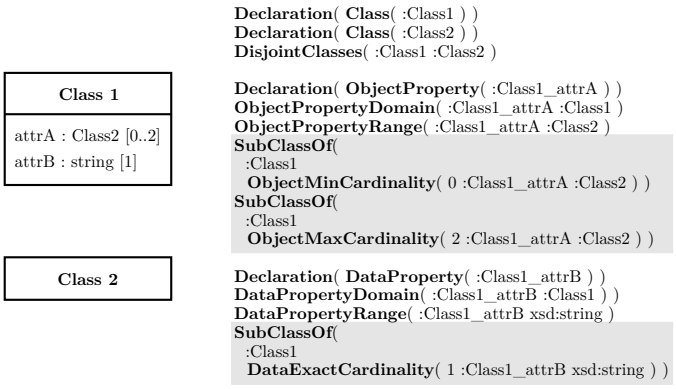


Fig. 16. Example for the transformation of klass-dependent attributes and associations with cardinality constraints.

For the direction OWL $\rightarrow$ UML it has to be considered that the CE `ObjectMinCardinality` and `ObjectMaxCardinality`—and those for data properties, respectively—define anonymous element types that specify restrictions on the occurrence of a property. If an anonymous element type, defined by such cardinality constraints is used as superclass C_p within an `SubClassOf`$(C_c\ C_p)$ axiom, the cardinality constraints become constraints of the class-dependent attributes of the subclass C_c. Figure 17 illustrates this case.

OWL offers two axioms to characterize properties as functional. However, both axioms are only syntactic shortcuts for a subclass axiom and a cardinality CE. Therefore, they can be transformed into cardinality constrains 0..1 of the corresponding instance of the UML meta class *Property*. The `InverseFunctionalObjectProperty` axiom to char-

```
Declaration( Class( :Book ) )
Declaration( Class( :Person ) )
Declaration( ObjectProperty( :author ) )
Declaration( DataProperty( :title ) )

SubClassOf( :Book
   ObjectMinCardinality( 1 :author :Person ))
SubClassOf( :Book
   ObjectMaxCardinality( 5 :author :Person ))
SubClassOf( :Book
   DataExactCardinatliy( 1 :title xsd:string ))
```

Book
author : Person [1..5]
title : string [1]

Fig. 17. Example for the transformation of various cardinality restricting axioms into cardinality constrains of class-dependent attributes.

acterize an object property as inverse-functional is also a syntactic shortcut. However, the problems with `SubClassOf` axioms and sufficient conditions for class membership mentioned above prevent a similar transformation. The restriction can be preserved by mapping the OWL property onto one member-end of an instance of the UML meta class *Association* and by setting a cardinality constraint 0..1 for the other member-end of that association.

C. Value constraints

In case of a value restriction, the second participant of a relationship type is set to exactly one value or an object. In UML, it can be defined that a class-dependent attribute is pre-set to a fixed value during the instantiation. In order to prevent the change of this value even during dynamic use, this value can also be marked as immutable. For both, object properties as well as data properties, OWL offers the possibility to set the second participant to a fixed value. The value constraint can also be achieved by defining a one-element element type and a corresponding cardinality axiom. However, with `ObjectHasValue` and `DataHasValue`, OWL offers shortcuts, which make the semantics clearer to a human reader.

Similar to the transformation of class-dependent attributes with cardinality constraints into `SubClassOf` axioms of the containing class, as a value constraint is transformed into a `SubClassOf` axiom as well. The super-type will be an instance of the OWL meta class `DataHasValue` (a data range). Within that data range, the fixed value and the data property generated from the class-dependent attribute are defined.

In the transformation OWL $\rightarrow$ UML, one can use the fact that if an element type defined by a `DataHasValue` CE is used as super-type C_p within `SubClassOf(`C_c C_p`)` axiom the fixed value becomes a necessary condition for instances of the class C_c. Every single instance of that class will have exactly this fixed value. In a UML model, this value can be defined for an instance of the UM meta class *Property* by setting the class-dependent attribute *default*.

```
Declaration( Class( :Book ) )
Declaration( DataProperty( :title ) )
Declaration( DataProperty( :printed ) )

SubClassOf( :Book
   DataExactCardinatliy( 1 :title xsd:string ))
SubClassOf( :Book
   DataHasValue( :printed "true"^^xsd:boolean ))
```

Book
title : string [1]
printed : boolean = true

Fig. 18. Example for the transformation of an axiom for a value constraint into a value constraint of a class-dependent attribute.

D. Part-Whole-Relationships

Part-whole relations (also known as composition and aggregation) are a special kind of relationship types. They play an important role in modelling [4, p. 137]. They can be used to express a certain semantic of the relationship. Additionally, further restrictions can be imposed on the respective relationship type. A part-whole relation is antisymmetric—i.e., if T is part of G, G can not be part of T. There is disagreement on whether part-whole relations are transitive or not [4, p. 142]. Since a part-whole relation is a binary relationship type, the previously made statements and observations for general relationship types also apply to this kind of relationship types.

UML knows two kinds of part-whole relations: aggregation and composition. They differ in both, (graphical) syntax and their semantics. An aggregation is anti-symmetric and transitive. Objects linked by it form an acyclic graph [9, p. 171]. It is allowed that a "part" is part of more than one "whole". Composition is a stronger form of aggregation. In addition to anti-symmetry and transitivity, a "part" may—at a time—be only part of a single ''whole". Futhermore, the existence of a "part" depends on the existence of the "whole". It can not exist without something it is part of. OWL has no special constructs to identify part-whole relationships.

Like other associations between two classes, part-whole relation types are transformed into object properties. Moreover, the additional restrictions mentioned above are taken into account:

a) aggregations are antisymmetric

b) an object must not be in an aggregation relation to itself—that would be a contradiction to the antisymmetry,

c) an object must not be part of more than one composition,

d) an instance of a class that is part of a composition must not exist alone.

The asymmetry can be achieved by adding an `AsymmetricObjectProperty` axiom to the object property that has been transformed from the association with aggregation or composition characteristic.

Restriction (b) can be transformed to OWL by adding an `IrreflexiveObjectProperty` to the ontology for each association with aggregation or composition characteristic. This axiom prohibits the use of the corresponding object property to connect an individual with itself.

Restriction (c) can be achieved by adding a `FunctionalObjectProperty` or an `InverseFunctionalObjectProperty` axiom. If the association with composition characteristic is bidirectionally navigable, it makes no difference what type of axiom is used. However, if the association is only navigable from one direction the following distinction has to be made. If the association is navigable from "part" to "whole", a `FunctionalObjectProperty` is used. A connection between an individual of the "part"-class to more than one individual of the "whole"-class would make the ontology inconsistent. An `InverseFunctionalObjectProperty` axiom is used if the association is navigable from "whole" to "part".

The enforcement of a constraint of the form (d) is not possible, since the open world assumption is used for OWL. The individual in question could be part of a composition that is not explicitly listed in the ontology.

E. Inverse

If a conceptual model contains an relationship type $R(E_1, E_2)$ with two participating element types E_1 and E_2, a common wish is to have the choice to use instances of both element types as first or second participant. To make this possible, one can define an inverse relationship type $R_{inv}(E_1, E_2)$.

For example, consider a book containing chapters. The relationship type $contains(Book, Chapter)$ is defined for the element type $Book$ and $Chapter$. Instances of type $Book$ must always appear as the first participant of such a relationship. If a relationship type $isContainedIn(Chapter, Book)$ that is inverse to $contains$ is defined, statements equvialent to those with $contains$ can be made with an instance of type $Chapter$ as first participant.

Although UML provides no possibility to explicitly mark two arbitrary associations as inverses of each other, the two ends of a binary bidirectional association can be seen as two inverse relations.

The definition of inverse relationship types is only possible for object property, not for data properties. A value (an instance of a datatype) must not contain properties itself. OWL offers three possibilities to use inverse relationship types:

1) An `InverseObjectProperties` is used to declare two previously defined object properties as inverses of each other..
2) The inverse of an object property can be used directly by using the object property expression `ObjectInverseOf` without assigning a name to it.
3) An object property is marked as inverse-functional.

It should be noted that for two individuals a and b the inverse object property $op_{inv}(b, a)$ is automatically part of the ontology if $op(a, b)$ is contained in the ontology.

During the transformation UML $\rightarrow$ OWL bi-directional associations are transformed into two—initially independent—object properties. However, such associations are equivalent to two directed mutually inverse associations. [9, p. 165] Therefore, an instance of the OWL meta class `InverseObjectProperties` is created and linked with the corresponding instances of `ObjectProperty`.

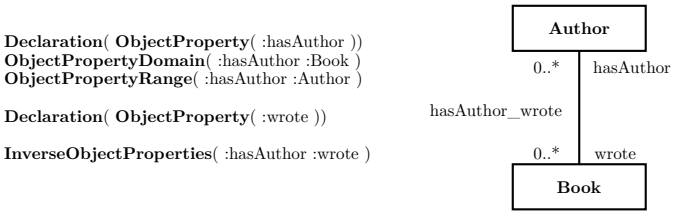


Fig. 19. Example for the transformation of an inverse object property into an instance of the UML meta class *Association*.

When transforming inverse object properties in OWL ontologies into UML no class-dependent attributes are used—in contrast to the generic case described above. Otherwise, the connection between the two links could not be seen. Instead, an instance of the UML meta class *Association* is used as transformation target. The two instance of the UML meta class *Property* occurring as member-ends are mutually set as the value of the *opposite* attribute.

VI. Constraints

Some very common constraints have been discussed in the previous sections, such as cardinality constraints or non-overlapping generalizations. In this section, further restrictions on element types and relationship types will be discussed, namely keys for element types and "conditional relationship types" that work on a combination of element and relationship types.

A. Key Constraints

Key constraints can be used to enforce that there are no two different instances of an element type for which all relations specified in the key have an identical value, or point to the same object. There are simple keys that are based on only one relationship, as well as composite keys, which are based on multiple relations.

UML offers the possibility to define a single key per element type. Class-dependent attributes (instances of the meta class *Property*) can be marked that are part of this key. These marked attributes can be used to identify an instance of the element type.

OWL offers the `HasKey` axiom to define composite keys. Such a key can not only be defined for named element type but also for any CE. The relationship types to be considered are divided into two sets, the object properties and the data properties. With the axioms `FunctionalObjectProperty` and `FunctionalDataProperty` OWL provides yet another way to define especially strong simple keys. An identity is defined independently of the element type of the object, on the basis of the relationship alone.

To transform a UML class with a key, i.e., some of its class-dependent attributes are marked with `isID=true`, a corresponding instance of the OWL meta class `HasKey` is added to the ontology.

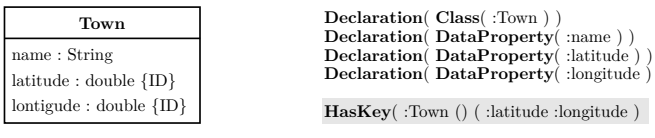


Fig. 20. Transformation of a composite key constraint.

The information that the properties appearing within `HasKey` axioms of an OWL ontology form a key can be transformed into an UML model by setting the value of the class-dependent attribute *isID* for the instances of the UML meta class *Property* that have been generated from those properties. The transformation of functional object and data properties has already been discussed above.

B. Conditional relationship types

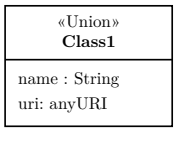
A conditional relationship type consists of a set of relationship types. An object must not appear more than once as a member of an instance of these relationship types. As an example, consider an address which might include either a visiting address or a postbox, but not both.

UML does not provide a special construct to defined such conditional relationship types. However, the ISO 19100 "UML profile" defines a new meta class *Union* with the desired semantics. Only one of a *Union*'s properties must be used. In an UML diagram an instance of a *Union* is depicted by adding the stereotype «Union» to a class symbol. However, this is not a "real" UML stereotype, as the semantics of the model element is changed. The set of the *Union*'s properties is defined by the set of class-dependent attributes.

Two different mappings have been developed to transform an instance of the meta class *Union* into OWL. The first mapping is only valid if the types of all class-dependent attribute are ether datatypes or classes— but not a mixture of both. In that case the attribute are transformed into object properties or data properties. By contrast, the second mapping also allows the transformation of a mixture of classes and datatypes. However, a larger number of axioms is required to reproduce the semantics.

Mapping 1: Let C be a class representing an instance of *Union*. Let $p_1 \dots p_n$ be its properties. It must be ensured that only a single property $p_x \in p_1 \dots p_n$ is specified for an individual. To achieve this, a helper property p_{Union} with domain C and the axioms $p_i \sqsubseteq p_{Union} \forall i \in 1..n$ are added to the ontology.

A `DataExactCardinality` axiom is used to restrict the number of p_{Union} properties for each individual of class C to exactly one. This prevents the setting of two or more different properties. Due to OWL's OWA it cannot be guaranteed that a property has been specified explicitly at all. This problem has been discussed above in the section on cardinality constraints.



Declaration(**Class**(:Class1))

Declaration(**DataProperty**(:name))
DataPropertyDomain(:name :Class1)
DataPropertyRange(:name xsd:string)
SubDataPropertyOf(:name :Class1_UnionProperty)

Declaration(**DataProperty**(:uri))
DataPropertyDomain(:uri :Class1)
DataPropertyRange(:uri xsd:anyURI)
SubDataPropertyOf(:uri :Class1_UnionProperty)

Declaration(**DataProperty**(:Class1_UnionProperty))
DataPropertyDomain(:Class1_UnionProperty :Class1)

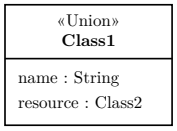
SubClassOf(
:Class1
DataExactCardinality(1 :Class1_UnionProperty))

Fig. 21. First approach for a transformation of an ISO 19100 Union.

Mapping 2: For each property $p_i \in p_1 \dots p_n$ of the *Union* an OWL helper class C_i is defined. By using a `Disjoint-Classes` axiom, it is stated that these n classes are disjoint in pairs. For each class, it is additionally stated that it is equivalent to the set of those individuals that are connected to exactly one individual or literal via p_i:

`EquivalentClasses(` C_i
`DataExactCardinality( 1` p_i `) )` bzw.
`EquivalentClasses(` C_i
`ObjectExactCardinality( 1` p_i `) )`

By using the first mapping only $(n+3)$ per UML property will be added to the ontology. The second mapping requires $(2n+1)$ additional axioms per property. Therefore, it is smart to use the first option if an instance of the meta class *Union* is composed exclusively of data types or classes, and to use the second option only when a mixture of both is present.



Declaration(**Class**(:Class1))
Declaration(**Class**(:Class2))

Declaration(**DataProperty**(:name))
DataPropertyDomain(:name :Class1)
DataPropertyRange(:name xsd:string)

Declaration(**ObjectProperty**(:resource))
ObjectPropertyDomain(:resource :Class1)
ObjectPropertyRange(:resource :Class2)

Declaration(**Class**(:Union_Class1_name))
EquivalentClasses(
:Union_Class1_name
DataExactCardinality(1 :name))

Declaration(**Class**(:Union_Class1_resource))
EquivalentClasses(
:Union_Class1_resource
ObjectExactCardinality(1 :resource :Class2))

DisjointClasses(:Union_Class1_name
:Union_Class1_resource)

Fig. 22. Second approach for a transformation of an ISO 19100 Union.

VII. Evaluation

This section deals with the question of whether the transformation rules presented in the previous sections are correct and—within their previously specified limits—complete. For this purpose, three different kind of analysis were conducted:

1) Coverage of the meta models
2) Analysis of individual transformation rules
3) Check the transformations automatically

Due to space limitations, only the third analysis is presented in detail.

One advantage of using QVT-R for the transformation is the generation of so-called "trace classes" and their instances during the execution of the transformation rules. Instances of the trace classes depend on the input models. In contrast, the trace classes itself are independent of the processed models. They are determined only by the transformation rules and the meta models. These recordings are used for tests 1) and 2).

Test 1 deals with the coverage of the meta models. It shows which part of the UML and OWL meta models is captured by the transformations at all. Further investigations are necessary, as an examination of the coverage of the meta-models is not sufficient for the evaluation of transformations. Even a complete coverage of the meta models cannot guarantee semantically preserving transformations. Such a complete coverage could actually be achieved by trivial and meaningless transformations. Consider the following example: all element types of the meta model M_1 are mapped to a single element type B of the target meta model M_2. Thus, a complete coverage is achieved for M_1. To also achieve complete coverage

for the element types of M_2, a second transformation rule is needed. For each instance of the element type A in meta model M_1 it creates an instance of every element type in meta model M_2. As a result a complete coverage of of M_2 is also achieved.

Therefore, it is necessary to investigate the transformation rules further. For this purpose, individual, mutually inverse transformation rules with their mutual dependencies and the dependencies to the meta element types are analysed in **test 2**. Rules that only artificially increase the coverage of the meta models would be detected by this test. Such a rule would attract attention because

- it is connected to instances of unusual many element types or
- it creates instances of unusual many element types.

Due to the complexity of a manual analysis of the transformation rules and the risk to overlook errors, an automatic verification of transformations is desirable. Such a verification is presented by **test 3**.

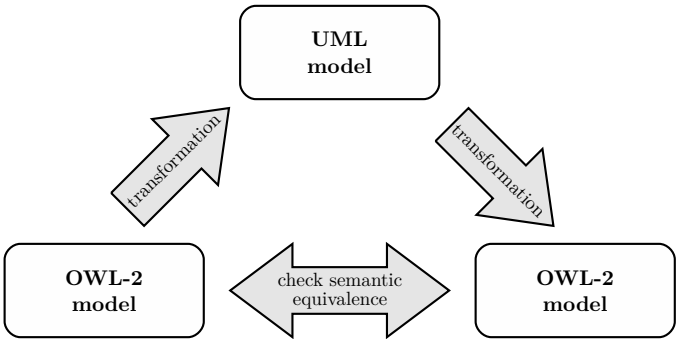


Fig. 23. Procedure for checking the correctness of the transformations.

Figure 23 shows a sketch of how to show the correctness of the transformation rules for certain parts of the meta model. The transformation rules are executed in both directions. After that input model and output model are compared *appropriately*.

It is advantageous to use an OWL ontology as input model (and thus also as output model). During the following comparison, available software tools such as reasoners can be used. The following shows how an " appropriate" comparison might look like.

Set U be the UML meta model, O the OWL meta model, u a model conforming to U, and o a model conforming the O. Let $\vec{T}_{UO}$ and $\vec{T}_{OU}$ be the transformations UML $\rightarrow$ OWL respective OWL $\rightarrow$ UML described above. Ideally, the consecutive execution of the transformations $o_2 = \vec{T}_{UO}(\vec{T}_{OU}(o_1))$ should create an ontology such that o_1 and o_2 are *semantically equivalent*.

What does *semantically equivalent* mean? It can be observed that there are models M_1 and M_2 with a different structure, for which each instance m that conforms to M_1 also conforms to M_2. Thus, the models describe the same (static) semantics. Similarly, models can be found that have the same informational content and differ only by the names of the elements. With a simple renaming, any instance that conforms to the first model, can be be transferred to an instance conforming to the second model. Overall, the checking for semantic equivalence can be cut down to the question of whether a *Total Ontology Mapping* [10] exists between both ontologies.

An ontology is a pair $O = (S, A)$ with S being the signature of the ontology and A its set of axioms. The signature describes the vocabulary used within the ontology. The set of axioms describes how the elements of S are put into relation.

A Total Ontology Mapping between an ontology $O_1 = (S_1, A_1)$ and an ontology $O_2 = (S_2, A_2)$ is a morphism $f : S_1 \rightarrow S_2$ that maps both signatures of the ontology in a way such that $A_2 \models f(A_1)$. All interpretations that satisfy the axioms of O_2 also satisfy the renamed axioms of O_1.

A. Computation of a Total Ontology Mapping for OWL-2

In OWL, the *signature* of the ontology is formed by the instances of the meta class `Entity`. The elements of the signature are divided into disjoint sets `Class`, `ObjectProperty`, `DataProperty`, `AnnotationProperty`, `Datatype`, `NamedIndividual`. Thus, the signature has the form$S = (S_C, S_{OP}, S_{DP}, S_{DT}, S_I)$. Annotations are ignored as they do not carry any semantic information. Since the sets are disjoint, the search for renaming can be restricted to one set. That significantly reduces the complexity of the search.

For simplicity, it is assumed that S_1 only contains elements that are used in A_1 and S_2 only contains elements that are used in A_2. Otherwise, unused items can be deleted without changing the statement of the axioms.

It is further assumed that the components of the signatures of the two ontologies have the same size: $|S_{X1}| = |S_{X2}|$, $X \in \{C, OP, DP, DT, I\}$. If this is not the case, an appropriate amount of previously unused elements is added to the smaller set.

In order to maintain a clear notation, only the subsets S_{C1} and $S_{C2}$for the classes are considered in details. The other four subsets S_{OP}, S_{DP},S_{DT}, and S_I are handled similarly.

The algorithm works as follows. Put the elements of S_{C1} and S_{C2} into an arbitrary order. The result are two ordered lists $S_{C1} = (c_1, \dots, c_n)$ and $S_{C2} = (d_1, \dots, d_n)$. For all possible permutations $\sigma_C : N \rightarrow N$, $N = \{1, \dots n\}$ check if every axiom $a \in f(A_1)$ can be inferred from A_2 with $f : (S_{C1}, \dots) \rightarrow (S_{C2}, \dots)$ and $\sigma_C(c_i) = d_i \ \forall i \in \{1, \dots n\}$. If such a permutation can be found, a Total Ontology Mapping between the ontologies O_1 and O_2 exists.

The procedure described in the previous paragraphs:

1) Apply the transformation $\vec{T}_{OU}$ to the input ontology o. The result is $m = \vec{T}_{OU}(o)$.
2) Apply the transformation $\vec{T}_{UO}$ to the UML model m. The result is $o' = \vec{T}_{UO}(m)$.
3) Use the algorithm to test if a Total Ontology Mapping between o and o' exists.
4) Use the algorithm to test if a Total Ontology Mapping between o' and o exists.

can be applied in instances of single meta classes or an arbitrary combination of meta class elements.

VIII. Related work

Two fundamentally different approaches for a transformation between UML and OWL-2 can be identified: XML-based transformations and transformations that are not based on XML.

A. Transformations based on XML

All XML-based approaches have a number of disadvantages in common. When working with documents containing a serialization of a model in concrete syntax only one model level is visible. Usually, only the names of meta models elements are available. The internal structure of the meta model and internal connections are inaccessible. XML-based transformations that use XML Metadata Interchange (XMI) documents and/or ontologies written in XML-based syntaxes of OWL and RDF lead to further problems. For example, the sequence of XML elements in two different serialization of one model can be almost completely different. It is easy to see that this leads to unnecessarily complex transformation rules. Besides others, [11][12] point out these problems as well.

Cranefield has addressed the connection between UML and ontologies in two articles: Cranefield and Purvis have examined how the UML and the Object Constraint Language (OCL) can be used to model ontologies [13] in general. The objective in this early work was not the transformation from UML to OWL, but rather the use of UML as an ontology modelling language. A transformation from UML class diagrams into Java code as well as into RDF-Schema is presented by Cranefield in a later article [14].

Falkovych presents a transformation of UML models into DAML + OIL (a predecessor of OWL) and RDFS using XSLT [15].

Gaševi Djuri et al. describe the transformation of a UML class diagram into an OWL ontology by using XSLT [16]. In the creation of the class model, a special UML profile "Ontology UML Profile"—defined by Djurić et al. in [17]—must be used.

Leinhos describes two variants for the transformation of UML models into OWL ontologies [18]. The UML models are serialized as XMI files. For the OWL ontologies the RDF/XML syntax is used. In one variant, specially constructed UML class diagrams are transformed into OWL ontologies. In the other variant, elements are added to the ontology that are not present in the original UML model and that do not match the semantics of the UML model.

B. Transformations not based on XML

Milanovic, Gasevic et al. describe the transformation of OCL rules into Semantic Web Rule Language (SWRL) rules, using the Atlas Transformation Language (ATL) [12][19]. It should be noted, that their approach is built upon meta models for OCL and SWRL. As the focus of our paper is on the transformation UML models and OWL-2 ontologies we consider the meta models for UML and OWL-2.

Hart et al. identify three groups of features. First, features that are more or less present in both languages. Second, features that are only available in UML and third, features only offered by OWL [20]. With respect to common features, examples are used to demonstrate how these examples would appear in both languages. This collection has been incorporated in some modified form as Chapter 16 of OMG's Ontology Definition Metamodel (ODM) specification [21]. However, only part of the model elements have been considered.

Höglund et al. use MOFScript to perform a transformation from UML to OWL-2 [22]. The aim of the work is the validation of meta models. Their transformation is a model-to-text transformation. Therefore, the OWL-2 meta model is not part of the transformation.

The idea of a model transformation between UML and OWL-2 was presented by the authors in [23] und [24]. However, these publications only present the idea and cover very few selected modelling elements. Very important to us is a very careful evaluation of the transformation rules which we presented in Section VII of this article.

IX. Conclusion and Future Work

In this paper, a systematic approach for an automatic transformation of conceptual models between the Model-Driven Architecture Technology Space and the Ontology Engineering Space Technology is presented. In contrast to previous works, an approach was chosen which abstracts from the concrete syntax or XML serialization and works on the level of the meta models of UML and OWL. As a result, it was possible to show independently of individual sample models, which model elements can be transformed and which can not be transformed.

It has been found that data models written in UML can be represented as OWL Ontologies quite well. Especially when certain restrictive rules—for example those the ISO 19100 family of standards specifies— are observed, the semantics of the data model will translate well. To be mentioned as problematic are: UML's possibility to restrict the visibility of model elements, abstract classes, certain kinds of generalization (non-overlapping but not complete), aggregation and composition (which can with minor exceptions be treated as ordinary relationship types), and the extension by stereotypes.

The different extent of the meta-models clearly suggests that OWL provides much more complex means of modelling already. The transformation of general ontologies in UML data models is not always possible. Particularly problematic is the definition of element types using nested class expressions as well as sufficient conditions. But even in these cases a transformation is often possible—e.g., cardinality constraints that appear as super-types. OWL constructs such as complementation and global properties can not be transformed in general. Only under the special condition that a single element type was defined as a super-type of all other element types, a transformation is still possible.

We applied the transformation technology presented in this article to improve the quality of historic statistical data, namely the so-called "Digital Reich Statistics" (1873-1883) of the German National Library of Economics. After digitization of the original data the library established a UML model for some economic data. The transformation of this model into an OWL-2 ontology allowed us to check the consistency of the UML model and the data.

In many cases modelling concepts can be implemented in UML data models by the use of Object Constraint Language (OCL) expressions. For example, OWL constructs such as the definition of element types via sufficient conditions can be realized using OCL expressions. As a MOF-compliant abstract syntax exists for OCL, that transformation could be carried out on meta model level—like the transformations described in this article. However, the additional use of OCL results in some difficulties, such as the question of whether even all atomic OCL expressions can be represented with OWL. It might be necessary to use rule languages, e.g., the Semantic Web Rule Language (SWRL) with its built-ins. However, this would make the transformation OWL $\rightarrow$ UML more complicated. OCL is a very rich language. By nesting expressions, arbitrarily complex OCL expressions can be generated. On the one hand, the transformation of these nested expressions becomes very complex. On the other hand, it is unclear whether these complex expressions can be expressed in an OWL ontology.

In the field of comprehensibility, UML is currently superior. If there was a corresponding intuitive graphical syntax for OWL with a selection of software tools for dealing with this syntax, it would certainly contribute to increase the use of OWL in the creation of conceptual models.

REFERENCES

[1] J. Zedlitz and N. Luttenberger, "Data types in UML and OWL-2," in *SEMAPRO 2013, The Seventh International Conference on Advances in Semantic Processing*, 2013, pp. 32–35.

[2] C. Eisenhut and T. Kutzner, *Vergleichende Untersuchungen zur Modellierung und Modelltransformation in der Region Bodensee im Kontext von INSPIRE*, München, 2010.

[3] M. Horridge and S. Bechhofer, "The OWL API: A Java API for OWL Ontologies," in *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*, R. Hoekstra and P. Patel-Schneider, Eds., 2009. [Online]. Available: http://ceur-ws.org/Vol-529/owled2009_submission_29.pdf

[4] A. Olivé, *Conceptual Modeling of Information Systems*, Berlin/Heidelberg/New York, 2007.

[5] M. Wahler, D. Basin, A. D. Brucker, and J. Koehler, "Efficient analysis of pattern-based constraint specifications," *Software and Systems Modeling*, vol. 9/2,, pp. S. 225–255, Heidelberg 2010. [Online]. Available: http://dx.doi.org/10.1007/s10270-009-0123-6

[6] OMG, "Unified Modeling Language, Superstructure Version 2.4," 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4/Superstructure

[7] XMLSchema-2, "XML Schema Part 2: Datatypes," 2004. [Online]. Available: http://www.w3.org/TR/xmlschema-2/

[8] W. Hesse and H. Mayr, "Modellierung in der softwaretechnik: eine bestandsaufnahme," *Informatik-Spektrum*, vol. 31/5,, pp. S. 377–393, Berlin/Heidelberg 2008.

[9] H. Balzert, *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*, Heidelberg,, 3. Auflage 2009, vol. 1.

[10] Y. Kalfoglou and M. Schorlemmer, "Ontology Mapping: The State of the Art" in *The Knowledge Engineering Review*, vol. 18/1, pp. S. 1–31, 2003. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2005/40

[11] K. Falkovych, M. Sabou, and H. Stuckenschmidt, "UML for the Semantic Web: Transformation-based Approaches," in *Knowledge Transformation for the Semantic Web*, vol. 95,, pp. S. 92–107, 2003.

[12] M. Milanović, D. Gašević, A. Guirca, G. Wagner, and V. Devedžić, "On Interchanging Between OWL/SWRL and UML/OCL," in *Proceedings of 6th Workshop on OCL for (Meta-) Models in Multiple Application Domains (OCLApps)*, 2006, pp. S. 81–95.

[13] S. Cranefield and M. Purvis, "UML as an Ontology Modelling Language," in *The Information Science Discussion Paper Series*, vol. 99/01, Dunedin 1999.

[14] S. Cranefield, "Networked Knowledge Representation and Exchange using UML and RDF," in *Journal of Digital information*, vol. 1/8, Austin 2001.

[15] K. Falkovych, "Ontology Extraction from UML Diagram,", Amsterdam, 2002.

[16] D. Gašević, D. Djuric, V. Devedzic, and V. Damjanovi, "Converting UML to OWL Ontologies," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. New York: ACM, 2004, pp. S. 488–489.

[17] D. Djurić, D. Gašević, V. Devedžić, and V. Damjanović, "A UML Profile for OWL Ontologies," in *Model Driven Architecture. European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers*, Berlin/Heidelberg, 2005, pp. S. 204–219. [Online]. Available: http://www.springerlink.com/content/49yb6365gymtryfg/

[18] S. Leinhos, "OWL Ontologieextraktion und -modellierung auf der Basis von UML Klassendiagrammen," Diplomarbeit, Universität der Bundeswehr München, München, 2006.

[19] M. Milanović, D. Gašević, A. Giurca, G. Wagner, and V. Devedžić, "Towards Sharing Rules Between OWL/SWRL and UML/OCL," in *Electronic Communications of the EASST Volume 5*,, 2006.

[20] L. Hart, P. Emery, B. Colomb, K. Raymond, S. Taraporewall a, D. Chang, Y. Ye, E. Kendall, and M. Dutra, "OWL Full and UML 2.0 Compared," 2004. [Online]. Available: http://www.omg.org/docs/ontology/04-03-01.pdf

[21] OMG, "Ontology Definition Metamodel," Object Management Group, 2009. [Online]. Available: http://www.omg.org/spec/ODM/1.0/

[22] S. Höglund, A. Khan, Y. Liu, and I. Porres, "Representing and Validating Metamodels using the Web Ontology Language OWL 2. TUCS Technical Report No. 973," Turku 2010. [Online]. Available: http://tucs.fi/publications/attachment.php?fname=TR973.full.pdf

[23] J. Zedlitz, J. Jörke, and N. Luttenberger, "From UML to OWL 2," in *Proceedings of Knowledge Technology Week 2011*, D. Lukose, A. R. Ahmad, and A. Suliman, Eds., Berlin/Heidelberg, 2012, pp. p. 154–163.

[24] J. Zedlitz and N. Luttenberger, "Transforming Between UML Conceptual Models and OWL 2 Ontologies," in *Proceedings of the Terra Cognita Workshop on Foundations, Technologies and Applications of the Geospatial Web, in conjunction with the 11th International Semantic Web Conference (ISWC 2012)*, D. Kolas, M. Perry, R. Grütter, and M. Koubarakis, Eds., 2012, pp. p. 15–26. [Online]. Available: http://ceur-ws.org/Vol-901/paper2.pdf