ODEDialect: a set of declarative languages for implementing ontology translation systems

Oscar Corcho, Asunción Gómez-Pérez

Ontological Engineering Group. Departamento de Inteligencia Artificial.
Facultad de Informática. Universidad Politécnica de Madrid.
Campus de Montegancedo, s/n. 28660 Boadilla del Monte. Madrid. (Spain)
{ocorcho,asun}@fi.upm.es

Abstract. The implementation of ontology translation systems is a complex task where many types of translation decisions have to be made. These decisions are usually hidden inside the systems source code. In order to allow building, maintaining and understanding more easily ontology translation systems, we propose ODEDialect, a set of languages to express translation decisions declaratively and at different layers: lexical, syntax, semantic, and pragmatic. This paper describes the three languages that comprise ODEDialect: ODELex, ODESyntax, and ODESem, which express transformations in the lexical, syntax, and semantic and pragmatic layers, respectively.

1 Introduction

Ontologies can be implemented in varied ontology languages (DAML+OIL, FLogic, KIF, LOOM, OCML, OIL, Ontolingua, OWL, RDF, RDF Schema, etc.) and ontology tools (KAON, OilEd, OntoEdit, the Ontolingua Server, OntoSaurus, Protégé-2000, WebODE, and WebOnto). For a detailed comparison of these languages and tools we refer to [10]. Languages and tools have their own expressiveness and reasoning capabilities, and are based on different knowledge representation (KR) paradigms and combinations of them (frames, first order logic, description logic, semantic networks, etc.). Besides, languages have different syntaxes, and tools have different APIs.

There are important connections and implications between the knowledge modelling components used to build an ontology in such languages and tools, and the KR paradigms used to represent formally such components. With frames and first order logic, the knowledge components commonly used are [11]: classes, relations, functions, formal axioms, and instances; with description logics, they are [2]: concepts, roles, and individuals; with semantic networks, they are: nodes and arcs between nodes; etc.

The **ontology translation problem** [11] appears when we decide to reuse an ontology (or part of an ontology) with a tool or language different from those where the ontology is available. If we force each ontology-based system developer, individually, to commit to the task of translating and incorporating to their systems the ontologies that they need, they will require both a lot of effort and a lot of time to achieve their objectives [19]. Therefore, ontology reuse will be highly boosted as long as we provide ontology translation services among those languages and/or tools.

Since ontology tools and languages have different expressiveness and reasoning capabilities, **translations between them are not straightforward nor easily reusable**. They normally require to take many **decisions at different levels**, which range from low layers (i.e., how to transform a concept name identifier from one format to the another) to higher layers (i.e., how to transform a ternary relation among concepts to a format that only allows representing binary relations between concepts).

However, current ontology translation systems do not usually take into account such a layered structure of translation decisions. Furthermore, in these systems **translation decisions are usually hidden inside their programming code**. Both aspects make it difficult to understand how ontology translation systems work.

To contribute solving this problem, in this paper we propose ODEDialect, a set of languages that allow expressing declaratively translation decisions at different levels: lexical, syntax, semantics, and pragmatics (this layered structure is based on the theory of signs [18]). This paper describes the three languages that comprise ODEDialect: ODELex, which expresses transformations in the lexical layer; ODESyntax, which expresses transformations in the syntax layer; and ODESem, which expresses transformations in the semantic and pragmatic layers.

This paper is structured as follows: section 2 describes the four layers where ontology translation problems may appear, with examples of how transformations have to be made at each layer. Section 3 describes the three languages that comprise ODEDialect, with their grammars and examples of their use. Section 4 presents the main conclusions of our work and future work. Section 5 presents related work on ontology translation.

2 Ontology translation layers

The layered classification of translation decisions presented in this paper is based on existing work on formal languages and on the theory of signs [18]. Such works consider the existence of several levels in the definition of a language: syntax (related to how the language symbols are structured), semantics (related to the meaning of those structured symbols), and pragmatics (related to the intended meaning of the symbols, that is, how symbols are interpreted or used).

In the context of semantic interoperability, some authors have proposed classifications of the problems to be faced when managing different ontologies in, possibly, different formats. We enumerate only the ones that are due to differences between the source and target formats[1]. Euzenat [7] distinguishes the following non-strict levels of language interoperability: encoding, lexical, syntactic, semantic, and semiotic. Chalupsky [5] distinguishes two layers: syntax and expressivity (aka semantics). Klein [14] distinguishes four levels: syntax, logical representation, semantics of primitives, and language expressivity, where the last three levels correspond to the semantic layer identified in the others. Figure 1 shows the relationship between these classifications.

The layers described in this section are mainly based on Euzenat's classification. This classification is the only one in the context of semantic interoperability that deals

[1] Semantic interoperability problems do not only appear because ontologies are available in different formats, but also because of their content, their ontological commitments, etc. We only focus on problems related exclusively to differences among ontology languages or tools.

with pragmatics (the term "semiotics" is used in this case to refer to a subset of pragmatics). However, we consider that the lexical and encoding layers do not need to be splitted when dealing with ontologies, so that they form a unique lexical layer.

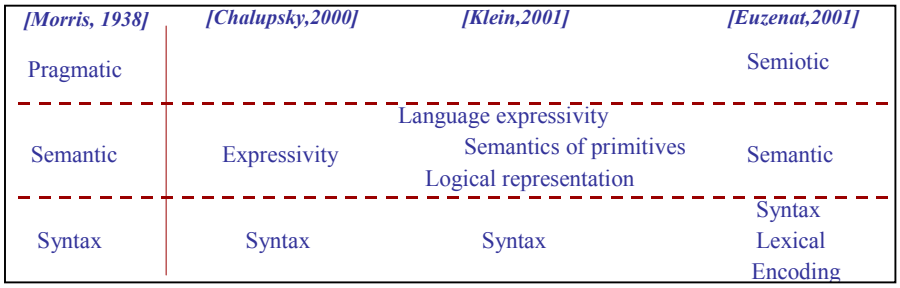

Fig 1. Relationships between classifications of semantic interoperability problems.

2.1 Lexical problems

The lexical layer deals with the "ability to segment the representation in characters and words (or symbols)" [7]. Different languages and tools normally use different character sets and grammars to generate their terminal symbols. Therefore, in this layer we deal with transformations of ontology component identifiers, of pieces of text used for natural language documentation purposes, and of values.

Lexical transformations mainly consist in replacing non-allowed characters by others (e.g., class identifiers in Protégé-2000 can contain blank spaces – for instance, *Travel Agency* –, and this is not possible in Ontolingua – it would be transformed to *Travel-Agency* –), or replacing identifiers that are reserved keywords in a format to other ones that are not reserved keywords (e.g., if an OWL ontology contains the class *:THING*, it cannot be transformed to Protégé-2000).

Other sources of problems in lexical transformations are related to the scope of ontology component identifiers and the restrictions related to overlapping identifiers. These problems appear when, in the source format, a component is defined inside the scope of another, and hence its identifier is local to the latter, and in the target format the correspondent component has a global scope. Therefore there could be clashes of identifiers in case that in the source format two components have the same identifier.

2.2 Syntax problems

This layer deals with the "ability to structure the representation in structured sentences, formulas or assertions" [7]. Ontology components in each language or tool are defined with different grammars. Hence, this translation layer deals with the problems related to how symbols are structured in the source and target formats, taking into account their derivation rules for ontology components.

The following types of transformations are included in this layer: transformations of ontology component definitions according to the grammars of the source and target formats (e.g., the grammar to define a concept in Ontolingua is different than that of OCML) and transformations of datatypes (e.g., the datatype "date" in WebODE must be transformed to the datatype "&xsd;date" in OWL).

With regard to the syntax differences between formats, we can distinguish basically three groups of languages and tools: Lisp-based formats (usual in many classical languages and tools, such as Ontolingua, LOOM, or OCML), XML-based formats (usual in ontology markup languages), and ad-hoc text formats (like in FLogic). Besides, several languages and tools provide ontology management APIs in different programming languages, such as Java, C++, Lisp, etc., which could be considered as another form of syntax for representing ontologies.

With regard to datatypes, we can distinguish basically two groups of languages and tools: those with their own datatypes (integer, float, number, string, etc.), and those that allow using XML Schema datatypes (usually ontology markup languages).

2.3 Semantic problems

The semantic layer deals with the "ability to construct the propositional meaning of the representation" [7]. Different ontology languages and tools can be based on the same KR paradigm, on different KR paradigms (frames, semantic networks, first order logic, conceptual graphs, etc.) or on combinations of them.

In this layer we deal not only with simple transformations (e.g., FLogic concepts are transformed into Ontolingua and OWL classes), but also with complex transformations of expressions that are usually related to the fact that the source and target formats are based on different KR paradigms (e.g., WebODE disjoint decompositions are transformed into subclass-of relationships and PAL constraints in Protégé-2000, FLogic instance attributes attached to a concept are transformed into datatype properties in OWL and unnamed property restrictions for the class).

Most of the work on ontology translation done so far has been devoted to solving the problems that arise in this layer. For example, there are several formal, semi-formal, and informal methods for comparing ontology languages and ontology tools' knowledge models ([1], [4], [8], [6], [15], etc.), which aim at helping to decide whether two formats have the same expressiveness or not. These approaches allow deciding whether knowledge can be preserved in the transformation and whether the reasoning mechanisms of the source and target formats will infer the same knowledge.

Basically, these studies analyse the expressiveness (and, in some cases, the reasoning mechanisms) of the source and target formats, so that we can know which types of components can be translated directly from a format to another, which ones can be expressed using other types of components from the target format, which ones cannot be expressed in the target format, and which ones can be expressed, although losing part of the knowledge represented in the source format.

In summary, the problems found in this layer are mainly related to the different KR formalisms in which the source and target formats are based. This does not mean that translating between two formats based on the same KR formalism is straightforward. There may be differences in the types of ontology components that can be represented in each of them. This is specially important with DL languages, since many different combinations of primitives can be used in each language, and hence many possibilities exist in the transformations between them, as shown in [8]. However, the most interesting results appear when the KR formalisms are different.

2.4 Pragmatic problems

This layer deals with the "ability to construct the pragmatic meaning of the representation (its meaning in context)". In this layer we deal with transformations to be made in the resulting ontology so that human users and ontology-based applications will notice as less differences as possible with respect to the ontology in the original format, either in one-direction transformations or in cyclic transformations.

Transformations in this layer require, among other, the following: adding special labels to ontology components so as to preserve their original identifier in the source format (e.g., adding an own slot to all the Protégé-2000 classes obtained when transforming an ontology from another tool or language); transforming sets of expressions into more legible syntactic constructs in the target format (e.g., transforming a set of Protégé-2000 PAL constraints into a single class description); somehow "hiding" completely or partially some ontology components that were not defined in the source ontology, but which have been created as part of the transformations (such as the anonymous classes that are usually created when transforming between a DL-based format to a frame-based format); etc.

2.5 Relationships between ontology translation layers

Figure 2 shows an example of a transformation from the ontology platform WebODE to the language OWL DL. In this example, we transform two ad hoc relations with the same name (*usesTransportMean*) and with different domains and ranges (a *flight* uses an *airTransportMean* and a *cityBus* uses a *bus*). In OWL DL the scope of an object property is global to the ontology, and so we cannot define two different object properties with the same name. The example shows that translation decisions have to be made at all layers, and that the decisions taken at one layer can affect the decisions to be made at the others, hence showing the complexity of this task.

Option 1 is driven by semantics: to preserve semantics in the transformation, two different object properties, with different identifiers, are defined. Option 2 is driven by pragmatics: only one object property is defined from both ad hoc relations, since we assume that they refer to the same meaning, but some knowledge is lost in the transformation (the one related to the object property domain and range). Finally, option 3 is also driven by pragmatics, with more care on the semantics: again, only one object property is defined, its domain and range is more restricted than in option 2, although we still lose the exact correspondence between each domain and range.

3 Description of ODEDialect

Taking into account the preceding transformation layers and the main characteristics of each of them, and the fact that implementing ontology translation decisions is difficult, we propose a set of languages to express such transformations declaratively. This set of languages will express transformations in the lexical layer (ODELex), in the syntax layer (ODESyntax), and in the semantic and pragmatic layers (ODESem). ODESem deals with problems in both the semantic and pragmatic layers because they require similar types of transformations, and thus they can be implemented similarly.

Fig 2. Example of translation decisions to be taken at several layers.

In the following sections we present the main features of each language, with examples extracted from the ontology translation system from WebODE to OWL DL.

3.1 ODELex: declarative specification of transformations at the lexical layer

Problems in the lexical layer are normally easy to handle, because it is usually enough to take into account the rules and conventions for creating identifiers and texts in the source and target formats. ODELex allows specifying all the transformations to be made at this layer. This language is similar to lex [16, 17], JLex [3], and other widely-used lexical analysers for building compilers. While these languages are aimed at building compilers, ODELex is optimised for building ontology translation systems, by restricting some of their primitives and by providing specific ones related to the construction of ontology translation systems, as described below.

An ODELex specification is composed of three parts, which contain: user code, declarations, and lexical rules. Java-style comments can be added at the beginning of the document, or inside the other parts. We will now describe each of these parts:

User code. The first part of an ODELex document contains the user code, where users may include any Java functions used in the rest of the specification plus any Java import statements needed for these functions. These functions usually implement complex transformations to be made to ontology component identifiers or pieces of text, or they are used as a kind of macro definitions for a sequence of transformations that have to be performed using standard functions from the Java API. In the following example, we have defined a Java function (*convertToURI*) that transforms a string value into a corresponding string value that is a valid URI:

```
import java.net.*;
private String convertToURI (String id){
       URI id_URI = new URI(URLEncoder.encode(id,"ISO-8859-1"))
       return id_URI.toString();
}
```

These definitions are copied verbatim into the Java source file generated for dealing with lexical transformations. In this part there is also the possibility of referencing transformation functions from a lexer created with typical lexical analysis tools.

Declarations. This part of an ODELex specification contains the declaration of the ontology components (from the source and target formats) dealt with by the lexical transformation tool. It also contains the declaration of the ontology components of the target format whose identifiers cannot overlap (which means that they cannot share the same identifiers because they share the same scope).

In the example below, the declaration part defines some of the WebODE ontology components (concepts, instance attributes, and ad hoc relations) and attribute values. With respect to OWL, it defines four ontology components (classes, object properties, datatype properties, and instances) and datatype values. It also states that the sets of identifiers of the four ontology components must be disjoint (they cannot overlap).

Three WebODE components are not first class citizens of the ontology, since they are defined inside the scope of others; instance and class attributes are defined inside the scope of a concept, and ad hoc relations are defined inside the scope of two concepts.

Finally, the *%transient* keyword states that once the transformation of a component has been performed, it is not interesting to store the result of such transformation. In the example, this happens with attribute values in both formats.

```
WebODE.Concept
WebODE.InstanceAttribute %scope (WebODE.Concept)
WebODE.Relation %scope (WebODE.Concept,WebODE.Concept)
WebODE.Value %transient

OWL.Class
OWL.ObjectProperty
OWL.DatatypeProperty
OWL.Instance
OWL.DatatypeValue %transient
no-overlap (OWL.Class, OWL.ObjectProperty,OWL.DatatypeProperty,
            OWL.Instance)
```

Lexical transformation rules. This part of an ODELex specification contains the actual transformations to be performed to each ontology component of the source format in order to obtain its correspondence in the target format. For each component in the lexical rule header, the following information must be specified:

- INIT: the initial transformation to be performed. For instance, in the example below we propose to transform the identifier of a WebODE ad hoc relation to an OWL ObjectProperty by converting the identifier to a URI, with the function *convertToURI* specified above.
- TABLE: if the component is not transient, it specifies how to store the result of the transformation. In the example below we propose to store the ad hoc relation identifier, and its two associated concept identifiers in the table *WebODE.Relation*, and the corresponding identifier obtained from the transformation in the table *OWL.ObjectProperty*, maintaining the corresponding links to each other.
- REPEATED: if the component is not transient and cannot be repeated under certain circumstances, it specifies an alternative transformation to be performed. For instance, if after the transformation the OWL object property identifier already existed as an OWL object property identifier and if the WebODE ad hoc relation with the same ad hoc relation identifier and domain concept already existed as well, then we propose to maintain the identifier to be provided for the transformed object property (and we obtain it with the predefined function *GET*). The same applies if the same ad hoc relation identifier and range concept already existed. In another situation, a new identifier is created by adding the character "1" to the current transformed identifier.
- OVERLAP: if the component is not transient and there cannot be overlaps in the target format, it specifies the transformation to be performed to the identifier already generated. This is repeated until there is no overlap. In the example, we add a number to the OWL object property identifier until there is no collision with other class, datatype property or instance identifiers (not object properties).

```
%WebODE. Relation IDENTIFIER
        WebODE.Concept IDENTIFIER WebODE.Concept IDENTIFIER
  /* The 2nd and 3rd identifiers are the domain and range */
  INIT: {$1=convertToURI(%1)}
  TABLE:{([WebODE.Relation,%1,%2,%3],[OWL.ObjectProperty,$1])}
  REPEATED:
    {[WebODE.Relation,%1,%2,_] ==>
              {$1=GET([WebODE.Relation,%1,%2,_])},
```

```
    [WebODE.Relation,%1,_,%3] ==>
            {$1=GET([WebODE.Relation,%1,_,%3])},
    default ==> {$1=addNumber($1)}}
OVERLAP: {$1=addNumber($1)}
```

3.2 ODESyntax: declarative specification of transformations at the syntax layer

Our approach assumes that the source and the target formats of the transformations have their own Java APIs defined. Thus the transformations expressed in this layer are simple, as in the lexical layer. With regard to the source format, in this layer we describe the correspondence between each component and its accessors (either for the component itself, for all the components of a specific type, or for pieces of information of each component). With regard to the target, in this layer we describe the correspondence between each component and its constructors, adding and updating methods, as well as the accessors that might be needed. In this layer we also specify correspondences between attribute datatypes of the source and target formats.

As with the lexical layer, in the syntactic layer we propose the use of the language ODESyntax, which allows specifying all the correspondences outlined above. This language is also based on another one available for building compilers, such as yacc [13, 17], JCup [12], etc. A specification in ODESyntax is divided in five parts: user code, declarations, accessors, constructor and updates, and datatype transformations.

User code. As with ODELex, the first part of an ODESyntax specification contains the user code, where all the auxiliary Java functions to be used in the rest of the specification are included. This code is copied verbatim into the Java source file generated for dealing with syntax transformations. In this part we can also refer to transformation functions created by typical syntax analysis tools.

Declarations. This part of ODESyntax declares the list of ontology components dealt with in the specification. This list does not need to contain the same components than the ODELex specification, since it has a different purpose (syntactic transformations instead of lexical ones). Besides, it does not have to consider whether a component is transient or not, nor whether there can be overlap or not with other components, since these problems are already solved by the lexical layer specification.

This component list is usually based on the knowledge models of the source and target formats, and usually corresponds as well to their APIs, because there is usually a strong correspondence between the format API and its knowledge model (either of the source or of the target format).

The declaration part also includes a list of namespaces, used to abbreviate long Java packages in the rest of the specification. To refer to these namespaces, the corresponding namespace identifier must be placed between square brackets (e.g., *[ode]* to refer to *es.upm.fi.dia.ontology.webode.service.*).

For example, the example below defines two namespaces (*ode* and *jenaOnt*), which correspond to the packages where the knowledge models of the source and target formats are defined. It also defines some of the ontology components used for the syntactic transformations, together with their scope, in case that they depend on other components, and with their corresponding Java class or interface.

```
%NAMESPACE ode es.upm.fi.dia.ontology.webode.service.;
%NAMESPACE jenaOnt com.hp.hpl.jena.ontology.;
/* WebODE components */
WebODE.Concept   %scope (WebODE.Ontology)     :[ode]Concept
WebODE.InstanceAttribute %scope (WebODE.Concept,WebODE.Ontology)
             :[ode]InstanceAttributeDescriptor
WebODE.Relation %scope (WebODE.Concept,WebODE.Concept)
             :[ode]TermRelation
/* OWL components */
OWL.Ontology              :[jenaOnt]Ontology
OWL.Class                 :[jenaOnt]OntClass
OWL.ObjectProperty        :[jenaOnt]ObjectProperty
OWL.DatatypeProperty      :[jenaOnt]DatatypeProperty
OWL.Instance              :[jenaOnt]Individual
```

Accessor methods. For each ontology component declared previously, this part of ODESyntax may specify which methods are used for the following purposes:

- To access all (*ALL*) the ontology components of a specific type (e.g., a method that retrieves all the concepts of an ontology).
- To access a specific ontology component (*INDIVIDUAL*) of a specific type (e.g., a method that retrieves a concept of an ontology, given its identifier).
- To access different pieces of information (*INFORMATION*) of the ontology component (e.g., methods or properties to access a concept identifier, a concept description, the superclasses of a concept, etc.).

For each method in the first and second groups, we indicate a number that identifies the method (there can be different methods with the same purpose), the method name and parameters, and the object that it returns. For the third group, we specify the piece of information's identifier (which determines how we will access this information from the semantic and pragmatic layers), the property or method to be used to access to that information given a specific object of that type, and the datatype.

The example below shows the declaration corresponding to an ontology component of the source format: a WebODE instance attribute. It shows that instance attributes are defined inside the scope of an ontology and of a concept. There are two methods that allow accessing all the instance attributes of a concept in an ontology: the first one is used to access the instance attributes that are defined explicitly in that concept, and the second one returns also those inherited through the concept taxonomy. Both of them return an array of objects of the class *InstanceAttributeDescriptor*.

There is one method to get a specific instance attribute, given the ontology name, the concept name and the attribute name.

Finally, the following information can be accessed from an instance attribute: the concept to which it belongs, the attribute name, description, type, maximum and minimum cardinality, maximum and minimum value, and explicit values. They are accessed using properties of the Java class *InstanceAttributeDescriptor*, except for the attribute type, which is accessed with an ad hoc function defined in the user code part.

```
%WebODE.InstanceAttribute IDENTIFIER
    WebODE.Concept IDENTIFIER WebODE.Ontology IDENTIFIER
ALL: {1:getInstanceAttributes(%3,%2)
                  :[ode]InstanceAttributeDescriptor[];
      //gets only inst attributes defined locally to the concept
```

```
      2: getInstanceAttributes(%3,%2,true)
                    :[ode]InstanceAttributeDescriptor[]
      //gets also inherited inst attributes
  }
INDIVIDUAL: {1:getInstanceAttribute(%3,%2,%1)
                    :[ode]InstanceAttributeDescriptor}
INFORMATION:
 {concept     :termName                          :String;
  name        :name                              :String;
  description:description                         :String;
  type        :getValueTypeName(valueType)       :String;
  maxCard     :maxCardinality                    :int;
  minCard     :minCardinality                    :int;
  maxValue    :maxValue                          :float;
  minValue    :minValue                          :float;
  values      :values                            :String[]}
```

If a specific method or property that we want to specify in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file that will extend the current API.

Constructor and update methods. For each ontology component of the target format that has been declared in the declaration part, this part of an ODESyntax specification may declare the methods that will be used for the following purposes:

- To create the ontology component in the target ontology (e.g., a constructor or a method that creates an OWL class). We use the keyword *CREATE*.
- To remove the ontology component from the target ontology (e.g., a method that removes a class from an OWL ontology). We use *REMOVE*.
- To add information about the component (e.g., the method or property used to add information about the class documentation, the superclasses, etc.). We use *ADD*.
- To remove completely some information of the ontology component (e.g., the method or property to be used to remove all the class documentations, all the class superclasses, etc.). We use *REMOVEALL*.
- To remove a value from a specific piece of information of the ontology component (e.g., the method or property to be used to remove a specific class documentation, a specific class superclass, etc.). We use *REMOVEINDIVIDUAL*.

For each method in the first and second groups, we indicate a number that identifies the method, the method name and parameters. For the rest of groups, we specify the piece of information's identifier (it determines how we will access this information from the semantic and pragmatic layers), and the property or method to be used to add, remove all, and remove one of the values of that piece of information, respectively. Not all the pieces of information must be specified in all cases, but only those used by the transformations in the semantic and pragmatic layers.

The example below declares an OWL class. There is one method to create classes in the target ontology, *createClass*, which receives as an input the identifier of the class. It also contains other methods to add a natural language description to the class and to remove all the descriptions, and others to add a superclass, to remove all the class superclasses, and to remove a specific superclass. The %1 parameter refers to the OWL class identifier, and the $1 in the parameter means that the value for this parameter (the actual description and the class identifier) will be provided by the

semantic or pragmatic layers in the first order. We will see how to define this information in the semantic and pragmatic transformation layers.

```
%OWL.Class IDENTIFIER
CREATE:       {1:createClass(%1)}
ADD:          {description :addComment(%1,$1);
               subclassOf  :addSuperClass(createClass($1))}
REMOVEALL:    {description :removeAllComments(%1);
               subclassOf  :removeAllSuperclasses(%1)}
REMOVEINDIVIDUAL:{subclassOf :removeSuperClass(createClass($1))}
```

As in other parts of ODESyntax, if a specific method or property used in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file, extending the API.
Datatype transformations. This part contains the transformations to be made to the attribute datatypes of the source format so as to transform them to the target format.
The example below shows how to specify these transformations. The first column, specifies the attribute name in the source format. The second column specifies its correspondence in the target format (in this example, all of them are XML Schema datatypes). The transformations will be checked sequentially, according to the order specified in this table. Besides, the "default" keyword can be used at the last line to specify any other kind of datatype that could be found.

```
"boolean":  "http://www.w3.org/2001/XMLSchema#boolean";
"string":   "http://www.w3.org/2001/XMLSchema#string";
"URL":      "http://www.w3.org/2001/XMLSchema#anyURI";
default %1: %1;
```

3.3 ODESem: declarative specification of transformations at the semantic and pragmatic layers

The previous sections have described how to specify declaratively the transformations in the lexical and syntactic layers. The main objective of both transformation layers is to abstract the low-level details of the source and target formats (their syntax specific features, the restrictions and naming conventions of ontology component identifiers, etc.), so as to allow specifying the semantic and pragmatic transformations – which are the most important and complex – at a higher abstraction level.

These translation decisions will be specified with the same declarative language: ODESem. An ODESem specification is divided into three parts: user defined code, declarations and semantic and pragmatic rules.

User code. This part contains auxiliary Java code used in the rest of the specification.

Semantic and pragmatic transformation rule declarations and processing order. This part of ODESem contains the declaration of the transformation rules to be defined later, together with the order in which they have to be processed. Unlike in ODELex and ODESyntax, the order of the definitions is relevant, since they define the order in which these transformation rules will be processed.

The example below shows the rule declarations for the WebODE export service to OWL DL. The rules 1 to 6 are in charge of the semantic and pragmatic transformations of WebODE components: it adds ontology information (ontology container), classes, object properties, disjoint and exhaustive decompositions, and instances. Finally, three pragmatic transformations remove redundant domains in OWL datatype properties, and redundant domains and ranges in object properties.

```
1: %AddOntologyContainer;
2: %AddClasses;
3: %AddObjectProperties;
4: %AddDisjointDecompositions;
5: %AddExhaustiveDecompositions;
6: %AddInstances;
7: %PostProcessing_RemoveDPRedundantDomains;
8: %PostProcessing_RemoveOPRedundantDomains;
9: %PostProcessing_RemoveOPRedundantRanges;
```

Semantic and pragmatic transformation rules. This part of the specification contains at least the rules declared previously, plus any other auxiliary rules that can be called from these ones. A transformation rule is defined with two parts:

- The LHS (Left Hand Side) of the rule contains the information needed to trigger the rule, which can be either the source ontology component to be transformed or a target component to be modified. In both cases, the LHS contains the ontology component type, as defined with ODESyntax, and an identifier that will be used to refer to the specific component in the RHS (Right Hand Side) of the rule. If no information is needed to trigger the rule, the keyword *NULL* must be used.
- The RHS of the rule contains the sequence of actions to be performed in order to obtain the corresponding ontology component(s) in the target format.

Three groups of actions can be performed: actions that create new ontology components, and add or remove components or information; actions that specify the control flow of the translation system; and actions that throw error messages, assign values to variables or call other functions, either predefined or defined by the user.

The following primitives can be used:

- CREATE. It creates (and returns) an ontology component in the target ontology. This action receives as input parameters the ontology component type to be created, the number of the specific constructor to be used, and the rest of parameters needed to create the ontology component.
- ADD. It adds one or several values to a specific property of an ontology component of the target ontology. The ontology component, the property and the value(s) are specified as input parameters.
- REMOVE. It removes a value from a specific property of an ontology component of the target ontology. The ontology component, the property and the value to be removed are specified as input parameters.
- REMOVEALL. It removes all the values from a specific property of an ontology component of the target ontology. The ontology component and the property are specified as input parameters.

The following control flow structures can be used:

- EXEC. It executes a rule with the set of parameters that match its antecedent.

- If *condition* {*actions*} else {*actions*}. It specifies the set of actions to be performed if the condition specified is evaluated as true, and, optionally, the set of actions to be performed if the condition specified is evaluated as false.
- ForEach *variable* IN *set_variable* {*actions*}. It specifies the set of actions to be performed for each ontology component inside the multiple-valued variable.

Finally, variable assignments can be performed with *var* = *value*, and the predefined functions *GETCOMPONENT* and *GETALLCOMPONENTS* can be used to obtain a specific component or all the components of the source or the target formats (with the parameters specified in the ODESyntax specification). The *ERROR* function can be used in cases where the options that allow executing it are not allowed.

Below we provide the code of a transformation rule that transforms WebODE concepts into OWL classes. For each WebODE concept, it creates an OWL class with the concept name, it adds a natural language description, if it exists, and it states that it is a subclass of the parent concepts. Finally, for each instance attribute of the concept, the rule *ADDInstanceAttributes* is triggered.

```
%AddClasses
WebODE.Concept concept -->
  {C = CREATE(OWL.Class,1,concept.name);
   if (concept.description != null)
        ADD(C,description,concept.description);
   ADD(C,subClassOf,concept.parentConcepts);
   forEach ia IN concept.instanceAttributes
       EXEC(%AddInstanceAttributes,WebODE.InstanceAttribute,ia);
  }
```

Finally, we show a transformation rule that corresponds to the pragmatic post-processing of datatype properties so as to remove the redundant domains. This rule checks whether the datatype property domain is a union of classes. In that case, if any of the classes in that union is already a subclass of any of the other concepts in the union, then the class can be removed, since it is already considered in the domain.

```
%PostProcessing_RemoveDPRedundantDomains
OWL.DatatypeProperty P -->
 {forEach U in P.domain {
    if (U.isUnionOf)
      forEach D in U.classes
        forEach E in U.classes
          if (E.isSubclassOf(D)) REMOVE(U,class,E)
 }
```

4 Conclusions and future work

This paper describes ODEDialect, a set of languages – ODELex, ODESyntax, and ODESem – that allow expressing declaratively ontology translation systems. With ODEDialect, translation decisions can be made at four different layers (lexical, syntax, semantic, and pragmatic), based on existing classifications of semantic interoperability and on the theory of signs. We assume that this layered approach makes it easier to construct and maintain ontology translation systems. The types of transformations at each layer are different, and thus the languages for expressing transformations at each layer are different as well, except for the semantic and pragmatic ones.

One of the design issues considered for the creation of these languages has been that they should allow expressing declaratively most of the transformations to be made at each layer. However, at the same time it should be flexible enough to allow representing any type of transformation required. We have achieved these objectives by proposing a large set of primitives in these languages and by allowing the inclusion of user-defined code in the general purpose programming language Java.

The ODEDialect language has been successfully used to specify translation decisions of the import and export services of the WebODE ontology engineering workbench to and from OWL DL, RDF(S), and Protégé-2000, although these ontology translation systems have not been generated automatically but manually from their ODEDialect specifications. This experience have given us enough insight to be able to build a compiler that automates these tasks in the near future.

5 Related work

Two systems allow creating ontology translation systems declaratively: Transmorpher and OntoMorph. *Transmorpher* [9] is aimed at facilitating the definition and processing of complex transformations in XML, using XSLT. Among other domains, this tool has been used to transform ontologies between DL languages, using DLML[2] and giving support to the ontology translation approach "family of ontology languages" [8]. Its main limitation is that it only deals with problems in the lexical and syntactic layers, and can only be applied to languages with XML syntax.

OntoMorph [5] allows specifying transformations between the source and the target formats by means of pattern-based transformation rules. Transformations are performed in two phases: syntactic rewriting and semantic rewriting. The last one needs the ontology or part of it translated into PowerLoom, so that this KR system can be used for certain kinds of reasoning (e.g., discovering whether a class is subclass of another). Since this tool is based on PowerLoom (and consequently on Lisp), it cannot handle easily all the problems that may appear in the lexical and syntax layers.

ODEDialect improves the support given by these systems by specifying transformations at more levels, so that ontology translation systems are easier to create, understand and maintain. Besides, ODEDialect can be applied to a wider range of formats, not necessarily based on XML or Lisp. On the contrary, the previous systems create ontology translation systems automatically from the specifications of ontology translation decisions. This is not possible yet in ODEDialect.

Acknowledgements

This work is supported by the IST project Esperonto (IST-2001-34373).

References

1. Baader F (1996) A Formal Definition for the Expressive Power of Terminological Knowledge Representation Languages. Journal of Logic and Computation 6(1):33–54

[2] *Description Logic Markup Language.* http://co4.inrialpes.fr/xml/dlml/

2. Baader F, McGuinness D, Nardi D, Patel-Schneider P (2003) *The Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, Cambridge, United Kingdom
3. Berk E (1997) *JLex: A lexical analyzer generator for Java*, Technical Report, Department of Computer Science, Princeton University. Latest version available at: http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html
4. Borgida A (1996) On the relative expressiveness of description logics and predicate logics. Artificial Intelligence 82(1-2):353–367
5. Chalupsky H (2000) *OntoMorph: a translation system for symbolic knowledge*. In: Cohn AG, Giunchiglia F, Selman B (eds) 7th International Conference on Knowledge Representation and Reasoning (KR'00). Breckenridge, Colorado. Morgan Kaufmann Publishers, San Francisco, California, pp 471–482
6. Corcho O, Gómez-Pérez A (2000) *A Roadmap to Ontology Specification Languages*. In: Dieng R, Corby O (eds) 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00). Juan-Les-Pins, France. Springer-Verlag, Lecture Notes in Artificial Intelligence (LNAI) 1937, Berlin, Germany, pp 80–96
7. Euzenat J (2001) *Towards a principled approach to semantic interoperability*. In: Gómez-Pérez A, Grüninger M, Stuckenschmidt H, Uschold M (eds) IJCAI2001 Workshop on Ontologies and Information Sharing, Seattle, Washington
8. Euzenat J, Stuckenschmidt H (2003) *The `family of languages' approach to semantic interoperability*. In: Omelayenko B, Klein M (eds) Knowledge transformation for the semantic web, IOS press, Ámsterdam, The Netherlands, pp49-63
9. Euzenat J, Tardif L (2001) *XML transformation flow processing*. Markup languages: theory and practice 3(3):285–311
10. Gómez-Pérez A, Fernández-López M, Corcho O (2003) *Ontological Engineering: with examples from the areas of knowledge management, e-commerce and the Semantic Web*, Springer-Verlag, New York.
11. Gruber TR (1993) *A translation approach to portable ontology specification*. Knowledge Acquisition 5(2):199–220
12. Hudson SE (1999) *Cup's User Manual*, Technical Report, Graphic Visualization and Usability Center, Georgia Institute of Technology. Latest version available at: http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.htm
13. Johnson SC (1975) Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey
14. Klein M (2001) *Combining and relating ontologies: an analysis of problems and solutions*. In: Gómez-Pérez A, Grüninger M, Stuckenschmidt H, Uschold M (eds) IJCAI2001 Workshop on Ontologies and Information Sharing, Seattle, Washington
15. Knublauch H (2003) Editing Semantic Web Content with Protégé: the OWL Plugin. 6th Protégé workshop. Manchester, United Kingdom
16. Lesk ME (1975) *Lex - A Lexical Analyzer Generator*, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey
17. Levine R, Mason T, Brown D (1992) *Lex & Yacc*, O'Reilly & Associates, second edition Sebastopol, Canada
18. Morris CW (1938) *Foundations of the theory of signs.* In: Neurath O, Carnap R, Morris CW (eds) International encyclopedia of unified science. Chicago University Press [reprinted in C W Morris 1971 *Writings on the theory of signs*. Mouton, The Hague]
19. Swartout B, Ramesh P, Knight K, Russ T (1997) *Toward Distributed Use of Large-Scale Ontologies*. In: Farquhar A, Gruninger M, Gómez-Pérez A, Uschold M, van der Vet P (eds) AAAI'97 Spring Symposium on Ontological Engineering. Stanford University, California, pp 138–148