Bridging the Gap Between Abstract and Concrete Services
– A Semantic Approach for Grounding OWL-S –

Steffen Balzer and Thorsten Liebig

University of Ulm, Dept. of Artificial Intelligence, Germany
{balzer|liebig}@informatik.uni-ulm.de

Abstract. OWL-S [1] is one of the emerging standards for the semantic description of web services in order to enable their automatic discovery, execution and composition by software agents. An important task within automatic execution of an OWL-S service is the bi-directional mapping between semantically higher level OWL-S service parameter descriptions and primitive XML Schema types of its grounding. OWL-S proposes to utilize XSL Transformations (XSLTs) for the mapping between these representation levels. However, this approach has a substantial shortcoming due to the fact that one OWL model can have many different RDF serializations whereas each requires a specific XSL stylesheet in the worst case. This severely limits the practical applicability of OWL-S in general. In this paper we present a simple approach of OWL-S parameter type mappings on a semantical basis. We therefore define an RDFS ontology of *RDF Mappings* that enable bi-directional mappings between OWL and XML Schema types. We show how to integrate RDF Mappings into the OWL-S grounding ontology and prove their feasibility by describing our prototypical implementation.

1 Introduction

OWL-S[1] is an ontology-based approach for the semantic description of web services that is inspired by research in the field of the Semantic Web. It intends to provide semantic markup that allows software agents to discover, execute, and compose web services automatically. An executable OWL-S service description consists of three parts: at least one service profile, exactly one process model, and at least one service grounding.

The service profile provides a way to describe those services which are offered by a provider, or those which are needed by a requester. In particular, the profile can be used to provide information about the offering organization, the function the service computes in terms of input, outputs, or conditions, and a set of additional service properties.

The process model describes the execution of a web service in detail by specifying the interrelations of different execution steps down to the level of atomic

[1] In the following we refer to the latest available version 1.0 of the OWL-S specification.

processes which are not further decomposable. In order to achieve the offered service results an agent has to execute the corresponding process model step by step considering all defined input/output dependencies and conditions.

The grounding of a web service specifies the details of how to access and communicate with the service on a technical level. It enables the application of a service by a mapping from an abstract specification of web service characteristics given in the process model to a concrete specification in terms of a particular protocol, message format, and serialization. Such a mapping approach has the advantage of de-coupling higher level service descriptions from concrete message specifications and therefore allows to layer on top of industry adopted standards more easily.

The OWL-S specification exemplarily defines a grounding for process descriptions to WSDL [2], which is the only type of developed grounding for OWL-S so far. Figure 1 gives an overview over the classes and properties of the grounding ontology for WSDL. In the following we focus on that highlighted

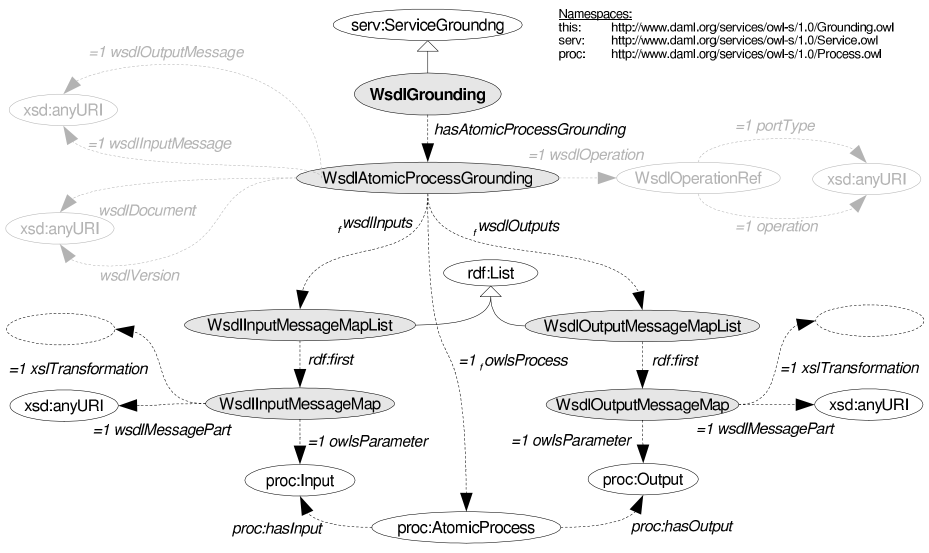


Fig. 1. The OWL-S grounding ontology

part of the grounding in figure 1, which is responsible for inputs and outputs of atomic OWL-S processes. A specific instance of a WsdlGrounding contains one WsdlAtomicProcessGrounding instance for each atomic process of the corresponding process model. The mapping of parameter types for each of them is defined using rdf:Lists of Wsdl{Input|Output}MessageMaps. OWL-S considers two ways of referring the appropriate OWL representations. If the web service is an "OWL native speaker" the OWL class representing the parameter type is referred directly by the property owlsParameter. For contemporary web services, e. g. WSDL based services, the grounding has to establish a binding between abstract inputs/outputs of OWL-S process steps and WSDL messages. This requires a bi-directional mapping of OWL types into XML Schema types used in WSDL communication protocols. The OWL-S specification proposes to use XSL Transformations to convert OWL parameters into XML Schema and vice versa.

However, OWL is developed as a vocabulary extension of the Resource Description Format (RDF). RDF is a data model whose models can be serialized in XML syntax in many syntactically different but semantically equivalent ways. As a result, the encoding of an OWL model in XML syntax is a one-to-many mapping. Since XSLT is a pure syntactical approach it can not be used to provide a sufficient solution for mapping abstract to concrete services in general. In this paper, we therefore propose an alternative, semantical approach for grounding OWL-S services suitable for general use independent of any serialization strategy. This approach is called the *RDF-Mapping* approach for OWL-S service parameters in the following.

The remainder of this paper is organized as follows. The next section gives a more detailed insight about the shortcomings of a XSLT-based grounding. In section 3 we propose our RDF-Mapping approach. Here, we introduce our RDF typemapping schema and provide corresponding transformation algorithms. Section 4 shortly describes our prototypical implementation followed by a brief discussion of related work in section 5. We will end with some conclusions and an outlook in section 6.

2 Grounding OWL-S Services with XSLT

The *eXtensible Stylesheet Language Transformation* [3] is a recursive programming language that allows XML documents to be transformed from one schema to another (e. g. see [4] for an introduction). XSLT follows a rule-based approach utilizing pattern matching substitutions and is therefore sensible to the structure of the XML source document. As mentioned before, OWL documents are used to specify a semantical model — usually called an ontology. An encoding of such a model may result in many syntactically different serializations. This is caused by mapping alternatives of OWL language elements to RDF graphs [5] and serializations thereof.

Within the task of grounding an OWL-S service this issue comes into play when linking parameters of OWL-S process steps with WSDL messages consisting of XML Schema values. However, the direction of mapping plays an important role here. As can be seen on the left hand side of figure 2, an output mapping from a XML Schema definition to OWL (from a lower to a higher level description) is less problematical. Here, serialization is predetermined by the WSDL parameter specification. In contrast, an input mapping, namely a mapping from OWL to XML Schema, may result in different serializations (see right hand side of figure 2). In order to guarantee general applicability this kind of mapping has to provide a style sheet for every possible serialization. Even worse, the existence

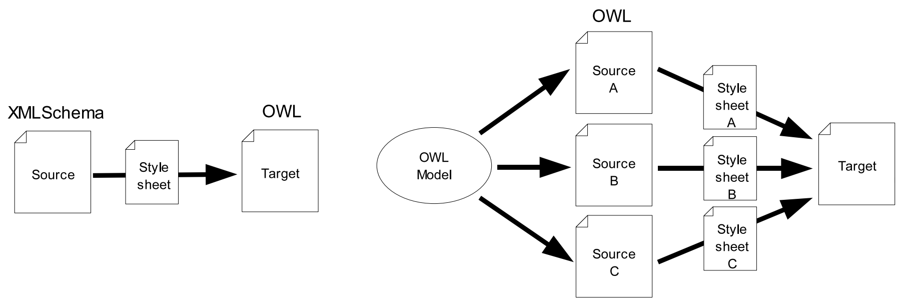


Fig. 2. XSL Transformations from XML Schema into OWL and vice versa (output mapping l.h.s., input mapping r.h.s.)

of nested relations or references results in an exponentially growing number of serializations.

Obviously, this method for grounding services does not meet the criteria of practicability and will very likely hamper the adoption of OWL-S at industry scale. However, the most important lesson from the above is the fact that grounding an OWL-S service has to be done on a semantical rather than syntactical level. Such a semantical, RDF-based approach is introduced in the following.

3 RDF Mappings

Our approach to semantic type mappings utilizes a simple RDFS ontology called the *RDF Typemapping Schema* to specify semantic dependencies between OWL-S parameter types and their corresponding low-level data types in XML Schema. Data transformations are performed by a specific algorithm. The declaration of RDF Mappings and the corresponding algorithm will be discussed in detail in this section.

3.1 RDF Typemapping Schema

RDF-Mapping documents define instances of classes specified by the *RDF Typemapping Schema* ontology which is shown in figure 3. They represent repositories of RDF-mappings, where each particular RDF-Mapping can be referenced in a message map of a service grounding using an URI, i.e. RDF-Mappings can easily be shared by different groundings.

RDFMapping is the base class of all type mappings that can be instantiated in an RDF-Mapping document. RDF mappings cover all XSD types that can be defined based on the WSDL 1.1 recommended approach for encoding abstract types using XSD [2]. XSD complex types which comply to SOAP compound types (see [6] for details) are covered by the XSDComplexTypeMapping class. XSD simple types which correspond to SOAP simple types are mapped with the XSDSimpleTypeMapping class. SOAP arrays are used in WSDL type definitions to specify value arrays. They play a special role due to the fact that they have not been considered in the XML Schema specification. They are mapped separately with the SOAPArrayTypeMapping class to allow a simple and straight forward definition of the transformation algorithm (see section 3.2). RDF Mappings are defined using nestings of RDFMapping sub-classes. The structure of RDF-Mappings is cloned from their corresponding XSD types which are referenced by the property hasXSDType. The property hasNestedName is specified in nested RDF Mappings to define the name of the corresponding nested XSD type element[2].

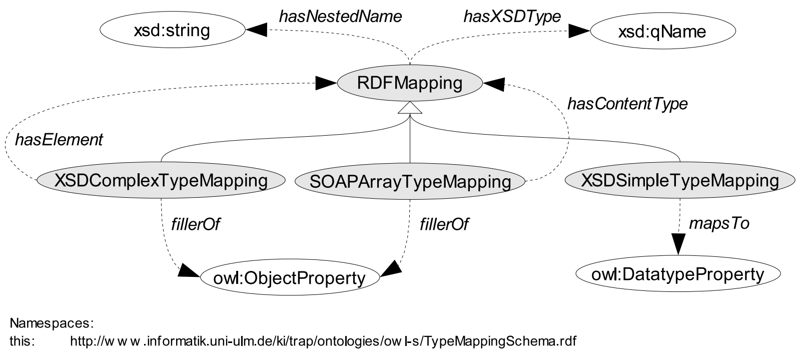


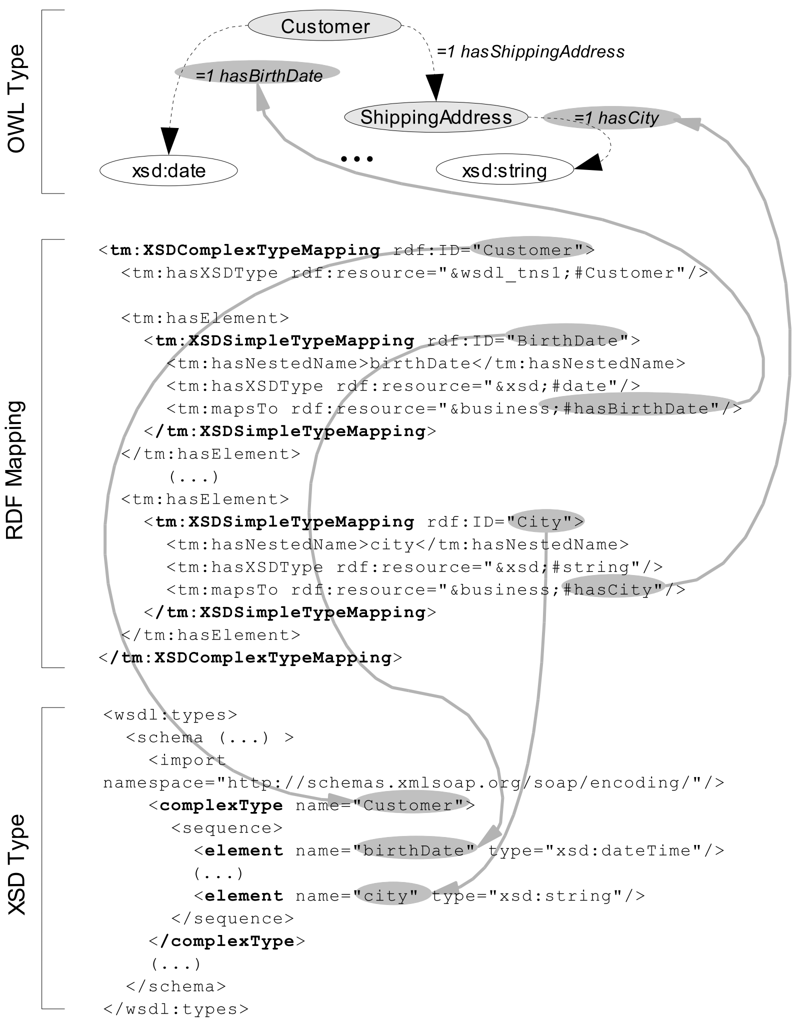
Fig. 3. RDFS ontology for the definition of RDF Mappings

RDF Mappings now exploit the fact that from an OWL perspective every data value to be transformed corresponds to a filler for the owl:DatatypeProperty class. On the XML Schema side such a value is an instance of an xsd:anySimpleType according to the XML Schema Datatype Specification [7]. This relationship is modelled with the mapsTo property of an XSDSimpleTypeMapping. It references the owl:DatatypeProperty of the corresponding OWL type definition. This justifies the reason for having chosen RDF as representation formalism. Due to the fact that an owl:DatatypeProperty is a sub-class of rdf:Property it can be used as a valid range for a rdf:Property while this is not possible in OWL DL. The fillerOf Property references the owl:ObjectProperty that connects OWL instances representing complex and array values to their preceeding instances. Its only purpose is to simplify the transformation algorithms.

Figure 4 shows an example of a simple RDF mapping. The Customer class represents an OWL type that can be used as an OWL-S parameter in an abstract

[2] This property is of practical use only and could be omitted because the indentifier can be extracted from the XSD type definition. However, WSDL4J (see section 4) which is used to parse the WSDL documents does not provide a programming model of the WSDLTypes section.

service description. It references both XSD types like xsd:date and other OWL types like ShippingAddress which itself aggregates XSD types in turn. The XSD type on the side of the concrete service description has a flat structure, i.e. all XSD types to be mapped are aggregated in one complex type by a sequence of simple type elements. The RDF-Mapping now clones the structure of the Customer XSD type and extends the XSD simple types by specifying a reference to their corresponding OWL attributes of the Customer OWL type. The generation of hierarchical OWL instances resp. flat XSD values is considered by the specific transformation algorithms which will be discussed in the next section.



```
<tm:XSDComplexTypeMapping rdf:ID="Customer">
  <tm:hasXSDType rdf:resource="&wsdl_tns1;#Customer"/>

  <tm:hasElement>
    <tm:XSDSimpleTypeMapping rdf:ID="BirthDate">
      <tm:hasNestedName>birthDate</tm:hasNestedName>
      <tm:hasXSDType rdf:resource="&xsd;#date"/>
      <tm:mapsTo rdf:resource="&business;#hasBirthDate"/>
    </tm:XSDSimpleTypeMapping>
  </tm:hasElement>
      (...)
  <tm:hasElement>
    <tm:XSDSimpleTypeMapping rdf:ID="City">
      <tm:hasNestedName>city</tm:hasNestedName>
      <tm:hasXSDType rdf:resource="&xsd;#string"/>
      <tm:mapsTo rdf:resource="&business;#hasCity"/>
    </tm:XSDSimpleTypeMapping>
  </tm:hasElement>
</tm:XSDComplexTypeMapping>
```

```
<wsdl:types>
  <schema (...) >
    <import
namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
    <complexType name="Customer">
      <sequence>
        <element name="birthDate" type="xsd:dateTime"/>
        (...)
        <element name="city" type="xsd:string"/>
      </sequence>
    </complexType>
    (...)
  </schema>
</wsdl:types>
```

Fig. 4. Example of an RDF Mapping

The following enumeration describes all information sources that are required to perform a transformation along with their responsibilities:

1. *WSDL documents* contain the definitions of the XSD types in their WSDL types section. These definitions serve as structural reference for generating parameter instances required by concrete services.
2. *OWL ontologies* contain the definitions of the OWL types that will be mapped to their XSD equivalents. They are used as a structural reference for generating OWL instances from the results returned by a concrete service.
3. *RDF-Mappings* establish a link between OWL and XSD type definitions enabling bijective type mappings as described above.
4. *OWL-S groundings* reference in their WsdlMessageMaps both the OWL parameter type of the abstract service indirectly with owlsParameter and the corresponding RDF-Mapping with rdfMapping. Thus, embedding an RDF-Mapping in a grounding finally connects the XSD type to its OWL type.

3.2 Transformation Algorithm

This section describes how a type mapping is performed on the basis of the information sources specified above. Figure 5 shows the abstract design of the *OWL-S Type Mapping Module* (TMM). The grounding of a service forms the root of the information model. The `TypeMapperFactory` class which aggregates the information model on its creation by obtaining a reference to the grounding is then used by the programmer to instantiate so called *type mappers* that implement the specific transformation algorithms for the particular RDF-Mappings. The mappings of inputs and outputs require different algorithms. Thus, considering all sub-classes of RDFMapping six different type mappers are required[3].

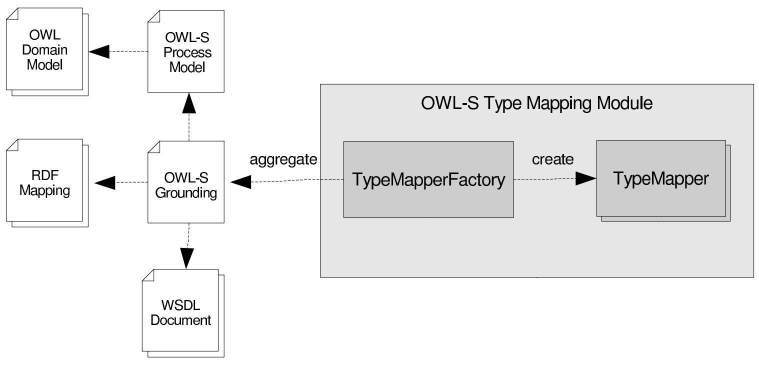


Fig. 5. Abstract design of the OWL-S Type Mapping Module

[3] Object classes that implement the different type mappers are named according to the schema `{Simple|Complex|Array}{Input|Output}TypeMapper`

The basic algorithm can be divided into two phases. In *phase I* concrete instances of the type mappers are generated recursively by calling either `createInputTypeMapper()` or `createOutputTypeMapper()` of the `TypeMapperFactory` with the corresponding WSDLMessageMap URI. Algorithm 1.1[4] shows exemplarily the creation of `InputTypeMappers`. `OutputTypeMapper`s will be created analogically. In both cases the algorithm will produce a type mapper hierarchy reflecting the structure of the RDF-Mapping and the XSD type definition. The creation methods return a reference to the root type mapper which delivers the interface for the transformation in phase II. During instantiation a type mapper collects required data from the information model, stores it in instance variables and instantiates its direct sub-type mappers. Arrays are represented in OWL using a special list construction similar to the one proposed in the OWL-S 1.0 DL ontologies[5]. Acording to the definition array elements cannot be instances of XSD datatypes because owl-list:first is defined as object property. Therefore, no `SimpleTypeMapper` needs to be considered for arrays (see line 38).

In *phase II* the transformation is performed by calling either the `transformToOWL()` or the `transformFromOWL()` method of a root type mapper. The transformation is propagated recursively through the type mapper hierarchy constructed in phase I. `transformToOWL()` which must be implemented by an output type mapper transforms an XSD value into an OWL instance, asserts the new instance in the provided knowledge base, and returns its URI. `transformFromOWL()` must be implemented by input type mappers and returns the transformed XSD value representation of a given URI of an OWL instance that resides in the provided knowledge base. The next two sections describe the abstract transformation algorithms of each type mapper in more detail.

From OWL to XML Schema Algorithm 1.2 shows the transformation procedure for this direction exemparily for complex and simple types. The *value containers* (see lines 5 and 14) represent an abstract model of an XML tree containing the XSD values that will be generated during transformation. A value container can be implemented in different ways, e.g. as XML document object model or as nested JavaBeans (see section 4 for more details). The `integrate()` method (see line 9) integrates nested value containers resp. XML trees into the tree of the next higher level. The root value container can then be used to serialize the complete model. The `generateContainerRepresentation()` method finally creates a representation of the OWL datatype value that can be integrated into the value container. Thus, the implementations of these two methods depend on the chosen implementation for the value containers. An implementation based on JavaBeans is presented in section 4.

The `determineDataOwner()` method (see line 23) traverses the assertion tree of the OWL root instance to determine the instance that forms the direct predecessor of the OWL datatype property that refers to the value to be transformed. This instance is called *data owner.*

[4] Error handling has been omitted in favour of brevity

[5] `http://www.daml.org/services/owl-s/1.0DL/generic/ObjectList.owl`

From XML Schema to OWL Algorithm 1.3 shows the procedure for mapping XML Schema values to OWL instances for complex and simple types again. The `transformToOWL()` method utilizes the mapping information collected in phase I to recursively generate an OWL instance tree from a given root value container in the provided knowledge base by calling the appropriate type mappers for all value containers nested in the root.

The root type mapper first asserts the instance tree in the given knowledge base by invoking the `createInstanceTree()` method (see lines 8 and 20). It asserts all necessary OWL instances and object relations according to the OWL class definition. The leaves, i.e. the datatype properies and their fillers, however, are avoided. They will be asserted when their corresponding `SimpleOutputTypeMapper`s will be called. `createInstanceTree()` could also cache data owners and *object owners*[6] to speed up execution of the following two methods.

The methods `retrieveDataOwner()` and `retrieveObjectOwner()` play a crutial role in expanding a flattened XSD value. Just like `detemineDataOwner()` for inputs, `retrieveDataOwner` (see line 23) returns the OWL instance from the generated instance tree a SimpleOuputTypeMapper asserts its transformed value to. The `retrieveObjectOwner()` method serves a dual purpose. Firstly, it is used to determine the OWL instance a `ComplexOutputTypeMapper` corresponds to. This instance is used as new root when calling the sub-type mappers. This way the instance tree is only traversed once in an incremental fashion. Secondly, an `ArrayOutputTypeMapper` determines its corresponding predecessor in the instance tree to be able to assert its array elements.

The asserted OWL instances can then be serialized using standard serialization methods provided by the DL knowledge base. The algorithm pursues a lazy assertion strategy, i.e. it only asserts relations and OWL instances that reside on paths from the root instance to mapped values.

3.3 Conceptual Restrictions

In order to achieve termination and practical soundness of the above transformation algorithms the following restrictions have been assumed[7].

1. Due to the strict hierarchical structure of RDF-Mappings and XSD types, cyclic definitions of OWL types are not allowed, i.e. a Customer for example cannot refer to a Customer as a property filler within its definition.
2. The mapsTo property of a XSDSimpleTypeMapping is only allowed to point to directly or indirectly reachable attributes of the OWL type referenced in the grounding. In other words, all referenced attributes in an RDF-Mapping must have the same OWL type as direct or indirect predecessor and this OWL type is referenced as OWL-S parameter type in the corresponding WsdlMessageMap.

[6] Analogical to data owners, an object owner determines the preceeding instance of an object property.

[7] A refinement of the mapping model and its algorithms may lead to less restrictive assumptions.

3. Currently, a data or object owner is determined by executing a breadth-first-search on the assertion graph of an OWL instance. Therefore, the occurence of the OWL attribute assertion must be unique within the whole assertion graph. As a consequence, sets cannot be defined simply by asserting the same attribute several times with different data fillers[8].
4. The hierarchical structure of an RDF-Mapping must exactly match the structure of the corresponding XSD type referenced via hasXSDType to ensure the correct generation of the XML representation of the XSD value.
5. Fillers of hasNestedName must match with the parameter names of the corresponding nested XSD types. A mismatch would lead to incorrect tag names within an XSD value representation.
6. An instance of XSDSimpleTypeMapping must own exactly one mapsTo property. Otherwise the mapping would be defined ambiguously.

It can easily be observed that the consistency of RDF-Mappings w.r.t. the above restrictions cannot be verified only on the basis of RDFS semantics. However, all restrictions can be tested using special purpose algorithms.

4 Implementation

This section describes the prototypical implementation of the proposed type mapping approach. It is part of an OWL-S process execution engine that has been developed by the authors. Figure 6 shows the concrete architecture of the TMM.

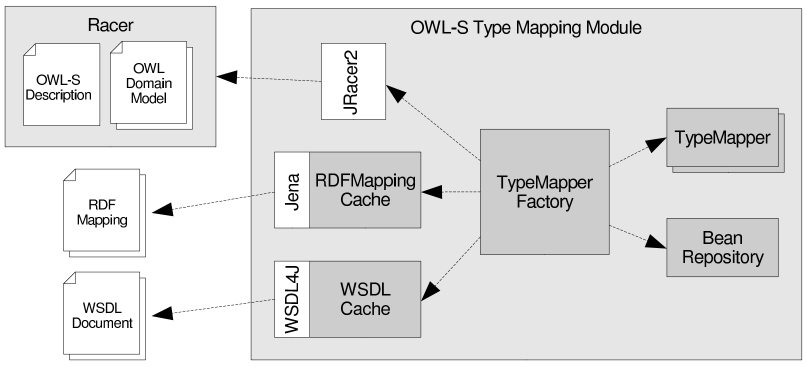


Fig. 6. Concrete architecture of the OWL-S Type Mapping Module

The different information sources are interfaced to Java with the following components: OWL domain models and OWL-S service descriptions are managed

[8] This restriction can be eliminated by saving the complete property chain in the RDF-Mapping e.g. as rdf:Seq.

by the description logics reasoning system *Racer* [8] which is accessed via its Java API *JRacer2*. RDF-Mappings and WSDL documents can be shared by different groundings. Therefore, they are loaded into caches. The *Jena framework*[9] provides graph based access to RDF-Mappings and basic RDFS inference. The *WSDL4J*[10] package delivers the java programming model for WSDL documents.

The bean repository plays a central role in the implementation of the value containers that have been introduced in section 3.2. In the TMM value containers and their nestings are implemented as *JavaBeans* in order to provide seamless integration into the *AXIS Web Service Framework*[11]. The beans are generated and compiled as required using Axis' WSDL2Java tool. Hereafter, the bean classes are loaded dynamically into the bean repository. Access to bean instances is realized via Java Introspection. The bean instances consumed or returned by the type mappers can directly be used in the Axis framework with its default bean serializers.

5 Related Work

The task of grounding OWL-S has rarely been addressed so far. Two more or less distantly related approaches of mediating XML data on a conceptual level are shortly analysed in the following.

The author of [9] proposes an RDFS meta-ontology called *RDF Transformations (RDFT)* for B2B integration. The services that a business provides and the documents which must be exchanged in order to invoke them are assumed to be represented in WSDL descriptions and their included XML Schema definitions. RDFT utilizes the *Process Specification Language PSL*[12] to augment WSDL with basic temporal semantics. Parts of the OMG's *Common Warehouse Model (CWM)*[13] have been adopted to describe mappings of RDF schemas and specific concepts like PSL events, WSDL messages, vocabularies etc. Therefore, different kinds of RDFT Bridges are specified. RDFBridges like Class2Class or Property2Property map between different RDF schemas whereas XMLBridges like Tag2Class or Tag2Property map between XML and RDF elements. If it is not feasible to specify the correspondence of two elements directly, XPath expressions can be used. Bridges are aggregated into Maps which are parsed to generate XSL stylesheets that are then used for XML document transformations. Despite the fact that RDFT does not support mappings with OWL classes it is more mature in satisfying business integration needs compared to our approach. It covers e.g. one-to-many and many-to-one document transformations which are not considered in RDF-Mappings, yet. However, transformations are still intended to be executed on a syntactic level by utilizing XML transformation languages and therefore still depend on serialization variants. In contrast, RDF-Mappings are

[9] `http://www.hpl.hp.com/semweb/jena.htm`
[10] `http://oss.software.ibm.com/developerworks/projects/wsdl4j`
[11] `http://ws.apache.org/axis/`
[12] `http://www.mel.nist.gov/psl/`
[13] `http://www.omg.org/cwm/`

completely serialization independent due to their model-based transformation algorithms.

The *Piazza* system [10] provides an architecture to answer queries over heterogeneous XML data resources distributed in a peer-to-peer network. The XML data (i.e. XML schema instances) and the domain knowledge (i.e. RDF/OWL ontologies) of different network nodes are integrated in a pairwise resp. point-to-point manner by defining mappings in a special language that utilizes XQuery. The queries can be issued from a node using its local schema and domain model. A query rewriting algorithm transforms the query using the defined mappings so it can be executed on neighbour nodes using their local model. Piazza however is focused on XML data retrieval and the proposed language is quite complicated. The mediation of data for execution of XML web services which is the main purpose of RDF Mappings has not been addressed.

6 Conclusion and Future Work

OWL-S aims to provide semantic markup in order to enable agents to discover, invoke, and compose web services. However, in the course of realizing a vertical prototype of an OWL-S web service we have been faced with a couple of serious difficulties on different conceptual levels of the OWL-S specification.

One of the major problems concerning web service invocation has been related with the WSDL grounding of OWL-S services. The OWL-S specification proposes to use XSLT for mapping between abstract service parameters and concrete WSDL messages. It has turned out, that this approach is not practical in general because of serialization variants. Our RDF-Mapping approach is based on a semantic mapping and is therefore independent from any syntactical issues. This mapping has been sucessfully implemented and tested. Even though the RDF-Mapping approach worked very well with our test cases we have not made any extensive evaluations or realistic benchmarks yet. However, our algorithms have been developed in a simple straightforward manner without exploiting any optimizations, yet.

A major feature of our approach is the fact, that the generation of RDF-Mappings can be automatized to a maximal extent. In fact, an automatic OWL-S service grounding via the RDF-Mapping approach only requires a manual specification of the mapsTo and fillerOf properties.

Moreover, we have shown that data transformations on a semantic level can be realized with structurally very simple descriptions like RDF-Mappings. The usage of additional languages like XSLT, XPath or XQuery could be omitted. Therefore, RDF-Mappings are simple to learn and easy to understand.

As a logical consequence, we aim to publish the RDF-Mapping system as a standalone module. Beyond that, we plan to develop a simple GUI-tool for intuitive and semi-automatic generation of RDF-Mappings. However, the development of the system is still at a very early stage and to be able to serve more realistic B2B scenarios current limitations of the approach as desribed in section 3.3 must be eliminated.

References

1. Ankolekar, A.: OWL-S: Semantic Markup for Web Services (2003) `http://www.daml.org/services/owl-s/1.0/owl-s.pdf`.
2. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. Technical report, Word Wide Web Consortium (2001) `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.
3. Clark, J.: XSL Transformations (XSLT) Version 1.0. W3C Recommendation (1999)
4. Kay, M.: XSLT Programmer's Reference. 2nd edn. Wrox (2001)
5. Patel-Schneider, P., Hayes, P., Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax (2004) W3C Recommendation.
6. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., Winer, D.: Simple Object Access Protocol (SOAP) 1.1. Technical report, Word Wide Web Consortium (2000) `http://www.w3.org/TR/SOAP`.
7. Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes. Technical report, Word Wide Web Consortium (2001) `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.
8. Haarslev, V., Möller, R.: Description of the Racer System and its Applications. In: Proc. Int. Workshop on Description Logics (DL-2001), Stanford, USA (2001)
9. Omelayenko, B.: RDFT: A Mapping Meta-Ontology for Business Integration. In: Proc. of the Workshop on Knowledge Transformation for the Semantic Web (KTSW 2002) at the 15th European Conference on Artificial Intelligence, Lyon, France (2002)
10. Halevy, A.Y., Yves, Z.G., Mork, P., Tatarinov, I.: Piazza: Data Management Infrastructure for Semantic Web Applications. In: Proc. of the 12th International Conference on World Wide Web (WWW 2003), Budapest, Hungary (2003)

Algorithm 1.1 Phase I: Type mapper creation

```
$mm \Leftarrow$ grnd:wsdlInputMessageMap
$pn \Leftarrow$ parameter name as filler of $mm$.grnd:wsdlMessagePart
$rm \Leftarrow$ RDF-Mapping as Filler of $mm$.grnd:rdfMapping

TypeMapperFactory.createInputTypeMapper($mm$, $pn$, $rm$)
$ct \Leftarrow$ OWL type as filler of $mm$.grnd:owlsParameter.proc:parameterType
if $rm$ is instance of tm:XSDSimpleTypeMapping then
    return new SimpleInputTypeMapper($rm$, $pn$, $ct$)
else if $rm$ is instance of tm:XSDComplexTypeMapping then
    return new ComplexInputTypeMapper($rm$, $pn$, $ct$)
else if $rm$ is instance of tm:SOAPArrayTypeMapping then
    return new ArrayInputTypeMapper($rm$, $pn$, $ct$)
end if

ComplexInputTypeMapper.<constructor>($rm$, $pn$, $ct$)
this.$rm \Leftarrow rm$; this.$pn \Leftarrow pn$; this.$ct \Leftarrow ct$
this.$nn \Leftarrow$ nested name as filler of $rm$.tm:hasNestedName
this.$op \Leftarrow$ OWL object property as filler of $rm$.tm:fillerOf
this.$xt \Leftarrow$ XSD complex type as filler of $rm$.tm:hasXSDType
$\mathcal{S} = \{\}$: Set of all sub-type mappers
for each Filler $e$ of $rm$.tm:hasElement do
    if $e$ is tm:XSDSimpleTypeMapping then
        $s \Leftarrow$ new SimpleInputTypeMapper($e$, null , null)
    else if $e$ is instance of tm:XSDComplexTypeMapping then
        $s \Leftarrow$ new ComplexInputTypeMapper($e$, null , null)
    else if $e$ is instance of tm:SOAPArrayTypeMapping then
        $s \Leftarrow$ new ArrayInputTypeMapper($e$, null , null)
    end if
    this.$S \Leftarrow S \cup \{s\}$
end for

ArrayInputTypeMapper.<constructor>($rm$, $pn$, $ct$)
this.$rm \Leftarrow rm$; this.$pn \Leftarrow pn$; this.$ct \Leftarrow ct$
this.$nn \Leftarrow$ nested name as filler of $rm$.tm:hasNestedName
this.$op \Leftarrow$ OWL object property as filler of $rm$.tm:fillerOf
this.$oe \Leftarrow$ OWL element type as range of $ct$.owl-list:first
this.$xt \Leftarrow$ XSD content type as filler of $rm$.tm:hasContentType
if $xt$ is instance of tm:XSDComplexTypeMapping then
    this.$s \Leftarrow$ new ComplexInputTypeMapper($xt$, "" , $oe$)
else
    this.$s \Leftarrow$ new ArrayInputTypeMapper($xt$, "" , $oe$)
end if

SimpleInputTypeMapper.<constructor>($rm$, $pn$, $ct$)
this.$rm \Leftarrow rm$; this.$pn \Leftarrow pn$; this.$ct \Leftarrow ct$
this.$nn \Leftarrow$ nested name as filler of $rm$.tm:hasNestedName
this.$oa \Leftarrow$ OWL attribute as filler of $rm$.tm:mapsTo
this.$xt \Leftarrow$ XSD datatype as range of this.$oa$
```

Algorithm 1.2 Phase II: Transformations from OWL to XML Schema

```
 1: $\mathcal{KB} \Leftarrow$ knowledge base holding the OWL instance
 2: $i \Leftarrow$ URI of the OWL instance
 3:
 4: ComplexInputTypeMapper.transformFromOWL($\mathcal{KB}$, $i$)
 5: $\mathcal{C}_o \Leftarrow$ outer *value container* for nested elements
 6: **for each** $s \in$this.$\mathcal{S}$ **do**
 7:    $j \Leftarrow$ *determineDataOwner*($\mathcal{KB}$, $s$, $i$)
 8:    $\mathcal{C}_i \Leftarrow$ $s$.transformFromOWL($\mathcal{KB}$, $j$)
 9:    *integrate*($\mathcal{C}_i$, $\mathcal{C}_o$)
10: **end for**
11: **return** $\mathcal{C}_o$
12:
13: SimpleInputMapper.transformFromOWL($\mathcal{KB}$, $i$)
14: $\mathcal{C} \Leftarrow$ *value container* for leaf element
15: $v \Leftarrow$ OWL value as attribute filler of $i$.(this.$oa$)
16: $r \Leftarrow$ *generateContainerRepresentation*(this.$xt$, $v$)
17: **return** $\mathcal{C} \cup \{r\}$
```

Algorithm 1.3 Phase II: Transformations from XML Schema to OWL

```
 1: $\mathcal{KB} \Leftarrow$ target knowledge base
 2: $\mathcal{C}_o \Leftarrow$ root value container
 3: $i_p \Leftarrow$ URI of the preceeding OWL instance or null
 4:
 5: ComplexOutputTypeMapper.transformToOWL($\mathcal{KB}$, $\mathcal{C}_o$, $i_p$)
 6: $i_0 \Leftarrow$ null
 7: **if** this.$nn$ = null **then**
 8:    *createInstanceTree*($\mathcal{KB}$, this.$ct$)
 9:    $i_0 \Leftarrow i_p$
10: **else**
11:    $i_0 \Leftarrow$ *retrieveObjectOwner*($\mathcal{KB}$, $i$, this.$op$)
12: **end if**
13: **for each** $(\mathcal{C}_i, s) : \mathcal{C}_i \in \mathcal{C}_o$ and $s \in$ this.$\mathcal{S}$ and $s$ sub-type mapper of $\mathcal{C}_i$ **do**
14:    this.$s$.transformToOWL($\mathcal{KB}$, $\mathcal{C}_i$, null)
15: **end for**
16: **return** $i_0$
17:
18: SimpleOutputTypeMapper.transformToOWL($\mathcal{KB}$, $\mathcal{C}_o$, $i_p$)
19: **if** this.$nn$ = null **then**
20:    *createInstanceTree*($\mathcal{KB}$, this.$ct$)
21: **end if**
22: $v \Leftarrow$ extract XSD simple value from $\mathcal{C}_o$)
23: $i \Leftarrow$ *retrieveDataOwner*($\mathcal{KB}$, $i_p$, this.$oa$)
24: assert attribute relation $i$.(this.$oa$) $\{v\}$ in $\mathcal{KB}$
25: **return** $i_r$
```