



## D24.2v0.1. WSMO Grounding

WSMO Working Draft 16 September 2005

**This version:**

<http://www.wsmo.org/TR/d24/d24.2/v0.1/20050916/>

**Latest version:**

<http://www.wsmo.org/TR/d24/d24.2/v0.1/>

**Previous version:**

<http://www.wsmo.org/TR/d24/d24.2/v0.1/20050905/>

**Authors:**

Jacek Kopecký  
 Matthew Moran  
 Dumitru Roman  
 Adrian Mocan

This document is also available in non-normative [PDF](#) version.

Copyright © 2005 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

### Table of contents

#### [1. Introduction](#)

##### [1.1 Purpose and Organisation of this Deliverable](#)

#### [2. Grounding WSMO Ontology to XML](#)

##### [2.1 Background and Related Work](#)

###### [2.1.1 XML](#)

###### [2.1.2 XML Schema](#)

###### [2.1.3 Previous Work on Mapping between XML Schema and Ontologies](#)

##### [2.2 Approaches to Grounding WSMO Ontologies to XML](#)

##### [2.3 Grounding WSMO Ontologies to XML by Creating Mappings at the Conceptual Level](#)

###### [2.3.1 Overview and Example](#)

###### [2.3.2 Steps to Ground WSMO to XML Schema](#)

##### [2.4 Definition of Mapping from XML Schema to WSMO](#)

###### [2.4.1 simpleType](#)

###### [2.4.2 complexType](#)

###### [2.4.3 Attribute](#)

###### [2.4.4 Element](#)

##### [2.5 Summary](#)

#### [3. Grounding WSMO to WSDL](#)

##### [3.1 WSDL Overview](#)

###### [3.1.1 Web Service Interface](#)

###### [3.1.2 Web Service Endpoints, Bindings](#)

###### [3.1.3 WSDL Documents](#)

###### [3.1.4 Note on the differences between WSDL 2.0 and WSDL 1.1](#)

##### [3.2 Grounding to existing WSDL descriptions](#)

###### [3.2.1 Correspondences between WSDL and WSMO](#)

###### [3.2.2 Grounding values](#)

###### [3.2.3 URIs for identifying WSDL components](#)

###### [3.2.4 Grounding example](#)

###### [3.2.5 Processing grounding information](#)

##### [3.3 Generating WSDL from WSMO choreography](#)

###### [3.3.1 Rules for generating WSDL](#)

###### [3.3.2 Example for the default grounding](#)

###### [3.3.3 Limitations of the default grounding](#)

#### [4. Conclusions and further work](#)

#### [References](#)

#### [Appendix A. Definition of Mappings from XML Schema to WSMO](#)

#### [Appendix B. Summary of Grounding Non-functional Properties](#)

#### [Acknowledgement](#)

#### [Change Log](#)

## 1. Introduction

Web services are, in general, systems that provide certain functionality to their clients and communicate with them by exchanging XML data over computer networks. In order to interoperate successfully, the client and the service must agree on what kinds of messages can be exchanged and when, and what exactly the message exchanges will mean. In other words, the client and the service must agree on a common interface.

In most cases it is the service provider that decides what the interface will look like, and the client has to understand the interface and comply with it. The interface must specify at least the following components:

- **message types** — what types of messages can be exchanged,

- **message serialization** — how information is serialized into bits and bytes,
- **message exchange** — who can send messages to whom and when
- **party obligations** — what are the responsibilities of the parties involved in the message exchange

In the area of Web services, XML Schema [\[XML Schema\]](#) and Web Services Description Language [\[WSDL\]](#) are used together to describe the different aspects of the interface. In particular, XML Schema constrains the message types in XML tree-based structure and WSDL specifies simple message exchanges (operations) and also provides serialization details for the messages. The ordering and meaning of the operations is not formally specified; it is usually implied or described by plain text instead. This is sufficient for human operators to create clients that can correctly use Web services.

To enable automatic discovery and invocation of Web services, the Web Services Modeling Ontology (WSMO, see [\[Roman et al. 2004\]](#)) describes functional and behavioral aspects of Web services using an approach based on logics and knowledge representation. First, to enable Web service discovery and composition, WSMO focuses on describing *what* Web services do using service *capabilities*. Second, to make it possible for clients to determine how to communicate with discovered services, WSMO specifies the interface of a Web service using the service *choreography*. (A WSMO interface may also contain an orchestration which describes how the Web service communicates with other Web services to achieve its functionality. This document does not currently deal with orchestration.)

A WSMO choreography [\[Roman et al. 2005\]](#) is a state machine, with its states described using ontologies, in terms of concepts, their instances and the relations between them. Inputs and outputs of the Web service are represented as instances of certain concepts that can be read or written by the client communicating with the service. Each concept in the choreography state ontology can be assigned to a role which determines whether the clients may read or update (or both) the values of instances of that concept. We will call these concepts and instances *accessible*. The purpose of WSMO grounding is to describe the exact mechanism, using which the client writes or reads the accessible instances.

The state of the art for Web service interface description is to use WSDL 2.0. If the technology infrastructure for Semantic Web services is to be successful, it is very important that this infrastructure be able to communicate with existing Web services described with WSDL. In the context of WSMO-compliant execution environments, only Web services with a WSMO description are available for the operations of discovery, composition, invocation, etc. Therefore, the first step in making a service having only a WSDL description available is to create its corresponding WSMO description. This makes the service *description* available to execution environment but does not solve the problem of how the *implementation* of the service can be invoked.

Once a service has both WSMO and WSDL descriptions, we need to ground the WSMO description in the WSDL description so that a WSMO execution environment, following the choreography of the service, will be able to know which actual messages to send and/or expect to receive. The WSDL description will provide the networking details, for example that the data should be serialized as SOAP XML messages and sent over HTTP, or that the data can be sent as RDF triples to a Triple Space (see [\[Martin-Recuerda et al., 2005\]](#)).

## 1.1 Purpose and Organisation of this Deliverable

The purpose of this deliverable is to describe how WSMO service descriptions can be grounded to WSDL on the basis that WSDL provides the current industry standard for defining how messages can be exchanged between services over the Internet. There are two aspects to this problem, each of which is addressed in its own section of the deliverable.

The first aspect of grounding is that the data model of the input and output messages for WSDL services is defined using one or more XML Schemas while the data model for a WSMO service is defined using the conceptual model provided by one or more WSMO ontologies. This leads to the requirement of how to map between the ontological data in the state machine and its representation as XML messages. The mapping between ontological and XML data is discussed in [section 2](#).

The second aspect of grounding is to specify how and when messages to and from the service are generated and sent. WSMO choreography only says that the client can read the data but in fact it is the responsibility of the service to send the data to the client in the form of a message. The grounding must also provide the necessary serialization and networking details, i.e. what underlying protocol (e.g. SOAP, HTTP) should be used for passing the messages, how the XML data is encapsulated in the underlying protocol, and where exactly the data should be sent. [section 3](#) discusses these aspects of grounding.

## 2. Grounding WSMO Ontologies to XML

Semantically, Web service inputs and outputs are described in WSMO using ontologies. Syntactically, they are described in WSDL using XML Schema. As already mentioned, we envision that semantic agents (e.g. an automatic goal fulfillment — service discovery and execution engine) must be able to communicate with plain syntactic Web services (e.g. existing Amazon Web services) because industry is currently deploying heavily only syntactical Web services.

During the communication of a semantic-level client and a syntactic-level Web service, two directions of data transformations are necessary: the client's semantic data must be written in an XML form that can be sent as a request to the service, and the response data coming back from the service must be interpreted semantically by the client. With respect to a client whose domain knowledge is represented using WSMO ontologies, this means that the request data instances must be transformed from WSMO to XML and, conversely, the response data instances must be transformed from XML back to WSMO.

The rest of this section is organized as follows. A background to XML and XML Schema as well as some details of previous work is provided in [Section 2.1](#). [Section 2.2](#) introduces three possible approaches for grounding the data model aspect of WSMO services. [Section 2.3](#) describes the the preferred approach of grounding based on creating the mappings at the conceptual level. [Section 2.4](#) describes the mappings for each of the four main XML Schema types with examples. The final section, [section 2.5](#), provides a summary and makes some observations based on the definition of the mappings. The complete definition of the mapping is included in [Appendix A](#).

### 2.1 Background and Related Work

In this part of the document we provide brief overviews of XML ([section 2.1.1](#)) and XML Schema ([section 2.1.2](#)) as background material for the later sections defining the mappings. [Section 2.1.3](#) examines previous work on mapping between XML Schema and ontologies and indicates how the efforts described here related to these earlier efforts.

#### 2.1.1 XML

XML has revolutionised the way data is exchanged and represented across the Web. It provides a standard language for describing document types in any domain imaginable facilitating the sharing of data across different systems and most notably, the Internet. XML is flexible and extensible allowing users to create their own tags to match their own specific requirements. As a result many languages have been designed for use in very different fields based on XML. Some examples of these are RDF as a semantic annotation language based on triples, XHTML

as a document markup language with stricter constraints than HTML, RSS for syndication of website feeds most often used by news sites and weblogs. Where two systems need to exchange data, an agreed XML document structure is often the simplest mechanism to achieve the desired results.

XML encodes data using a set of identifiable tags representing elements that may have attributes associated with them. Attributes are used to assign name-value pairs to specific elements. In XML documents, tags can be nested but may not overlap. Nested elements provide a means to define the structure of an XML document. For example, the XML snippet in Listing 1 denotes that the element with the tag 'book' has subelements with the tags 'author', 'publisher' and 'yearPublished'. The element with the tag 'book' has one attribute denoting the book's title. This is an example of a case where an attribute could be just as easily represented as a sub-element.

Listing 1. XML Snippet

```
01 <book title="Harry Potter">
02   <author>J. K. Rowling</author>
03   <publisher>Random House</publisher>
04   <yearPublished>2000</yearPublished>
05 </book>
```

However, despite all its advantages, XML documents are syntactic and by themselves, provide no description of the semantics of the tags used. These semantics must be agreed by the users of the XML before they exchange any messages. In reality, many applications using XML depend on a human reading the document and interpreting the tags based on their natural language meaning. Although this is effective for smaller scale applications, it does not scale to cases where the recipient of the XML may be an automated agent that does not know the semantics of the XML tags in advance.

### 2.1.2 XML Schema

XML Schema is a W3C Recommendation defining a schema language for XML. XML Schema provides a way to define constraints on the syntax and structure of an XML document. It defines the legal elements and attributes that can appear including their types. Additionally it can specify the order and number of child elements and can define default and fixed values for elements and attributes. In the sense that XML Schema can define the valid data and elements for XML documents, there is some overlap between XML Schema and WSMO ontologies. Although, principally, XML Schema addresses the problem of constraining the syntax and structure of a set of XML documents while WSMO ontologies model the semantic relationships in a particular domain.

The next paragraphs provide a description of the XML Schema components that are most interesting from the perspective of creating a mapping to WSMO. These components are: *simple type definition*, *complex type definitions*, *attribute declarations*, and *element declarations*. Listing 2 is used to provide examples of each component.

**Simple Type Definition:** XML Schema provides a wide range of built-in datatypes, for example, string, integer, boolean, float, decimal etc. These are examples of one form of simple type definitions. Another form of simple type definitions can be used to provide constraints on the values of built-in types. For example, it may be necessary to restrict the allowed values of the positiveInteger datatype to a particular maximum value.

**Complex Type Definition:** Can be used to (i) define a datatype composed of sub-elements of other datatypes, (ii) define the allowed structure of child elements using the keywords *all*, *sequence* and *choice* or (iii) extend or restrict the definition of an existing complex type. Additionally, the values of elements can be accompanied by constraints on their values.

**Attribute Declarations:** XML Schema attributes are an association between a name and a simple type definition. They are used to define simple attributes for XML type definitions. Unlike element declarations, attributes can not contain other elements or other attributes. Attributes can either be defined in the scope of the entire XML Schema or inside a specific complexType definition.

**Element Declarations:** An element declaration is an association between a name and a type definition, either simple or complex. The scope of element declarations can be either global or in the scope of a containing complex type. Element declarations define what element instances are valid for an XML document conforming to that XML Schema.

Listing 2 provides an example of a simple XML Schema (modified from an example in [Griffen, 2002]).

Listing 2. Example of XML Schema

```
01 <xsd:schema xmlns:xsd="http://www.w3.org/2000/08/XMLSchema">
02
03   <xsd:element name="resumes" type="resumeTypes"/>
04
05   <xsd:complexType name="resumeTypes">
06     <xsd:sequence>
07       <xsd:element name="applicantName" type="xsd:string"/>
08       <xsd:element name="jobsAvailable" type="jobListType"/>
09     </xsd:sequence>
10     <xsd:attribute name="applicationDate" type="xsd:date"/>
11   </xsd:complexType>
12
13   <xsd:complexType name="jobListType">
14     <xsd:sequence>
15       <xsd:complexType name="job" type="jobDesc">
16         <xsd:attribute name="jobid" type="xsd:string"/>
17       </xsd:complexType name="job" type="jobDesc">
18     </xsd:sequence>
19   </xsd:complexType>
```

```

20
21 <xsd:complexType name="jobDesc">
22   <xsd:element name="title" type="xsd:string">
23   <xsd:element name="salary">
24     <xsd:simpleType>
25       <xsd:restriction base="xsd:positiveInteger">
26         <xsd:maxExclusive value="55000">
27       </xsd:restriction>
28     </xsd:simpleType>
29   </xsd:element>
30 </xsd:complexType>
31
32 </xsd:schema>

```

### 2.1.3 Previous Work on Mapping between XML Schema and Ontologies

There has been significant work done on investigating the relationship between ontologies and XML Schema and whether or not it is relevant to compare the two. In [\[Klein et al., 2001\]](#), the authors demonstrate that although ontologies and XML Schema provide data description at different levels of abstraction, it is useful to compare the languages used to represent the data, and to derive some conclusions from the similarities and differences uncovered. Using the Ontology Inference Layer (OIL), defined in [\[Horrocks et al., 2000\]](#), as the ontology language for the comparison, they highlight the languages' purposes as the core difference between them. XML Schema is concerned with defining the vocabulary and the constraints on the structure of well-formed XML documents. Ontology languages, including OIL and WSML, are concerned with providing a formal specification of a shared domain theory. Ontologies are not explicitly concerned with defining the structure of documents or even the exact representation of data items (e.g. format of a date datatype) but rather defining the meaning of the data that the documents contain. Significantly, the authors of [\[Klein et al., 2001\]](#) outline as the results of their comparison that both XML Schema and OIL can both be more expressive depending on the viewpoint adopted when defining the data model. The authors also propose a methodology for translating an ontology specification (in this case, OIL) into an XML Schema. Our approach for the translation between WSML and XML is intended to be based on those presented in [\[Klein et al., 2001\]](#); with the difference that we are focusing on lowering ontology instances to XML instances and are not interested in obtaining an XML Schema from ontology.

Other attempts in this direction, have attempted to add semantic meta-information to instance data, to make it available for tasks involving reasoning. Such an approach proposed by [\[Melnik, 1999\]](#) develops an RDF interpretation for XML documents. It considers that both structural and semantic mark-up can coexist in the same document and uses RDF to annotate existing XML documents.

Another type of approach avoids the requirement of changing or extending existing technology. In [\[Trastour et al., 2004\]](#) the authors propose two sets of mappings, the first is used for lifting XML Schemas to OWL ontologies and the second is used to translate XML documents into RDF graphs. We use a similar approach to that used in the model for the lifting operations, with the difference that we propose the creation of only a single set of mappings (between XML Schema elements and WSMO ontology meta-model) and use it for both schema lifting *and* instance lifting.

The symmetric part of the problem, the lowering, was addressed in several other works investigating the relations between ontology representation languages and document structuring techniques (schemas) [\[Klein et al., 2001\]](#). In [\[Erdmann & Studer, 1999\]](#) it is shown how an XML document type definition (DTD) is generated from a given ontology. XML instances are linked to that ontology and intelligent information integration and retrieval from the semi-structured documents (i.e. XML documents) is made possible.

## 2.2 Approaches to Grounding WSMO Ontologies to XML

The goal of this section is to describe a mechanism that will allow the author of a WSMO description of an existing service to create a mapping between the WSMO conceptual model and the XML Schema data model. Three possible approaches to this task are:

1. Create mappings at the conceptual (WSMO) level involving creating a WSMO ontology for the XML Schema used in the WSDL.
2. Use XSLT to create direct mapping between XML and the XML syntax of the WSMO ontology.
3. Use a direct mapping between the source XML data and the target WSMO ontology, using a mapping language specifically developed for this purpose.

The first approach and focus of this section is to create the mappings between the two models at the conceptual (or ontological) level. A mapping between the meta model for XML Schema and the WSMO Ontology Metamodel is defined. This allows the automatic generation of an ad-hoc ontology for a specific XML Schema. Existing WSMO mediation tools can then be used to create mappings between the WSMO ontology used by the creator of the WSMO service description and the WSMO ontology created from the XML Schema. Based on these mappings, two sets of mapping rules can be created to be applied to instance data at run-time. One set of rules will be used to translate WSMO instances to XML instances. The other rule-set will be used to translate XML instances to WSMO instances. The benefit of this approach is that all mappings are created at the conceptual level taking advantage of the more expressive mapping techniques available for mapping between ontologies. It also decouples the definition of the mappings between ontologies from how these mappings will actually be executed for instance data at runtime.

A second approach is to create direct transformations between the XML used for the WSDL messages and the XML syntax of the ontology used for the WSMO description. The most prominent transformation language for this purpose is the XSL Transformations language (XSLT) which has widespread tool support. There are two main problems with this approach. One problem is that the mappings take place only on the syntactic level. There would be no possibility to use reasoning to provide a more sophisticated mapping. Another problem is that XSLT is designed to map between hierarchically structured XML documents. Representing a WSMO ontology using the WSML/XML syntax results in a flat document structure which would make the XSLT creation awkward and unnatural.

A third approach is to create a mapping language that would allow a direct translation between XML instances and WSMO instances. In this case a new language needs to be invented specifically for this transformation. A new mapping would be required between each WSDL service description (the XML part) and the ontology used by the corresponding WSMO description. There would be a limited opportunity for re-use of mappings. Additionally it would overlook the existing and ongoing work in developing data mediation tools based on semantic descriptions.

## 2.3 Grounding WSMO Ontologies to XML by Creating Mappings at the Conceptual Level

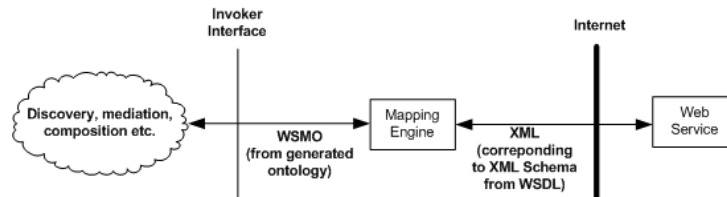
This section contains two subsections. The first of these, [section 2.3.1](#), provides an example for the first approach for grounding WSMO ontologies to XML as listed in the last section. The second subsection, [section 2.3.2](#), describes the steps required to make the grounding a reality.

### 2.3.1 Overview and Example

To get an overview of what is involved, we imagine as example, a semantic service designer with the task of providing a semantic description for an Amazon Web service selling books. We consider the example in terms of grounding the data used in the messages sent and received by the service. The designer first creates an ad-hoc WSMO ontology for book sales based on the XML Schema used in the service's WSDL description (using tool support).

In the simplest case, the designer finds that this generated ontology defines a suitable conceptual model for books and uses it when defining the messages for the service's choreography in the WSMO service description. Figure 1 illustrates the data mappings required to for this grounding.

Figure 1: Simple Use Case — No Mediation



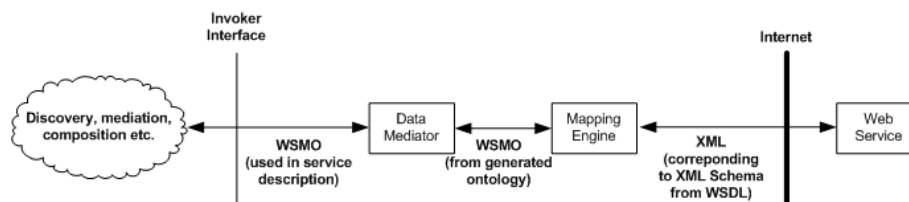
A single pair of mappings is required here:

1. to map from instances of the generated WSMO ontology to instances of the corresponding XML and
2. to map in the opposite direction - from instances of XML to instances of the corresponding WSMO ontology.

It should be possible for these mappings to be automatically generated based on the rules used to create the ad-hoc WSMO ontology from the XML Schema.

A more complex case is where the designer wishes to use an existing ontology for books to create the WSMO service description. This may be because using an established ontology might be more beneficial in terms of advertising the WSMO service description for discovery. In this case a data mediator is required as shown in Figure 2.

Figure 2: Simple Use Case - With Mediation



The data mediator is required to mediate between the ontology used in the WSMO service description and the ontology created by from the XML Schema of the WSDL. The mappings used by the mediator would be created at design time using the existing data mediation tools developed for WSMO.

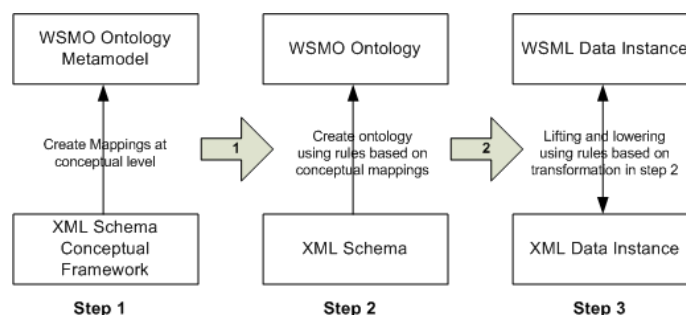
### 2.3.2 Steps to ground WSMO to XML Schema

Three distinct steps are required to ground the data part of WSMO service descriptions to the XML Schema used in WSDL:

- Define a mapping between the XML Schema Conceptual Model [[XML Schema](#)] to the WSMO Ontology metamodel.
- Create an executable description of the mappings in the first point to enable the automatic creation of ad-hoc WSMO ontologies from specific XML Schema.
- Create the bidirectional mappings rules to be used for the transformation between XML instances and WSMO instances. These mapping rules should be created at the same time as the generation of the ad-hoc WSMO ontology from an XML Schema. The creation of these mapping rules should be automatic as they should be completely derived from the actions described in the first two bullet points.

Figure 3 illustrates the activities described in the bullet points.

Figure 3: Data Model Layers for WSMO and XML



The left-most column indicates the fundamental task of creating a mapping between the XML Schema Conceptual Framework and the WSMO

Ontology Metamodel. Defining this mapping allows for the creation of executable rules that can be used to automatically *lift* any XML Schema to an equivalent ad-hoc WSMO ontology. The central column of Figure 3 depicts this lifting. The right-most column in the figure shows the bidirectional mapping rules that will be necessary to *lower* instances of WSMO data to instances of XML data and conversely lift instances of XML data to instances of WSMO data. As described earlier, *lowering* is required when a client using the WSMO description of a service for activities such as discovery, mediation and composition wants to make an invocation on the actual service using its WSDL interface description. *Lifting* is required when the service responds using instances of XML data that need to be returned to the semantics client.

The arrows labeled 1 and 2 indicate dependencies between the steps required to define WSMO grounding. Arrow 1 indicates that the mappings at the conceptual level are used to create the executable rules for lifting an XML Schema to WSMO ontology. Arrow 2 shows that during the creation of the ad-hoc WSMO ontology, bidirectional rules are created for lifting XML instances to WSMO instances and for lowering WSMO instances to XML instances.

## 2.4 Definition of Mapping from XML Schema to WSMO

In this section we provide a broad description of the mappings for each of the four primary types of XML Schema elements. For some of the types, multiple nested mappings are possible which results in quite detailed mapping descriptions. For complete detailed descriptions and listings of all the mappings, the reader is referred to [Appendix A](#). In this section we provide a representative mapping for each of the four primary XML Schema types. [Section 2.4.1](#) provides a mapping for a simpleType. [Section 2.4.2](#) provides a mapping for a complexType. [Section 2.4.3](#) provides a mapping for attributes. Finally, [section 2.4.4](#) provides a mapping for elements.

The notation  $T()$  is used in the following listings to denote the transformation function defined by the mappings.

### 2.4.1 Simple Type Definitions

For the XML Schema built-in types, no mapping is required as WSMO supports the use of the built-in types defined by the XML Schema namespace at [\[XMLSchema\]](#). Where a simple type definition is used to create a new type based on restricting one of the built-in type, the mapping results in the creation of a WSMO concept with type of the built-in type in WSMO along with an axiom that defines the restriction. Where the simpleType definition is based on a list of other simpleTypes, as in listing 3, the mapping results in the creation of a concept with an attribute resulting from the transformation of the list.

Listing 3. Mapping for a simpleType based on a List

```

T(<simpleType id=ID name=NCName any attributes>
  (annotation?, list)
</simpleType>) ->

  'concept' NCName
    T(annotation, ID) ?
    T(list)

T(<simpleType id=ID any attributes>
  (annotation?, list)
</simpleType>, X) ->

  'concept' X
    T(annotation, ID) ?
    T(list)

```

### 2.4.2 Complex Type Definitions

Complex type definitions can contain sub-components that are a mixture of elements, attributes and other complex type definitions. They can also contain keywords to indicate the correct structure for XML to be compliant with the type (sequence, all and choice). Complex types always map to a concept in WSMO. Sub components with a simple built-in type are mapped to attributes of the concept with the same built-in type. Sub components with simple types that are not built-in are mapped to attributes with the type of the mapped simple type definition. A sub component that itself is a complex type leads first to the creation of a corresponding concept. Listing 4 shows an example of the mapping for a complexType.

Listing 4. Mapping for complexType

```

T(<complexType
  id=ID ?
  name=NCName ?
  abstract=true|false ?
  mixed=true|false ?
  block=(#all | list of (extension | restriction)) ?
  final=(#all | list of (extension | restriction)) ?
  any attributes >
  (annotation?,
    (simpleContent | complexContent |
      ((group | all | choice | sequence)?,
        ((attribute|attributeGroup)*,anyAttribute?)
      )
    )
  )
)

</complexType >) ->

'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'complexType'
    'abstract' 'hasValue' ('true' | 'false') ?
    'mixed' 'hasValue' ('true' | 'false') ?
    'block' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
      { getID(T(restriction1)), getID(T(restrictionm)) } ) ?
    'final' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
      { getID(T(restriction1)), getID(T(restrictionm)) } ) ?
  'endnfp'

  ( 'simpleContentAttribute' 'ofType' getID(T(simpleContent)) |
    'complexContentAttribute' 'ofType' getID(T(complexContent)) ) |

  ( 'groupAttribute' 'ofType' getID(T(group)) |
    'allAttribute' 'ofType' getID(T(all)) |
    'choiceAttribute' 'ofType' getID(T(choice)) |
    'sequenceAttribute' 'ofType' getID(T(sequence)) ) ?

  getQName(attribute1) 'ofType' getID( T(attribute1) ) ?
  ...
  getQName(attributen) 'ofType' getID( T(attributen) ) ?
  getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?
  ...
  getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?

  T(annotation, ID) ?

```

### 2.4.3 Attributes

Attribute elements in XML Schema can have as parent, the *element* element, the schema itself or a complexType elements. When the parent is the schema itself, the attributes have schema-wide scope. We map attributes to WSMO concept definitions as shown in Listing 5. XML Schema attribute.

Listing 5. Mapping for attribute based on an internally defined simpleType

```

T(<attribute
  id=ID name=NCName default=string fixed=string form=qualified | unqualified
  use=optional | prohibited | required any attributes
  >
  (annotation?, (simpleType) ) )
</attribute > ) ->

'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'attribute'
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'use' 'hasValue' 'optional' | 'prohibited' | 'required' ?
  'endnfp'

  'attributeSimpleType' 'ofType' getID\(T\(simpleType\) \)

T(annotation, ID) ?

```

#### 2.4.4 Elements

Elements can also have schema-wide scope and, like XML Schema attributes, are mapped to WSMO concepts. Listing 6 gives an example.

Listing 6. Mapping for element Element

**With 'name' attribute:**

```

T(<element
  id=ID name=NCName substitutionGroup=QName default=string fixed=string form=qualified | unqual
  maxOccurs = nonNegativeInteger | unbounded minOccurs = nonNegativeInteger nillable = true | fals
  abstract = true | false any attributes >

  (annotation?, ((simpleType | complexType)?, (unique | key | keyref) * ) )

</element > ) ->

'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'element'
    'substitutionGroup' 'hasValue' QName ?
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'maxOccurs' 'hasValue' (nonNegativeInteger | 'unbounded') ?
    'minOccurs' 'hasValue' nonNegativeInteger ?
    'nillable' 'hasValue' ('true' | 'false') ?
    'abstract' 'hasValue' ('true' | 'false') ?
  'endnfp'

  'simpleTypeAttribute' 'ofType' getID\(T\(simpleType\) \) |
  'complexTypeAttribute' 'ofType' getID\(T\(complexType\) \)

T(annotation, ID) ?

```

## 2.5 Summary

As stated in [Klein et al., 2001] there are many differences between ontologies and XML Schema which from one viewpoint suggests that it is not a good idea to identify a mapping between them. Ontologies in many ways provide a higher level of abstraction than XML Schema and a more expressive way to deal with the specification of a conceptual model including representing relationships between concepts and the formal axioms constraining on how the concepts can be instantiated. On the other hand XML Schema provide a way to define both the vocabulary and structure that a compliant XML document must use. WSMO is less verbose than XML but it depends on the reader's

perspective which of the two languages is more human-readable. What is certain is that XML is a purely syntactic language and although the use of XSLT to transform XML documents to other formats is quite powerful, it is still rooted in syntax and takes no account of the semantics of the underlying data. Using a combination of XML and XSLT to transform between different schema definitions at the data instance level provides little scope for reuse. We believe that developing a mechanism to lift XML instances to equivalent WSMO instances based on mapping between the XML Schema and the WSMO Ontology offers a more flexible approach. We also believe that the potential for reuse of data mediation tools is high and that much of this process can be automated.

One of the challenges is how to avoid losing structural information about the XML document during translation to WSMO. For example, when mapping an instance of XML conforming to the XML Schema to an instance of a WSMO ontology fragment conforming to the created WSMO ontology, the information regarding which WSMO concepts corresponded to XML Schema elements and which to XML Schema attributes must not be lost. The mappings described in [Appendix A](#) maintain the XML Schema structural information using the non-functional properties mechanism of WSMO.

### 3. Grounding WSMO to WSDL

In the grounding of a WSMO service description to WSDL, we aim to provide all information necessary for the clients to be able to communicate with the service using Web service technologies (for example using SOAP over HTTP).

As indicated in the introduction, WSMO choreography describes the interface between the Web service and its clients. A choreography is a state machine whose states are made up of concept instances and relations between them. For the purpose of marking inputs and outputs, certain concepts may be assigned one of the *roles* "in", "out" or "shared". The client may write (create or change) instances of concepts with roles "in" and "shared", and similarly it may read instances of concepts with roles "out" and "shared". WSMO choreography also allows roles "controlled" and "static", but such concepts are not accessible from outside the choreography and as such they are not involved in grounding.

In the WSDL view of Web services, the client can send messages to a Web service and the service can send messages back to the client. The main purpose of this section is to describe how the client generates concrete messages represent writing instances (roles "in" and "shared") in the choreography state, and how existing instances (with roles "out" and "shared") within the state can be sent as the appropriate messages to the client, in effect making the client read them.

The XML contents of the messages and the networking details necessary for their transmission are captured in WSDL descriptions. [Section 3.1](#) below presents a short overview of the relevant parts of the Web Services Description Language. In case there is an existing WSDL description for a Web service, the appropriate mappings from WSMO can be established by following the rules in [section 3.2](#). Alternatively, when no WSDL description exists for a WSMO Web service, [section 3.3](#) specifies how one can be generated, so that WSDL-based tooling can be used to access the Web service.

#### 3.1 WSDL Overview

Web Services Description Language describes Web services in two levels — an XML-based reusable abstract interface and the concrete details regarding how and where this interface can be accessed. All descriptions in WSDL are centered on the Web service and all terminology follows the service's point of view, for example input messages are messages coming into the service from the network and output messages are messages generated by the service and sent to the network. The first three subsection below talk about various aspects of WSDL descriptions, based on WSDL version 2.0, namely about abstract Web service interfaces ([section 3.1.1](#)), binding them to concrete wire protocols and endpoints ([section 3.1.2](#)) and finally about the overall organization of WSDL documents ([section 3.1.3](#)). The fourth subsection details the relevant differences in the older version, WSDL 1.1.

##### 3.1.1 Web Service Interface

On the abstract level, a Web service interface is described in terms of data schemas and simple message exchanges. In particular, WSDL models *interfaces* as sets of related *operations*, each consisting of one or more messages. For example an *interface* of a ticket booking Web service can have operations for querying for a trip price and for the actual ticket booking:

Listing 7. Illustrative example of a WSDL interface

```

01 <interface name="BookTicketInterface">
02   <operation name="queryPrice" pattern=".../in-out">
03     <input element="ns:TripSpecification"/>
04     <output element="ns:PriceQuote"/>
05     <outfault ref="TripNotPossible"/>
06   </operation>
07   <operation name="bookTicket" pattern=".../in-out">
08     <input element="ns:BookingRequest"/>
09     <output element="ns:Reservation"/>
10     <outfault ref="CreditCardNotValid"/>
11     <outfault ref="TripNotPossible"/>
12   </operation>
13   <fault name="TripNotPossible" element="ns:TripFailureDetail" />
14   <fault name="CreditCardNotValid" element="ns:CreditCardInvalidityDetail" />
15 </interface>

```

In WSDL, an operation represents a simple exchange of messages that follows a specific message exchange pattern (MEP). The simplest of MEPs, "In-Only", allows a single application message to be sent to the service, and "Out-Only" symmetrically allows a single message to be sent by the service to its client. Somewhat more useful is the "Robust-In-Only" MEP, that also allows a single incoming application message but in case there is a problem with it, the service may reply with a fault message. Perhaps the most common MEP is "In-Out", which allows an incoming application message followed either by an outgoing application message or an outgoing fault message. Finally, an interesting MEP commonly used in messaging systems is "In-Optional-Out" where a single incoming application message may (but need not) be followed either by a fault outgoing message or by a normal outgoing message, which in turn may be followed by an incoming fault message (i.e. the client may indicate to the service a problem with its reply).

Particular messages (incoming, outgoing) in an operation reference XML Schema element declarations to describe the content. Fault messages, however, reference faults defined on the interface level (see above the `<outfault>` element), with the intention that semantically equivalent faults can be shared by different operations. Additionally, there may be multiple fault references for the same MEP fault message — in effect WSDL faults are typed and one operation can declare that it can result in any number of alternative faults (apart from the single success message).

### 3.1.2 Web Service Endpoints, Bindings

In order to communicate with a Web service described by an abstract interface, a client must know how the XML messages are serialized on the network and where exactly they should be sent. In WSDL, on-the-wire message serialization is described in a *binding* and then a *service* construct enumerates a number of concrete *endpoint* addresses.

A binding generally follows the structure of an interface and specifies the necessary serialization details. The WSDL specification contains two predefined binding specifications, one for SOAP (over HTTP) and one for plain HTTP. These bindings specify how an abstract XML message is embedded inside a SOAP message envelope or in an HTTP message, and how the message exchange patterns are realized in SOAP or HTTP. Due to extensive use of defaults, simple bindings only need to specify very few parameters, as in the example below. A notable exception to defaulting in binding are faults, as in SOAP every fault must have a so called fault code with two main options, Sender or Receiver, indicating who seems to have a problem. There is no reasonable default possible for the fault code.

Bindings seldom need to contain details specific to a single actual physical service, therefore in many cases they can be as reusable as interfaces, and equivalent services by different providers only need to specify the different endpoints, sharing the interface and binding descriptions.

The *service* construct in WSDL represents a single physical Web service that implements a single interface. The Web service can be accessible at multiple endpoints, each potentially with a different binding, for example one endpoint using an optimized messaging protocol with no data encryption for the secure environment of an intranet and a second endpoint using SOAP over HTTPS for access from the Internet.

Listing 8. Example of a WSDL binding and service

```

01 <binding
02     name="SOAPTicketBooking"
03     interface="BookTicketInterface"
04     type="http://www.w3.org/2004/08/wsd/soap12"
05     wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/" >
06     <fault ref="TripNotPossible" wsoap:code="soap:Receiver"/>
07     <fault ref="CreditCardNotValid" wsoap:code="soap:Sender"/>
08 </binding>
09
10 <service
11     name="DERITicketBooking"
12     interface="BookTicketInterface">
13     <endpoint
14         name="normal"
15         binding="SOAPTicketBooking"
16         address="http://deri.example.org/tickets" />
17 </service>

```

### 3.1.3 WSDL Documents

Apart from the interfaces, bindings and services described above, WSDL documents can contain further elements, enclosed in the root `<description>` element.

In order to facilitate true reuse of interfaces or bindings, WSDL documents can be modularized by using include and import mechanisms. When a WSDL document is parsed, imports and includes are resolved so the resulting model is not aware that some pieces may have come from different actual files.

As a container for data type information, WSDL documents have a section called `<types>`. Actual schemas can either be embedded directly in this section or referred to using the appropriate import statements, for example external XML Schema documents can be imported by putting the `<xs:import>` element directly in the `<types>` section. By default, WSDL uses XML Schema to describe data, but WSDL extensibility allows other data type systems to be used instead.

Finally, every element in a WSDL document can be annotated with documentation elements or it can contain extensibility elements or attributes.

### 3.1.4 Note on the differences between WSDL 2.0 and WSDL 1.1

This note details the differences between WSDL version 1.1 [\[WSDL11\]](#), a specification authored by several companies and submitted to the W3C as the basis for standardization work, and WSDL version 2, the resulting draft standard. While this document uses the cleaner version 2 of WSDL, actual deployment prefers WSDL 1.1 because WSDL 2 is not yet finished and implemented. This note aims to limit any confusion stemming from the situation that some readers may only be familiar with WSDL 1.1.

The first notable difference is that several constructs from WSDL 1.1 were renamed in WSDL 2. In particular, *portType* in WSDL 1.1 is known as *interface* in WSDL 2 and *port* in WSDL 1.1 (occurring within a *service*) is now known as *endpoint*. Also, the WSDL document root element is called *definitions* in WSDL 1.1 and *description* in WSDL 2. Importantly, the intention of all these renamed constructs is unchanged between the two WSDL versions.

A larger difference is that while WSDL 2 uses XML Schema element declarations to describe messages, WSDL 1.1 had a special construct, *message*, that contained potentially several *parts*, each referencing a single XML Schema element or type declaration. However, the use of

multiple *parts* in a single *message* is usually translatable to a single element containing a sequence of elements (one for each *part*), making the different approaches in WSDL 1.1 and in WSDL 2 equivalent for all practical purposes.

### 3.2 Grounding to existing WSDL descriptions

In the state signature of a WSMO choreography, certain concepts can be marked with role "in", "out" or "shared", and the instances of these concepts can then be accessed by the client. In terms of Web service message exchanges this means that messages coming from the client can result in the creation or updates of instances of concepts with the role "in" or "shared", and conversely messages from the service represent instances of concepts with the role "out" or "shared". This section describes how we can establish a concrete mapping between the accessible concepts and the appropriate WSDL interface messages.

Aside from mapping the choreography to a WSDL interface, we need to provide information on how the XML messages described in the interface are serialized on the wire and where they need to be sent. All this information is present in a WSDL service, therefore this section also describes how the appropriate WSDL service can be referenced from a WSMO Web service.

In the following subsections, we first discuss in [section 3.2.1](#) how the appropriate correspondences between WSDL and WSMO constructs are established. Then we describe in [section 3.2.2](#) the values of the grounding properties in various specific scenarios. This relies on using URIs to refer to WSDL components, therefore we follow with [section 3.2.3](#) that provides a way of constructing the necessary URIs. [Section 3.2.4](#) then shows an example choreography grounded to a WSDL. Finally [section 3.2.5](#) talks about how a grounded choreography could be processed and executed.

#### 3.2.1 Correspondences between WSDL and WSMO

First, let us clarify what is meant by "appropriate" WSDL messages and services, as used in the text above. In a WSMO choreography, data is represented as instances of application-specific ontological concepts and relations. In Web service messages, data is represented as XML trees conforming to application-specific schemas. The creator of the grounding mappings must know which parts of the XML trees correspond to which ontological instances. [Section 2](#) specifies how the correspondence can be described so that the machine can transform between the two forms automatically, but usually a human is required to establish and/or verify that some WSDL message actually corresponds to some ontological data. This can be done, for example, by reading the definitions and documentation of the WSDL messages and ontology concepts and deciding if they match.

A similar process must be used already when deciding that a particular WSMO Web service's choreography should be grounded to a particular WSDL interface. The remainder of this document assumes that a suitable WSDL interface has been selected for grounding and that the pairs of corresponding XML types and ontological concepts are known.

It is important to emphasize that a single input or output concept in WSMO choreography does not necessarily correspond directly to a single message in WSDL. Web services tend to use a small number of operations with relatively large and complex messages [\[WSArch\]](#), and in fact coarse-grained messages are suggested by many proponents of Web services as best practice that reduces networking overhead and simplifies the interactions. On the other hand, it has not been investigated yet whether WSMO choreography designers should make their state ontologies coarse-grained or fine-grained. Ontologies tend to be fine grained due to the principle of separation of concern - if a concept serves two orthogonal purposes, it will likely be split into two related concepts, with the hope that this might facilitate reuse.

Due to these granularity differences, we expect that a WSDL interface operation can generally correspond to one or more transition rules. We have identified several important correspondence patterns, described in the subsections below, which might be useful for designers creating a WSMO choreography and grounding it to a WSDL.

##### 3.2.1.1 Single rule for one request/response operation

When the granularity of the WSMO choreography description of a Web service matches that of the WSDL interface for this service, each request/response operation will correspond to a single transition rule. The *in* concept used in the rule's condition will be grounded to the input message of the operation, and the *out* concept whose instance is created in the rule's update statement.

For example, the operation *getStockQuote*, which takes a stock symbol as its input and returns the current price as the output, would correspond to a transition rule that matches on stock quote request instances (containing the stock symbol) and results in the creation of response instances with the appropriate prices. The concept *StockQuoteRequest* would be grounded to the input message of *getStockQuote*, and the concept *StockPrice* would be grounded to the output message.

##### 3.2.1.2 Multiple rules for an aggregate operation

In order to achieve coarser granularity, an operation in a WSDL interface may aggregate a number of logically separate actions, which could otherwise be modeled as separate operations and executed in sequence in a finer-grained interface. The choreography may model these actions separately, to achieve better clarity of intent, in an example of granularity mismatch between the WSDL interface and the choreography.

The aggregate operation will carry a number of logically separate pieces of data in the input message, and likely an equal number of responses in the output message. These pairs of input data and responses will each be handled by a different transition rule in the choreography. In this example all the *in* concepts used in the conditions of these transition rules will be grounded to the same single input message of the WSDL operation, and likewise the *out* concepts created in the update statements of the separate rules will all be grounded to the one output message of the operation.

As an example situation where this pattern would occur we can take the operation *SetResourceProperties* (from Web Services Resource Framework - WSRF), which aggregates in a single operation any number of property inserts, updates or deletions. The WSMO choreography could be modeled with three separate transition rules, each handling one of the three property change operators, and all the concepts *InsertResourceProperty*, *UpdateResourceProperty* and *DeleteResourceProperty* would be grounded to the single input message of the operation *SetResourceProperties*.

##### 3.2.1.3 Grounding one concept to multiple operations

This pattern does not deal directly with the correspondence between transition rules and operations, instead it describes a special case that can occur in the previous patterns, of which a grounding designer should be aware.

In certain situations, detailed in the following paragraphs, multiple operations in a WSDL interface can transfer the same *in* or *out* concept, and

each such concept has to be grounded to all the appropriate messages.

For *in* concepts, it is possible that two or more operations do in fact have the same input. For example if an interface contains an aggregate operation (like the *SetResourceProperties* above) together with the constituent operations (in WSRF those would be *InsertResourceProperties*, *UpdateResourceProperties* and *DeleteResourceProperties*, each only allowing one particular change operator). In this case, the concept *InsertResourceProperty* would be grounded to both the input message of the operation *InsertResourceProperties* and to the input message of *SetResourceProperties*.

For *out* concepts, we have a situation of multiple operations with different inputs, but with the same output. For example the Amazon Web Services interface has methods *ItemSearch* and *ItemLookup*, which both return a list of matching items. The input message of *ItemSearch* specifies the search criteria, the input message of *ItemLookup* carries a list of particular item IDs, but the output messages of these two operations are exactly the same - an *ItemList*. The two operations would be modeled as two different transition rules in the choreography (both according to [section 3.2.1.1](#)) but the concept *ItemList* would have to be grounded to the output messages of both of the operations at the same time.

### 3.2.2 Grounding values

As already mentioned, a WSMO choreography assigns externally accessible concepts to specific *roles*, which are either "in", "out" or "shared". Additionally, WSMO choreography requires that all accessible concepts also provide a grounding mechanism by pointing to it with its URI after the keyword *withGrounding*. There can be multiple grounding specifications that will describe the allowed values for this property and what they mean. In this specification, we describe what values the property can have to ground the concepts to WSDL messages, using the following rules:

1. grounding may be specified using multiple URI values (cardinality issues are discussed in the following rules)
2. grounding of a concept with role "in" or "shared" must contain a URI identifying an appropriate WSDL *input* or *input fault* message
3. grounding of a concept with role "in" MUST NOT contain a URI identifying a WSDL *output* or *output fault* message
4. grounding of a concept with role "out" or "shared" must contain a URI identifying an appropriate WSDL *output* or *output fault* message
5. grounding of a concept with role "out" MUST NOT contain a URI identifying a WSDL *input* or *input fault* message
6. one concept can be grounded to multiple WSDL messages with the same direction because one piece of data can in fact be transmitted alternatively in different messages, as shown above in [section 3.2.1.3](#)
7. multiple concepts can be grounded to a single WSDL message because Web service messages should be coarse-grained and transfer as much data as available (as discussed above in [section 3.2.1](#), especially [section 3.2.1.2](#)), and the data grounding can map multiple instances to a single XML tree

Note that the rules above treat WSDL fault messages as equivalent with normal application messages. This is the case because fault messages can also carry useful information and so they can also be received and translate into "in" or "shared" concept instances or they can be emitted from "out" or "shared" concept instances.

Finally, in order to provide the on-the-wire serialization and endpoint information, we need to link from a WSMO Web service to a WSDL service. We propose a new *endpointDescription* non-functional property, which contains a URI identifying the appropriate WSDL service. This new non-functional property is defined in [Appendix A](#).

### 3.2.3 URIs for identifying WSDL components

As described in Appendices A.2 and C of [\[WSDL\]](#), the URIs for WSDL 2 components are created by combining the namespace URI with fragment identifiers that unambiguously identify components on any level of granularity within a WSDL file. For our purposes, we need URI references for Interface Message References and Interface Fault References (both owned by operation components), and for Services.

The fragment identifier for Interface Message References is defined as `wsdl:interfaceMessageReference(interface/operation/message)` with the three parts in parentheses replaced with the following:

1. *message* is the {message label} property of the Interface Message Reference component (see below),
2. *operation* is the local name of the operation that contains the message,
3. *interface* is the local name of the interface owning the operation.

The {message label} property is defined in the message exchange pattern used by the operation and indirectly identifies the direction of the message. In all the MEPs predefined by WSDL 2, the message labels are either "In" or "Out" (note the capitalization) and the corresponding messages have directions matching the labels.

Similarly, the fragment identifier for Interface Fault References is defined as

`wsdl:interfaceFaultReference(interface/operation/message/fault)` and the four parts in parentheses are replaced as follows:

1. *fault* is the local name of the interface fault referenced by the fault message
2. *message* is the {message label} property of the Interface Fault Reference component (see below for an important difference from {message label} on Interface Message References),
3. *operation* is the local name of the operation that contains the fault message,
4. *interface* is the local name of the interface owning the operation.

Note: the {message label} property does not directly point to a fault message in the message exchange pattern, instead it points to the message to which this fault is related. The kind of relation depends on the *faulting rules* used by the message exchange pattern. In particular, in some MEPs a fault can replace any message after the first one (in the usual input/output MEP, a fault can replace the output message), thus ending the message exchange; the message label is then the label of the replaced message and the direction of the fault is the same as that of the replaced message. On the other hand in other MEPs any application message may trigger a fault (the robust in-only MEP, for example, consists of a single incoming message and an optional fault message back, in case something is wrong with the incoming message) and in this case the message label is the label of the message that triggered the fault and the direction is **opposite** — the fault returns from the receiver of the original message. For illustration, an incoming fault reference may have message label "Out". Therefore to compute the URI for a fault reference, if the message label is not explicit in the WSDL file (it may be defaulted), the knowledge of the operation's MEP is necessary.

Finally, the fragment identifier for WSDL Services is defined as `wsdl:service(service)` where the *service* parameter is the local name of the WSDL service.

For instance, if *BookTicketInterface* from [section 3.1.1](#) and the *DERITicketBooking* service from [section 3.1.2](#) are defined in a WSDL file with the target namespace "http://example.com/", the URIs for the two messages and two faults in operation *bookTicket* and for the whole service will be the following:

```

http://example.com/#wsdl.interfaceMessageReference(BookTicketInterface/bookTicket/In)
http://example.com/#wsdl.interfaceMessageReference(BookTicketInterface/bookTicket/Out)
http://example.com/#wsdl.interfaceFaultReference(BookTicketInterface/bookTicket/Out/CreditCardNotValid)
http://example.com/#wsdl.interfaceFaultReference(BookTicketInterface/bookTicket/Out/TripNotPossible)
http://example.com/#wsdl.service(DERITicketBooking)

```

When using WSDL 1.1, because there is no such standard way of creating the reference URIs, we simply adapt the above approach, using the appropriate *portType* name where interface name is expected and reusing the strings "In" and "Out" in lieu of input and output message labels.

### 3.2.4 Grounding example

The following is a simple choreography for a ticket booking service (with the WSDL interface specified earlier in [listing 19](#)), including highlighted grounding information for its accessible concepts. This ontology has been adapted from [\[Feier, 2005\]](#).

Listing 9. Example choreography with concrete grounding.

```

01 namespace { _ "http://example.org/bookTicket#",
02   dc _ "http://purl.org/dc/elements/1.1#",
03   tr _ "http://example.org/tripReservationOntology#",
04   wsml _ "http://www.wsmo.org/wsml/wsml-syntax#",
05   po _ "http://example.org/purchaseOntology#"
06 }
07
08 ontology _ "http://example.org/BookTicketInterfaceOntology#"
09   nonFunctionalProperties
10     dc#title hasValue "Book Ticket Interface Ontology"
11     dc#creator hasValue "DERI Innsbruck"
12     dc#description hasValue "an ontology that redefines concepts and
13       relations from other ontologies in order to reuse them in the
14       choreography and orchestration; two additional non-functional
15       properties are defined for the targeted concepts and relations:
16       mode and grounding"
17     dc#publisher hasValue "DERI International"
18   endNonFunctionalProperties
19
20   importsOntology { _ "http://www.example.org/tripReservationOntology",
21     _ "http://www.wsmo.org/ontologies/purchaseOntology"
22   }
23
24   concept reservationRequest subConceptOf tr#reservationRequest
25   concept reservation subConceptOf tr#reservation
26   concept temporaryReservation subConceptOf tr#reservation
27   concept creditCard subConceptOf po#creditCard
28   concept negativeAcknowledgement
29
30   webService _ "http://example.org/BookTicketService#"
31     nonFunctionalProperties
32       wsml#endpointDescription _ "http://example.com/#wsdl.service(DERITicketBooking)"
33     endNonFunctionalProperties
34
35   interface BookTicketInterface
36     choreography BookTicketChoreography
37     stateSignature
38     importsOntology _ "http://example.org/BookTicketInterfaceOntology#"
39     in
40       reservationRequest withGrounding
41         _ "http://example.com/#wsdl.interfaceMessageReference
42           (BookTicketInterface/bookTicket/In)"
43       creditCard withGrounding
44         _ "http://example.com/#wsdl.interfaceMessageReference
45           (BookTicketInterface/bookTicket/In)"
46     out
47       reservation withGrounding
48         _ "http://example.com/#wsdl.interfaceMessageReference
49           (BookTicketInterface/bookTicket/Out)"
50       negativeAcknowledgement withGrounding
51         _ "http://example.com/#wsdl.interfaceFaultReference
52           (BookTicketInterface/bookTicket/Out/CreditCardNotValid)"

```

53	controlled
54	temporaryReservation
55	transitionRules
56	[...]

### 3.2.5 Processing grounding information

From the point of view of WSDL, Web services are software entities that communicate by sending and receiving messages. In this model, both services and their clients must be able to choose at the appropriate times that they want to send a message or that they want to be able to receive a message. These decisions are currently usually hard-coded in the service or client implementation. By describing the choreography interface, WSMO attempts to automate the task of using previously unknown Web services.

WSMO Choreography describes the service's interface as a state machine that communicates with its clients using what we call "accessible" concepts in its state. The client can write instances of concepts with role "in" and "shared", and it can read instances of concepts with role "out" and "shared". For grounding in WSDL, these reads and writes must be translated to sending and receiving messages. This section specifies how a client following (executing) a service's choreography interface makes decisions to send or to expect receiving the messages that the choreography's accessible concepts are grounded to.

To clarify — this section does not specify how a choreography is executed, it only assumes that the client, after discovering a service, can follow the service's choreography, for example by executing it in its (the client's) local choreography engine.

To be able to emit messages to the service, the client must have the necessary data available to it. At least for the initial message that starts the conversation, the message data comes from the client's previous knowledge. For example, the client may know the user's detailed goal, it may have the results of a previously invoked Web service, or it may have the capability of directly and interactively requesting the necessary data from the user. In the remainder of this section, we simply assume the client has some data available to it that it may send to the Web service. From the point of view of the state machine, this available data is in the state.

The process followed by the client is as follows:

1. The client initiates the choreography, i.e. starts its execution.
2. When a rule fires having used some "in" or "shared" instances in its condition, these instances are marked internally to be sent to the service.
3. Whenever the client has some instances marked to be sent, it will perform the following steps:
  - a. Select the WSDL messages to which the instances are grounded.
  - b. Filter out the WSDL messages to which multiple concepts are grounded and some of them are not available to the client (see rule 7 in [section 3.2.2](#)).
  - c. From the remaining WSDL messages, select those that the client can currently send (i.e. the client has the initiative in the MEP of the message's operation; see below for more discussion of this point).
  - d. If more than one message remains, one should be selected using some out-of-bounds mechanism that we do not specify.
  - e. Send the selected message to the service, removing the "to be sent" mark from the instances sent in the message.
4. As the rules in the choreography fire, the update statements may change some instances with the mode "out" or "shared". The client's choreography engine should not attempt to perform these updates, instead the client will now expect that the service sends a message with the appropriate new data.
5. From the client's point of view, the conversation may end when the client receives the required data from the service.

Above, the rule 3c talks about messages that the client can currently send. While it would seem that the client may choose to send any message at any moment, the WSDL contract constrains this behavior: if an operation in WSDL has multiple messages in its message exchange pattern, these messages can only be sent in the temporal sequence specified by the MEP. For example, in an out-in operation (request/response from the service), the in message (the response) cannot be sent before the out message (the request). In other words, if the client could send a message but the operation mandates that another message must precede it, the client must wait until that preceding message has been sent.

It is an open question on whether the client can stop the conversation with the service when it gets the response it seeks (see point 5 above); such a situation might occur when a client only needs a part of a service's functionality. This consideration is left for later, though, as examples of such situations are not currently clear.

## 3.3 Generating WSDL from WSMO choreography

The previous section has specified how a WSMO Web service description (and especially its choreography) can be grounded to a concrete existing WSDL interface. However, we can envision that Web services design can start on the semantic level, i.e. first creating a WSMO description. A Web service can even be implemented on the semantic level in frameworks like WSMX. Still, such a Web service should be able to receive and emit XML messages. This section, by providing a set of rules for creating a WSDL description for a WSMO Web service, effectively proposes a **default grounding**, a default communication protocol for WSMO Web services.

WSMO describes the interface between a Web service and its clients using the so called choreography. In a choreography, ontology concepts can be marked with roles "in", "out" and "shared" to indicate that they can be read or written by the environment, effectively by the client. We call these concepts and their instances "accessible". Since Web services communicate by sending messages, receiving a message from the client means writing an accessible instance, and reading an accessible instance is substituted by the service sending the instance in a message to the client.

The following subsections first describe the exact rules for generating the default WSDL description from a WSMO Web service and its choreography (in [section 3.3.1](#)), then show an example of a generated WSDL description in [section 3.3.2](#) and finally discuss the limitations of our default binding approach in [section 3.3.3](#).

### 3.3.1 Rules for generating WSDL

The following is a skeleton of a WSDL document generated from a WSMO Web service and its choreography:

Listing 10. Skeleton of a generated WSDL document

```

01 <description xmlns="http://www.w3.org/2005/05/wsd"
02     targetNamespace="Web service namespace ID"
03     xmlns:xs="http://www.w3.org/2001/XMLSchema"
04     xmlns:soap="http://www.w3.org/2005/05/wsd/soap" >
05     xmlns:data="ontology ID" >
06
07 <types>
08   <xs:import namespace="http://www.wsmo.org/wsml/wsml-syntax#"
09     schemaLocation="http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-xml-syntax.xsd" />
10   <xs:schema targetNamespace="ontology ID" >
11     element declarations
12   </xs:schema>
13 </types>
14
15 <interface name="Web service local name" >
16   operations
17 </interface>
18
19 <binding name="DefaultSOAPBinding" >
20   type="http://www.w3.org/2004/08/wsd/soap12"
21   soap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/" >
22 </binding>
23
24 <service name="Web service local name"
25   interface="Web service local name" >
26   <endpoint name="Web service local name"
27     binding="DefaultSOAPBinding"
28     address="Web service endpoint address" />
29 </service>
30 </description>

```

In the listing above, the **emphasized** items are placeholders as follows:

#### Web service namespace ID

If the IRI of the WSMO Web service can be split into a namespace/local name pair (called sQName in [de Bruijn, 2005]), the *Web service namespace ID* is the namespace part. Otherwise, it is the whole Web service IRI.

#### Web service local name

If the IRI of the WSMO Web service can be split into a namespace/local name pair, the *Web service local name* is the local name part. Otherwise, it is "service".

#### Web service endpoint address

If the WSMO Web service has a non-functional property *endpointAddress* (see Appendix A), its value becomes the *Web service endpoint address*. Otherwise this address must be provided externally when generating the WSDL.

#### ontology ID

The IRI of the state ontology used by the choreography of the WSMO Web service.

#### element declarations

XML Schema element declarations generated for all the accessible concepts as described below.

#### operations

WSDL operations generated for all the accessible concepts as described below.

For every accessible concept in the choreography state signature we create an XML Schema element declaration according to the following template:

Listing 11. Skeleton of a generated XML Schema element declaration

```

01 <xs:element name="concept name">
02   <xs:complexType>
03     <xs:sequence>
04       <xs:element ref="wsml:instance" />
05     </xs:sequence>
06   </xs:complexType>
07 </xs:element>

```

In the listing above, the *concept name* is the local name of the accessible concept. The generated element declarations populate the schema in the resulting WSDL document. The content of the generated element is a WSML/XML instance element, i.e. we reuse WSML/XML instead of generating an XML Schemas for WSMO concepts, but generating such schemas would also be possible; for now we leave this as potential future work.

Next, for every "in" and "shared" concept in the choreography state signature we create an input WSDL operation according to the following template:

Listing 12. Skeleton of a generated input WSDL operation

```

01 <operation name="writeconcept name"
02     pattern="http://www.w3.org/2005/05/wsd/in-only">
03   <input element="data:concept name" />
04 </operation>

```

And finally, for every "out" and "shared" concept in the choreography state signature we create an output WSDL operation according to the following template:

Listing 13. Skeleton of a generated output WSDL operation

```

01 <operation name="concept nameUpdated"
02     pattern="http://www.w3.org/2005/05/wsd/out-only">
03   <output element="data:concept name" />
04 </operation>

```

In the two listings above, the *concept name* is the local name of the accessible concept, i.e. the name of the element declaration generated for this concept. For example, for a "shared" concept Vehicle we will generate an element declaration called Vehicle and two operations, readVehicle and writeVehicle. The generated operations populate the interface in the resulting WSDL document.

With the full WSDL generated, it is straightforward to annotate the WSMO description with grounding information as described in [section 3.2.2](#).

### 3.3.2 Example for the default grounding

The following listing is a definition of a Web service adapted from [Listing 21](#). The Web service here includes the highlighted endpoint address, otherwise it is unchanged.

Listing 14. Example WSMO Web service description with endpoint address

```

01 namespace { "http://example.org/bookTicket#",
02   wsml _ "http://www.wsmo.org/wsml/wsml-syntax#" }
03
04 webService _ "http://example.org/BookTicketService#"
05   nonFunctionalProperties
06     wsml#endpointAddress _ "http://example.org/bookTicketSOAPEndpoint"
07   endNonFunctionalProperties
08
09   interface BookTicketInterface
10     [...]

```

The WSDL generated for this Web service would look like this:

Listing 15. Example WSMO Web service description

```

01 <description xmlns="http://www.w3.org/2005/05/wsd"
02     targetNamespace="http://example.org/"
03     xmlns:xs="http://www.w3.org/2001/XMLSchema"
04     xmlns:soap="http://www.w3.org/2005/05/wsd/soap" >
05     xmlns:data="http://example.org/BookTicketInterfaceOntology" >
06
07   <types>
08     <xs:import namespace="http://www.wsmo.org/wsml/wsml-syntax#"
09       schemaLocation="http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-xml-syntax.xsd" />
10     <xs:schema
11       targetNamespace="http://example.org/BookTicketInterfaceOntology" >
12       <xs:element name="reservationRequest">
13         <xs:complexType>
14           <xs:sequence>
15             <xs:element ref="wsml:instance" />
16           </xs:sequence>
17         </xs:complexType>
18       </xs:element>
19       <xs:element name="reservation">
20         <xs:complexType>
21           <xs:sequence>
22             <xs:element ref="wsml:instance" />
23           </xs:sequence>
24         </xs:complexType>

```

```

25     </xs:element>
26     <xs:element name="creditCard">
27       <xs:complexType>
28         <xs:sequence>
29           <xs:element ref="wsml:instance" />
30         </xs:sequence>
31       </xs:complexType>
32     </xs:element>
33     <xs:element name="negativeAcknowledgement">
34       <xs:complexType>
35         <xs:sequence>
36           <xs:element ref="wsml:instance" />
37         </xs:sequence>
38       </xs:complexType>
39     </xs:element>
40   </xs:schema>
41 </types>
42
43 <interface name="BookTicketService" >
44   <operation name="writeReservationRequest"
45     pattern="http://www.w3.org/2005/05/wsdl/in-only">
46     <input element="data:reservationRequest" />
47   </operation>
48   <operation name="reservationUpdated"
49     pattern="http://www.w3.org/2005/05/wsdl/out-only">
50     <output element="data:reservation" />
51   </operation>
52   <operation name="writeCreditCard"
53     pattern="http://www.w3.org/2005/05/wsdl/in-only">
54     <input element="data:creditCard" />
55   </operation>
56   <operation name="negativeAcknowledgementUpdated"
57     pattern="http://www.w3.org/2005/05/wsdl/out-only">
58     <output element="data:negativeAcknowledgement" />
59   </operation>
60 </interface>
61
62 <binding name="DefaultSOAPBinding" >
63   type="http://www.w3.org/2004/08/wsdl/soap12"
64   wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/" >
65 </binding>
66
67 <service name="BookTicketService"
68   interface="BookTicketService" >
69   <endpoint name="BookTicketService"
70     binding="DefaultSOAPBinding"
71     address="http://example.org/bookTicketSOAPEndpoint" />
72 </service>
73 </description>

```

### 3.3.3 Limitations of the default grounding

If we compare the WSDL interface in [Listing 7](#) with the generated interface in [Listing 15](#), it is apparent that a hand-crafted WSDL presents a more natural and usable interface than one generated from a WSMO choreography. This is the effect of the limitations of the default grounding, as detailed in this subsection.

Perhaps the most visible limitation is that the default grounding cannot generate in-out (or request/response) operations. This is because we cannot easily guess a meaningful correlation between some concepts being written in the choreography state and some others being created, especially if there are intermediate steps, like the partial temporaryReservation in our example.

On a similar note, the default grounding cannot guess which concepts could be transmitted together in a single WSDL message, therefore distinct operations have to be generated for all accessible concepts.

Further, the default grounding cannot distinguish between normal application data and data indicating failure, which are distinguished in WSDL as normal messages and faults. The difference between normal messages and faults is in any case subjective and not unambiguous. For these reasons the generated WSDL interfaces do not contain faults.

In most WSDL interfaces, operations usually begin with a message from the client, implying that a response channel will be available for this operation, which is true, for example, for Web services using synchronous HTTP. In contrast, the default grounding generates out-only

operations (sending a single message from the service to the client) for "out" concepts, so the client must be able to indicate where it can receive such messages. To do so, we recommend the use of the WS-Addressing [\[WSAddressing\]](#) `wsa:ReplyTo` header. At the time of writing this, there is no agreed way of describing this behavior in the generated WSDL, but the Web Services Addressing working group at W3C plans to include a marker for this purpose.

And finally, due to the use of one-way "fire-and-forget" operations, any faults generated by the receiver of the messages will be lost. WSDL 2 currently provides robust one-way message exchange patterns (MEPs), allowing the creation of operations that only transmit a single application message, but may optionally transmit a fault back. These MEPs are not supported in the standard bindings (SOAP and HTTP), but if they get supported in the future, we may consider using them instead of the "fire-and-forget" MEPs.

To conclude, the default grounding has its uses, especially when used only as the concrete communication protocol for two WSMO-enabled nodes. For interoperability with existing syntactic-level Web services we suggest that a suitable WSDL description be created by the service provider and then the WSMO description of the service can be grounded to this WSDL using the rules of [section 3.2](#).

## 4. Conclusions

This document defines WSMO Grounding by presenting two independent areas of relationship between WSMO and the syntactical descriptions of a Web service: data in WSMO ontologies has to be mapped to XML data, usually described using XML Schema, and the functional and behavioral service descriptions in WSMO have to be related to the description construct present in WSDL. Using the grounding specified in this document, WSMO Web services can interoperate with currently deployed SOAP Web services and client frameworks.

## References

- [de Bruijn, 2005]** J. de Bruijn (editor): *The Web Service Modeling Language WSML*, version 0.2 available at <http://www.wsmo.org/TR/d16/d16.1/v0.2/20050320/>
- [Erdmann & Studer, 1999]** M. Erdmann and R. Studer: *Ontologies as Conceptual Models for XML Documents*. Twelfth Workshop on Knowledge Acquisition, Modeling and Management, KAW 1999, Alberta, Canada
- [Feier, 2005]** C. Feier (editor): *WSMO Primer*, version 0.2 available at <http://www.wsmo.org/TR/d3/d3.1/v0.2/20050401/>
- [Griffen, 2002]** G. Griffen: *XML and SQL Server 2000*, p32, New Riders Publishing, 2002, ISBN 0-7357-1112-7
- [Horrocks et al., 2000]** I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. Van Harmelen, M. Klein, S. Staab, and R. Studer: *OIL: The Ontology Inference Layer, Technical Report IR-479*, Vrije Universiteit Amsterdam, September 2000
- [Klein et al., 2001]** M. Klein, D. Fensel, F. van Harmelen, I. Horrocks: *The Relation between Ontologies and XML Schemas*. Linköping Electr. Art. in Comp. and Inf. Sci. 6 (2001)
- [Martin-Recuerda et al., 2005]** F. Martin-Recuerda, B. Sapkota (editors): *WSMX Triple-Space Computing*, version 0.1, available at <http://www.wsmo.org/TR/d21/v0.1/>
- [Melnik, 1999]** S. Melnik: *Bridging the Gap between RDF and XML*, <http://www-db.stanford.edu/~melnik/rdf/fusion.html>
- [Roman et al. 2004]** D. Roman, U. Keller, H. Lausen (editors): *Web Service Modeling Ontology - Standard (WSMO - Standard)*, version 1.0 available at <http://www.wsmo.org/2004/d2/v1.0/>
- [Roman et al. 2005]** D. Roman, J. Scicluna, C. Feier, M. Stollberg, and D. Fensel : *Ontology-based Choreography and Orchestration of WSMO Services*, WSMO Deliverable D14, v0.2 available at <http://www.wsmo.org/TR/d14/v0.2/20050702/>
- [Trastour et al., 2004]** D. Trastour, M. Ferdinand, C. Zircpins: *Pragmatic Reasoning- Support for Web-Engineering: Lifting XMLSchema to OWL*, ICWE 2004, Munich, Germany.
- [WSAddressing]** M. Gudgin, M. Hadley (editors): *Web Services Addressing 1.0 - Core*, W3C Working Draft, 31 March 2005, available at <http://www.w3.org/TR/2005/WD-ws-addr-core-20050331/>
- [WSArch]** D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard (editors): *Web Services Architecture*, W3C Working Group Note, 11 February 2004, available at <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>
- [WSDL]** R. Chinnici, J-J. Moreau, A. Ryman, S. Weerawarana (editors): *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Working Draft, 10 May 2005, available at <http://www.w3.org/TR/wsd120/>
- [WSDL11]** E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001, available at <http://www.w3.org/TR/2001/NOTE-wsd1-20010315>
- [XMLSchema]** H. Thompson, D. Beech, M. Maloney, N. Mendelsohn (editors): *XML Schema part 1: Structures*, W3C Recommendation, 2001, available at <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>

## Appendix A. Definition of Mappings from XML Schema to WSMO

### A.1 Introduction

This section defines the mappings from XML Schema to WSMO for each XML Schema component in turn. We construct mapping transformation functions for the XML Schema components to WSMO having the following form:

$T(\text{XML\_Schema\_component}) \rightarrow \text{WSMO\_Fragment}$

where `XML_Schema_component` is an XML representation of an XML Schema component and `WSMO_Fragment` is a fragment of WSMO written using WSML. Recursive calls are allowed and by applying these transformation functions to the XML Schema components, WSMO

fragments are obtained.

The notation,  $T( \dots )$ , is used to define the mapping functions. It should be noted that expressions like:

$T( \text{argument} ) \rightarrow \text{result}$

don't have anything to do with any kind of rules from particular logical formalism; the semantics of these constructs is that by applying the function  $T$  on the given argument a particular result is obtained.

We also define helper functions that can be used as a shorthand by multiple mappings. The helper functions return another helper function, a QName or an identifier that can be used to uniquely identify a WSMO element. The notation for the helper functions is:

$\text{helperFunctionName}( \text{argument} ) \rightarrow \text{result}$

As before the semantics of the helper functions is that by applying them to the input parameters, the result is obtained. An example of a helper function is:

$\text{retrieve}( \langle \text{restriction id=ID, base=QName, any attributes} \rangle \dots \langle / \text{restriction} \rangle ) \rightarrow \text{QName}$

In this example, the expression, *any attributes*, indicates any additional attributes with non-schema namespace as defined in [\[XMLSchema\]](#). This is a feature that provides for extensibility in XML Schema. In this version of this document, we do not consider the transformation of such additional attributes.

Comments are denoted in the mapping listings using the notation borrowed from programming languages such as Java and C++:

// any comment

### A.1.1 Structure of the Appendix

The following subsection defines the helper functions used in the transformation mappings. Subsequent sections provide the detailed mappings for each of the following XML Schema elements (and their sub-elements): *simpleType*, *complexType*, *element* and *attribute*.

## A.2 Helper Functions

Helper functions are used to retrieve an element from a complex expression. The retrieved element may then be used to provide a name or a unique identifier to represent the expression. Complex expressions may be an XML representation of an XML Schema element or they may be a WSMO fragment represented in WSML.

### Notes:

We use the '?' in the transformation function to mark that an element is optional. Its correspondent transformation is also marked by a '?'.

We use the '\*' in the transformation function to mark that an element can occur zero or multiple times.

We use the '|' in the transformation function to denote a disjunction between elements e.g. *X hasValue true | false* means the value of X is either TRUE or FALSE.

The results of all calls of any  $T( )$  functions inside any helper functions must be appended to the transformed document.

```

// Return a QName uniquely identifying a restriction
getQName(<restriction id=ID base=QName any attributes>
...
</restriction> ) -> QName

// Return the QName uniquely identifying the explicitly typed list
getQName(<list id=ID itemType=QName any attributes>
(annotation?)
</list> ) -> QName

// Return the QName uniquely identifying the list whose type is defined by the contained simpleType definitic
getQName(<list id=ID any attributes>
(simpleType)
</list> ) -> getQName(simpleType)

// Return a QName uniquely identifying the restriction defining the simpleType
getQName(<simpleType id=ID any attributes>
(annotation, restriction)
</simpleType>, X ) -> getQName(restriction)

// Return the id of the specified entity
getId( ) -> id

// Generate and returns a unique identifier for the specified XML Schema element.
// This uniques identifier is then used to create a unique name for the WSMO concept
// that results from a transformation.
getUniqueID( ) -> uniqueId

// Return a QName uniquely identifying a complexContent element
getQName(<complexContent id=ID mixed=true|false any attributes >
...
<complexContent> ) -> QName

// Get the local fragment (without namespace prefix) of a qualified name.
getLocalPart(QName) -> Local part of a QName

```

### A.2.1 Handling Name and ID XML Schema element attributes

Most of the XML Schema elements described below have optional name and id attributes. The name attribute where present is used in the mapping to provide the name of the derived WSMO entity. Where the name is not present, a variable (usually X) is included as an additional argument to the mapping e.g. `T(<simpleType> ... </simpleType>, X)`. The additional variable should provide a value for the name of the derived WSMO element

In the cases where XML Schema simpleTypes do not have an *id* attribute, one of the helper functions `getUniqueID()` or `getId()` is used. The examples in the following sections show the cases for simpleTypes having the *id* attribute. The transformation for simpleTypes without *id* would use `getUniqueID()` instead.

### A.3 Mapping for simpleType

A simpleType can have an optional child component of an *annotation* along with one of the following components: a *restriction*, a *list* or a *union*. A simpleType can also optionally have the attributes *id* and *name*. The *id* attribute specifies a unique ID for the element. It is also possible to include additional attributes with with a non-schema namespace.

The *name* attribute is required if the simpleType element is a child of the schema element, otherwise it is not allowed. In general, simpleTypes are mapped to a WSMO concept. Where the simpleType has a *name* attribute, this is used as the name for the concept. If no *name* attribute is present, a helper function is used to create a unique name for the concept that results from the transformation.

The mapping of a single simpleType may result in more than one WSMO components. In such cases, where the simpleType has an *id* attribute, this is used to link the components together. Where there is no *id* attribute, a helper function is used to create a unique identifier.

Listing A.2 shows the XML representation of an XML Schema simpleType definition taken from [\[XMLSchema\]](#). This forms the basis for building the mappings to WSMO. In the rest of the section, we look at the definition of helper functions and the definition of the transformations for each of the possible alternatives.

Listing A.2. XML Representation of simpleType Definition Schema Components

```

<simpleType id=ID name=NCName any attributes>
  (annotation?, (restriction | list | union))
</simpleType>

  <annotation id=ID any attributes>
    (appinfo | documentation)*
  </annotation>

    <appInfo source=anyURL>
      Any well-formed XML content
    </appInfo>
    <documentation source=URI_reference xml:lang=language>
      Any well-formed XML content
    </documentation>

  <restriction id=ID base=QName any attributes>
    Content for simpleType:
    (annotation?,
      (simpleType?,
        (minExclusive|minInclusive|maxExclusive|maxInclusive|totalDigits|
          fractionDigits|length|minLength|maxLength|enumeration|whiteSpace|pattern)*
        ))
  </restriction>

  <list id=ID itemType=QName any attributes>
    (annotation?,(simpleType?))
  </list>

  <union id=ID memberTypes="list of QNames" any attributes>
    (annotation?,(simpleType*))
  </union>

```

### A.3.1 Mapping for a simpleType based on a Restriction with a 'name' Attribute

In the mapping definition below, as in the other definitions, the *annotations* component of the XML Schema simpleType is optional.

Listing A.3. Map simpleType with name Attribute

```

T(<simpleType id=ID name=NCName any attributes>
  (annotation?,restriction)
</simpleType> ) ->

  'concept' NCName 'subConceptOf' QName(restriction)
    T( annotation, ID ) ?
    T( restriction)

```

The listing shows the transformation of a simpleType defined as a restriction. The restriction can be on a built-in type of the XML Schema or on another simpleType definition. The transformation maps the simpleType to a concept with the name, *NCName*. This concept is defined as a subconcept identified by the QName of the transformed restriction. We define the transformation of the restriction and the annotation later in this section.

### A.3.2 Mapping for a simpleType based on a Restriction without a 'name' Attribute

Listing A.4. Map simpleType without a name Attribute

```

T(<simpleType id=ID any attributes>
  (annotation?,restriction)
</simpleType>, X ) ->

  'concept' X 'subConceptOf' getQName(restriction)
    T(annotation, ID) ?
    T(restriction)

```

In this case, the simpleType has no name. An additional parameter, X, is added to the transformation to provide the name for the WSMO concept that will be the result of the transformation. An example of making a call to this transformation would be:

```
T( simpleType, ( "simpleType_" + getUniqueld( ) ) )
```

where getUniqueld( ) returns a unique ID for the transformed simpleType being transformed and the prefix "simpleType\_" is chosen as a convention for naming the concept resulting from the transformation..

### A.3.3 Mapping for a simpleType based on a List

The following listing shows two mappings. Both are for simpleTypes based on a list subcomponent. The first mapping is for a simpleType with a *name* attribute. The second mapping is for a simpleType without a *name* attribute.

Listing A.5. Map simpleType based on a List

```

T(<simpleType id=ID name=NCName any attributes>
  (annotation?, list)
</simpleType>) ->

  'concept' NCName
    T(annotation, ID) ?
    T(list)

T(<simpleType id=ID any attributes>
  (annotation?, list)
</simpleType>, X ) ->

  'concept' X
    T(annotation, ID) ?
    T(list)

```

The result of transformation of the list is defined inside the concept that is the product of the transformation of the simpleType in this case. For this reason there is no need to include ID as an argument for the transformation. This is in contrast to the result of the transformation of the restriction shown earlier. The transformation for the list is shown later in this section.

### A.3.4 Mapping for a simpleType based on a Union

Similarly to the transformations for simpleTypes based on lists, listing 11 shows two mappings. In this case, both are for simpleTypes based on a union subcomponent. The first mapping is for a simpleType with a *name* attribute. The second mapping is for a simpleType without a *name* attribute.

Listing A.6. Map simpleType based on a Union

```

T(<simpleType id=ID name=NCName any attributes>
    (annotation?, union)
  </simpleType> ) ->

  'concept' NCName 'subConceptOf' getId( T(union) )
    T(annotation, ID) ?

T(<simpleType id=ID any attributes>
    (annotation?, union)
  </simpleType>, X ) ->

  'concept' X 'subConceptOf' getId( T(union) )
    T(annotation, ID) ?

```

In XML Schema, a union type allows an element or attribute value to be one or more instances of one type drawn from the union of multiple atomic and list types. An atomic type is either a built-in type or a derived type defined using a restriction. The result of transformation of the union is a concept defined as the subconcept of the set of transformations of the individual types within the XML Schema union. Built-in types in XML Schema are transformed to themselves in WSMO. The transformation of the restricted and union types are defined in other parts of this section.

### A.3.5 Mapping for a List

A list may be defined to contain elements of a simpleType that has already been defined elsewhere in the XML Schema. Alternatively, a list may be defined to contain elements of a simpleType that is defined within the scope of the list itself. In the former case, the simpleType will have a name while in the latter case, there will be no associated name. The transformations for a list type are:

Listing A.7. Mapping for a List

```

T(<list id=ID itemType=QName any attributes>
    (annotation?)
  </list> ) ->

  'hasValues' (1 *) 'ofType' QName
    T(annotation, ID) ?

T(<list id=ID any attributes>
    (annotation?, (simpleType))
  </list> ) ->

  'hasValues' (1 *) 'ofType' getID( T (simpleType, ('list_simple_type_' + getUniqueId( ) ) ) )
    T(annotation, ID) ?

```

In the second case above, the transformation of the embedded simpleType (a WSMO concept) must appear somewhere in the resulting WSMO ontology. The `getId()` helper function returns the unique identifier of this concept which is then used to identify the type of instances that are permitted for the WSMO value list.

### A.3.6 Mapping for a Union

A union type enables an instance of an XML element, or attribute, to be one of more than one type drawn from a union of atomic and list types. The XML Schema definition of a union considers three possible types that can be used. The first is built-in simpleTypes which are directly supported in WSMO without any change. The second are simpleTypes that are defined elsewhere in the XML Schema and have a *name* attribute which can be used to identify them. The third are simpleTypes defined within the scope of the union itself. These simpleTypes do not have a defined *name* attribute. The transformation for the union type is:

Listing A.8. Mapping for a Union

```

T(<union id=ID memberTypes="t1 t2 ... tn" any attributes >
    (annotation?), (simpleType1 simpleType2 ... simpleTypem ) )
  </union> ) ->

```

```

'concept' 'union_' + getUniqueld( )
'subConceptOf' { ' t1, t2, ..., tk,
                 getID( T( tk+1 ) ), getID( T( tk+2 ) ), ... getID( T( tn ) ),
                 getID( T( simpleType1, 'item_in_union_' + getUniqueld( ) ) ),
                 getID( T( simpleType2, 'item_in_union_' + getUniqueld( ) ) ) ... ,
                 getID( T( simpleTypem, 'item_in_union_' + getUniqueld( ) ) ) }'
T(annotation, ID) ?

```

The transformation considers the three different simpleTypes that can be included in a union type. The types,  $t_1$  to  $t_k$ , represent built-in types. They are unchanged by the transformation as WSMO supports the built-in types of XML Schemas. The types,  $t_{k+1}$  to  $t_n$ , are simpleTypes that are defined elsewhere in the XML Schema. The transformation of these simpleTypes is defined in the section 'Mapping for a simpleType with a name Attribute' above. The types, simpleType<sub>1</sub> to simpleType<sub>m</sub>, represent types that are defined within the scope of the union definition itself.

### A.3.7 Mapping for an Annotation

Listing A.9. Mapping for an Annotation

```

T(<annotation id=ID any attributes >
  appinfo1 appinfo2 ... appinfon
  documentation1 documentation2 ... documentationn
</annotation, X >) ->

'nonFunctionalProperties'
  'hasIdentifier hasValue' X
  'annotationId hasValue' ID
  T( appinfo1 )
  T( appinfo2 )
  ...
  T( appinfon )
  T( documentation1 )
  T( documentation2 )
  ...
  T( documentationn )
'endNonFunctionalProperties'

```

### A.3.8 Mapping for Appinfo

Listing A.10. Mapping for Appinfo

```

T(<appinfo source=anyURL >
  xml_content
</appinfo >) ->

'appinfoSource hasValue' anyURL
'appinfo hasValue' xml_content

```

### A.3.9 Mapping for Documentation

Listing A.11. Mapping for Documentation

```

T(<documentation source=URI_Reference xml:lang=language >
  xml_content
</documentation >) ->

'documentationSource hasValue' URI_Reference
'dc#language hasValue' language
'documentation hasValue' xml_content

```

### A.3.10 Mapping for a Restriction

The XML Schema restriction element is used to define restrictions on a simpleType, simpleContent or complexContent XML Schema element. In this section we define the transformation for a restriction on a simpleType. All possible facets (e.g. minExclusive, maxExclusive etc.) of a simpleType that are possible are listed in the XML Schema fragment at the top of listing 17. The transformation for the facets are given below in listing 18.

The base attribute of the XML Schema element is required and defines the name of a built-in type, simpleType or complexType element defined in this schema or another schema.

Listing A.12. Mapping for a Restriction

```

T(<restriction id=ID base=QName any attributes >
  (annotation?, (simpleType?, (
    minExclusive? | minInclusive? | maxExclusive? | maxInclusive? | totalDigits? | fractionDigits? |
    length? | minLength? | maxLength? | enumeration? | whiteSpace? | pattern? )*)
  ))
</restriction>, Y ) ->

'axiom' Y'_constraint'
T( annotation, ID ) ?
'definedBy'
  '!- naf ('
    ?X 'memberOf QName
      'and' ?X 'memberOf getId(T( simpleType ) ) ?

    'and' T( minExclusive, ?X ) ?
    'and' T( minInclusive, ?X ) ?
    'and' T( maxExclusive, ?X ) ?
    'and' T( maxInclusive, ?X ) ?
    'and' T( totalDigits, ?X ) ?
    'and' T( fractionDigits, ?X ) ?
    'and' T( length, ?X ) ?
    'and' T( minLength, ?X ) ?
    'and' T( maxLength, ?X ) ?
    'and' T( enumeration, ?X ) ?
    'and' T( whiteSpace, ?X ) ?
    'and' T( pattern, ?X ) ?
  ')

```

### A.3.11 Mapping for the Facets of a simpleType Restriction

We show transformations for a subset of the facets. The transformations not shown here follow the same pattern as those shown and can be easily worked out.

Listing A.13. Mapping for Facets of a simpleType Restriction

```

T(<xs:minInclusive value="Y"/>, X) -> X '>=' Y
T(<xs:minExclusive value="Y"/>, X) -> X '>' Y
T(<xs:maxInclusive value="Y"/>, X) -> X '<=' Y
T(<xs:maxExclusive value="Y"/>, X) -> X '<' Y

```

## A.4 Mapping for complexType

A complexType element is an XML element that contains other elements and/or attributes. A complexType element can be defined by a particular grouping of XML elements using keywords including sequence and choice to define the required structure of these elements in conforming XML fragments. A complexType can also be defined as restrictions or extensions to an existing simpleType or complexType by using the simpleContent or complexContent elements.

The following sections define the mappings for complexType elements including the possible variations defined in [\[XMLSchema\]](#). The mappings are defined in the following order:

- Mapping for complexType
- Mapping for complexContent
- Mapping for simpleContent

- Mapping for restriction
- Mapping for extension
- Mapping for group
- Mapping for all
- Mapping for choice
- Mapping for sequence

#### **A.4.1 Mapping for a complexType**

Listing A.14. Mapping for complexType

**With 'name' attribute:****T(<complexType**

```

  id=ID ?
  name=NCName ?
  abstract=true|false ?
  mixed=true|false ?
  block=(#all | list of (extension | restriction)) ?
  final=(#all | list of (extension | restriction)) ?
  any attributes >
  (annotation?,
    (simpleContent | complexContent |
      ((group | all | choice | sequence)?,
        ((attribute|attributeGroup)*,anyAttribute?)
      )
    )
  )
)

```

**</complexType > ->****'concept' NCName****'nfp'**

```

  'xmlType' 'hasValue' 'complexType'
  'abstract' 'hasValue' ('true' | 'false') ?
  'mixed' 'hasValue' ('true' | 'false') ?
  'block' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
    { getID(T(restriction1)), getID(T(restrictionm)) } ) ?
  'final' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
    { getID(T(restriction1)), getID(T(restrictionm)) } ) ?

```

**'endnfp'**

```

  ('simpleContentAttribute' 'ofType' getID(T(simpleContent)) |
  'complexContentAttribute' 'ofType' getID(T(complexContent))) |

```

```

  ('groupAttribute' 'ofType' getID(T(group)) |
  'allAttribute' 'ofType' getID(T(all)) |
  'choiceAttribute' 'ofType' getID(T(choice)) |
  'sequenceAttribute' 'ofType' getID(T(sequence))) ?

```

```

  getQName(attribute1) 'ofType' getID( T(attribute1) ) ?

```

```

  ...

```

```

  getQName(attributen) 'ofType' getID( T(attributen) ) ?

```

```

  getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?

```

```

  ...

```

```

  getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?

```

```

  T(annotation, ID) ?

```

**Without 'name' attribute:****T(<complexType**

```

  id=ID ?
  ...
  >
  (annotation?,
    (restriction|extension))
</complexType >, X) ->

```

**'concept' X****'nfp'**

```

  'xmlType' 'hasValue' 'complexType'

```

```

  ...

```

**'endnfp'**

```

    ('simpleContentAttribute' ofType getID(T(simpleContent)) |
     'complexContentAttribute' ofType getID(T(complexContent)))
    ...
    getQName(attributeGroupm) ofType getID( T(attributeGroupm) ) ?
T(annotation, ID) ?

```

#### A.4.2 Mapping for a complexContent

Listing A.15. Mapping for complexContent

```

T(<complexContent
  id=ID
  mixed=true | false
  any attributes >
  (annotation?,
  (restriction | extension))
</complexContent >, X) ->

  X ofType getID( T( restriction|extension ) )

  'nfp'
  'xmlType' 'hasValue' 'complexContent' ?
  'mixed' 'hasValue' 'true' | 'false' ?
  'endnfp'

  T(annotation, ID) ?

```

A complexContent element will always be a child element of a complexType element. The complexContent element maps to an attribute of the concept retrieved from the mapping of the enclosed restriction or extension element. As in some of the mappings related to simpleTypes, an additional variable, X, is required by the transformation. This variable supplies the name of the attribute resulting from the mapping.

#### A.4.3 Mapping for a simpleContent

Listing A.16. Mapping for simpleContent

```

T(<simpleContent id=ID any attributes >
  (annotation?,
  (restriction|extension))
</simpleContent >, X) ->

  X ofType getID( T( restriction|extension ) )
  'nfp'
  'xmlType' 'hasValue' 'simpleContent' ?
  'endnfp'
  T(annotation, ID) ?

```

As with the complexContent element, the simpleContent element is always a child element of a complexType element, and maps to an attribute of the concept retrieved from the mapping of the mapping of the enclosed restriction or extension element.

#### A.4.4 Mapping for a Restriction on complexContent

Listing A.17. Mapping for restriction for complexContent

```

T(<restriction id=ID base=QName any attributes >
  (annotation?,
  (group|all|choice|sequence) ?,
  ((attribute|attributeGroup)*, anyAttribute?))
</restriction >, X) ->

  'concept' X
  'nfp'
    'xmlType' 'hasValue' 'restrictionForComplexContent'
    'baseType' 'hasValue' QName
  'endnfp'

  ( 'groupAttribute' 'ofType' getID(T(group) ) |
  'allAttribute' 'ofType' getID(T(all) ) |
  'choiceAttribute' 'ofType' getID(T(choice) ) |
  'sequenceAttribute' 'ofType' getID(T(sequence) ) ) ?

  getQName(attribute1) 'ofType' getID( T(attribute1) ) ?
  ...
  getQName(attributen) 'ofType' getID( T(attributen) ) ?
  getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?
  ...
  getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?
  T(annotation, ID) ?

```

#### A.4.5 Mapping for a Restriction on simpleContent

Listing A.18. Mapping for restriction for simpleContent

```

T(<restriction id=ID base=QName any attributes >
  (annotation?, (simpleType?, (
    minExclusive? | minInclusive? | maxExclusive? | maxInclusive? | totalDigits? |
    fractionDigits? | length? | minLength? | maxLength? | enumeration? | whiteSpace? |
    pattern? )*) ? ,
  ( (attribute | attributeGroup)*, anyAttribute? ) )
</restriction >, X ) ->

'concept' X
  'nfp'
    'xmlType' 'hasValue' 'restrictionForSimpleContent'
    'baseType' 'hasValue' QName
  'endnfp'

  (+++)
T(<restriction id=ID base=QName any attributes >
  (annotation?, (simpleType?, (
    minExclusive? | minInclusive? | maxExclusive? | maxInclusive? | totalDigits? |
    fractionDigits? | length? | minLength? | maxLength? | enumeration? |
    whiteSpace? | pattern? )*) )
  (+++)

  getQName(attribute1) 'ofType' getID( T(attribute1) ) ?
  ...
  getQName(attributen) 'ofType' getID( T(attributen) ) ?
  getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?
  ...
  getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?

T(annotation, ID) ?

```

As in the previous section, the mapping for the restriction element for complexContent results in the creation of a new concept. If present in the definition of the restriction, attributes and attributeGroups are mapped to attributes of the newly created concept. The section of the mapping delimited with '(+++)' is exactly the same as the mapping for a [restriction on a simpleType](#), which results in the creation of an axiom.

#### A.4.6 Mapping for an Extension

Listing A.19. Mapping for an extension

```

T(<extension id=ID base=QName any attributes >
  (annotation?,
  (group|all|choice|sequence) ?,
  ((attribute|attributeGroup)*, anyAttribute?))
</extension >, X) ->

  'concept' X
  'nfp'
    'xmlType' 'hasValue' 'extension'
    'baseType' 'hasValue' QName
  'endnfp'

  ( 'groupAttribute' 'ofType' getID(T(group)) |
  'allAttribute' 'ofType' getID(T(all)) |
  'choiceAttribute' 'ofType' getID(T(choice)) |
  'sequenceAttribute' 'ofType' getID(T(sequence)) ) ?

  getQName(attribute1) 'ofType' getID( T(attribute1) ) ?
  ...
  getQName(attributen) 'ofType' getID( T(attributen) ) ?
  getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?
  ...
  getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?
  T(annotation, ID) ?

```

The structure of the extension XML element is the same as the structure of the restriction-for-complexContent XML element. Consequently, the mappings for both are identical.

#### A.4.7 Mapping for a Group

The attributes 'name' and 'ref' are mutually exclusive. A mapping for each of the two cases is provided below.

**With a value for the 'name' attribute:**

Listing A.20. Mapping for a group

```

T(<group id=ID name=NCName
  maxOccurs=nonNegativeInteger | unbounded ?
  minOccurs=nonNegativeInteger ?
  any attributes >
  ( (annotation?,
  (all | choice | sequence) ) ?
</group > ) ->

  'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'group'
    'maxOccurs' 'hasValue' (nonNegativeInteger | 'unbounded') ?
    'minOccurs' 'hasValue' nonNegativeInteger ?
  'endnfp'

  ( 'allAttribute' 'ofType' getID(T(all)) |
  'choiceAttribute' 'ofType' getID(T(choice)) |
  'sequenceAttribute' 'ofType' getID(T(sequence)) ) ?

  T(annotation, ID) ?

```

**With a value for the 'ref' attribute:**

Listing A.21. Mapping for a group with a value for the 'ref' attribute

```

T(<group id=ID ref=QName >, X ) ->

'concept' X
  getLocalPart(QName) 'ofType' getID(T(retrieveElement( QName ))) ?
  'nfp'
    'xmlType' 'hasValue' 'group'
  'endnfp'

```

The 'name' attribute specifies a name for the group and is used only when the schema element is the parent of this group element. The 'ref' attribute refers to the name of another group. As mentioned already the 'name' and 'ref' attributes cannot both be present.

The 'xmlType' non functional property is used to declare the type of the source XML Schema element. This is to help maintain structural information in the mapping. (all, group, sequence, choice) from which the WSMO concept has been mapped.

#### A.4.8 Mapping for the All element

Listing A.22. Mapping for all

```

T(<all id=ID maxOccurs=1 minOccurs=0 | 1 any attributes >
  ( annotation?, element* )
  </all >, X ) ->

'concept' X
  'nfp'
    'xmlType' 'hasValue' 'all'
    'maxOccurs' 'hasValue' '1' ?
    'minOccurs' 'hasValue' '0' | '1' ?
  'endnfp'

  getQName(element1) 'ofType' getID( T(element1) ) ?
  ...
  getQName(elementn) 'ofType' getID( T(elementn) ) ?
  T(annotation, ID) ?

```

The 'compositorType' non functional property is used to declare the type of XML Schema compositor element (all, group, sequence, choice) from which the WSMO concept has been mapped.

#### A.4.9 Mapping for the Choice element

The choice element allows only one of the elements contained in the 'choice' declaration to be present within the containing element.

Listing A.23. Mapping for Choice

```

T(<choice id=ID maxOccurs=nonNegativeInteger|unbounded minOccurs=nonNegativeInteger any attribute.
  (annotation?, (element | group | choice | sequence | any) *)
</choice >, X ) ->

  'concept' X
    'nfp'
      'xmlType' 'hasValue' 'choice'
      'maxOccurs' 'hasValue' nonNegativeInteger | 'unbounded' ?
      'minOccurs' 'hasValue' nonNegativeInteger ?
    'endnfp'

    getQName(element1) 'ofType' getID( T(element1) ) ?
    ...
    getQName(elementn) 'ofType' getID( T(elementn) ) ?

    getQName(group1) 'ofType' getID( T(group1) ) ?
    ...
    getQName(groupm) 'ofType' getID( T(groupm) ) ?

    getQName(choice1) 'ofType' getID( T(choice1) ) ?
    ...
    getQName(choicep) 'ofType' getID( T(choicep) ) ?

    getQName(sequence1) 'ofType' getID( T(sequence1) ) ?
    ...
    getQName(sequenceq) 'ofType' getID( T(sequenceq) ) ?

    getQName(any1) 'ofType' getID( T(any1) ) ?
    ...
    getQName(anyr) 'ofType' getID( T(anyr) ) ?

    T(annotation, ID) ?

```

#### A.4.10 Mapping for the Sequence element

The 'sequence' element specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times. The mapping is almost identical to that for the 'choice' element in the previous section. The only difference is that the value of the 'compositorType' non-functional property is set to 'sequence'.

Listing A.24. Mapping for Sequence

```

T(<sequence id=ID maxOccurs=nonNegativeInteger|unbounded minOccurs=nonNegativeInteger any attrib
  (annotation?, (element | group | choice | sequence | any) *)
</sequence >, X ) ->

  'concept' X
    'nfp'
      'xmlType' 'hasValue' 'sequence'
      'maxOccurs' 'hasValue' nonNegativeInteger | 'unbounded' ?
      'minOccurs' 'hasValue' nonNegativeInteger ?
    'endnfp'

    T(annotation, ID) ?
    ...

```

## A.5 Mapping for Attribute and AttributeGroup

### A.5.1 Mapping for the AttributeGroup

The `attributeGroup` element is used to group a set of attribute declarations so that they can be incorporated as a group into complex type definitions. We map `attributeGroups` to a concept that can be used as a building block for other more complex concepts.

The 'name' and 'ref' attributes of the 'attributeGroup' element are mutually exclusive. The following two subsections define the mappings for each case.

#### Mapping for an `attributeGroup` element with the 'name' attribute:

Listing A.25. Mapping for `AttributeGroup`

```
T(<attributeGroup id=ID name=NCName ref=QName any attributes >
  (annotation?, ((attribute|attributeGroup)*, anyAttribute?) )
</attributeGroup >) ->

'concept' X
'nfpp'
  'xmlType' 'hasValue' 'attributeGroup'
'endnfpp'

getQName(attribute1) 'ofType' getID( T(attribute1) ) ?
...
getQName(attributen) 'ofType' getID( T(attributen) ) ?

getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?
...
getQName(attributeGroupm) 'ofType' getID( T(attributeGroupm) ) ?
T(annotation, ID) ?
```

We represent the possibility of multiple attribute values using the subscripts from 1 to n where  $n > 1$ . Similarly for `attributeGroups`, we use the subscripts from 1 to m where  $m > 1$ .

#### Mapping for an `attributeGroup` element with the 'ref' attribute:

Listing A.26. Mapping for an `attributeGroup` element with the 'ref' attribute

```
T(<attributeGroup id=ID ref=QName >, X ) ->

X ofType getID(T( retrieveElement( QName ) ))
```

### A.5.2 Mapping for an Attribute

The XML Schema attribute element is used to define a data type that can be used at the schema level or within a more complex type definition. For our purpose, we map the XML Schema attribute element to a WSMO concept.

Attributes in XML Schema are always defined as a `simpleTypes`. This can be reference to an existing definition of a `simpleType` element using the 'type' attribute or can be by definition of a `simpleType` within the definition of the attribute itself. The following two subsections provide the definitions of mappings based on an externally and internally defined `simpleType` respectively. In the first case, there are two further sub-mappings as the 'name' and 'ref' attributes are mutually exclusive

#### Mapping based on an externally defined `simpleType`:

Listing A.27. Mapping for attribute based on an externally defined `simpleType`

```
With 'name' attribute:
T(<attribute
  id=ID ?
  name=NCName
  type=QName
  default=string ?
  fixed=string ?
  form=qualified | unqualified ?
  use=optional | prohibited | required ?
  any attributes
>
(annotation?)
</attribute >) ->

'concept' NCName
'nfpp'
```

```

    'xmlType' hasValue 'attribute'
    'default' hasValue string ?
    'fixed' hasValue string ?
    'form' hasValue 'qualified' | 'unqualified' ?
    'use' hasValue 'optional' | 'prohibited' | 'required' ?
  'endnfp'

```

```

  getLocalPart(QName) ofType getID(T(retrieveElement( QName )))

```

```

T(annotation, ID) ?

```

**With 'ref' attribute:**

```

T(<attribute id=ID ref=QName any attributes >
  (annotation?)
</attribute >, X) ->

```

```

'concept' X
  getLocalPart(QName) ofType getID(T(retrieveElement( QName )))  T(annotation, ID) ?

```

**Mapping based on an internally defined simpleType:**

Listing A.28. Mapping for attribute based on an internally defined simpleType

```

T(<attribute
  id=ID ?
  name=NCName ?
  default=string ?
  fixed=string ?
  form=qualified | unqualified ?
  use=optional | prohibited | required ?
  any attributes
>
  (annotation?, (simpleType) ) )
</attribute >) ->

```

```

'concept' NCName
  'nfp'
    'xmlType' hasValue 'attribute'
    'default' hasValue string ?
    'fixed' hasValue string ?
    'form' hasValue 'qualified' | 'unqualified' ?
    'use' hasValue 'optional' | 'prohibited' | 'required' ?
  'endnfp'

```

```

  'attributeSimpleType' ofType getID(T(simpleType) )

```

```

T(annotation, ID) ?

```

## A.6 Mapping for Element

**Mapping based on an externally defined simpleType or complexType:**

Listing A.29. Mapping for element Element

**With 'name' attribute:**

```

T(<element
  id=ID ?
  name=NCName ?
  type=QName (for type) ?
  substitutionGroup=QName ?
  default=string ?
  fixed=string ?

```

```

    form=qualified | unqualified ?
    maxOccurs = nonNegativeInteger | unbounded ?
    minOccurs = nonNegativeInteger ?
    nillable = true | false ?
    abstract = true | false ?
    any attributes
  >
  (annotation?)
</element > ) ->

'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'element'
    'substitutionGroup' 'hasValue' QName ?
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'maxOccurs' 'hasValue' (nonNegativeInteger | 'unbounded') ?
    'minOccurs' 'hasValue' nonNegativeInteger ?
    'nillable' 'hasValue' ('true' | 'false') ?
    'abstract' 'hasValue' ('true' | 'false') ?
    'block' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
                                { getID(T(restriction1)), getID(T(restrictionm)) } ) ?
    'final' 'hasValue' ('#all' | { getID(T(extension1)), getID(T(extensionn)) } |
                                { getID(T(restriction1)), getID(T(restrictionm)) } ) ?
  'endnfp'

  getLocalPart(QName) 'ofType' getID(T(retrieveElement( QName (for type) ))) ?
T(annotation, ID) ?

```

**Without 'name' attribute:**

```

T(<element
  id=ID ?
  type=QName (for type) ?
  ...
>
  (annotation?)
</element >, X) ->

'concept' X
  'nfp'
    'xmlType' 'hasValue' 'element'
    ...
  'endnfp'

  getLocalPart(QName (for type)) 'ofType' getID(T(retrieveElement( QName (for type) ))) ?
T(annotation, ID) ?

```

**With 'ref' attribute:**

```

T(<element id=ID ref=QName any attributes >
  (annotation?)
</element >, X) ->

'concept' X
  getLocalPart(QName) 'ofType' getID(T(retrieveElement( QName ))) ?
  'nfp'
    'xmlType' 'hasValue' 'element'
  'endnfp'

T(annotation, ID) ?

```

## Mapping based on an internally defined simpleType or complexType:

Listing A.30. Mapping for element Element

## With 'name' attribute:

```

T(<element
  id=ID ?
  name=NCName ?
  substitutionGroup=QName ?
  default=string ?
  fixed=string ?
  form=qualified | unqualified ?
  maxOccurs = nonNegativeInteger | unbounded ?
  minOccurs = nonNegativeInteger ?
  nillable = true | false ?
  abstract = true | false ?
  any attributes
>
  (annotation?, ((simpleType | complexType)?, (unique | key | keyref )* ))
</element > ) ->

  'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'element'
    'substitutionGroup' 'hasValue' QName ?
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'maxOccurs' 'hasValue' (nonNegativeInteger | 'unbounded') ?
    'minOccurs' 'hasValue' nonNegativeInteger ?
    'nillable' 'hasValue' ('true' | 'false') ?
    'abstract' 'hasValue' ('true' | 'false') ?
  'endnfp'

  'simpleTypeAttribute' 'ofType' getID\(T\(simpleType\) \) |
  'complexTypeAttribute' 'ofType' getID\(T\(complexType\) \)

  T\(annotation, ID\) ?

```

## Without 'name' attribute:

```

T(<element
  id=ID ?
  ...
>
  (annotation?, ((simpleType | complexType)?, (unique | key | keyref )* ))
</element >, X ) ->

  'concept' X
  'nfp'
    'xmlType' 'hasValue' 'element'
    ...
  'endnfp'

  'simpleTypeAttribute' 'ofType' getID\(T\(simpleType\) \) |
  'complexTypeAttribute' 'ofType' getID\(T\(complexType\) \)
  T\(annotation, ID\) ?

```

## Note:

The unique, key and keyref XML Schema facilities for the 'element' element provide a mechanism for applying referential integrity constraints on the data in an XML document. We do not provide a mapping for this to WSMO as it is unlikely to come up in the normally simpler XML Schema descriptions used by WSDL documents. This should be tackled as future work only if it is required.

## Appendix B. Summary of Grounding Non-functional Properties

In this document, we use the *withGrounding* construct defined in [Roman et al. 2005] and we introduce two new non-functional properties, *endpointDescription* and *endpointAddress*, defined below.

*endpointDescription*, when used on a WSMO Web service, complements the abstract choreography grounding information with concrete binding and endpoint data. It should contain a URI that points to a description of the physical endpoint(s) that this service makes available. For instance, using a URI syntax described in [section 3.2.3](#), this property could point to a WSDL service.

*endpointAddress*, also used on a WSMO Web service, indicates only the address of an endpoint of the Web service. The protocol used to access this endpoint is defined by the default grounding described in [section 3.3](#). The value of this property should be a URI pointing to the physical endpoint of the service.

## Acknowledgement

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperanto; by Science Foundation Ireland under the DERI-Lion project; by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects RW<sup>2</sup> and TSC.

The editors would like to thank to all the [members of the WSMO working group](#) for their advice and input to this document.

## Change Log

Date	Author	Description
2005/09/16	Matt	Small corrections to document structure - included description of subsections at the start of each major document section.
2005/08/31	Matt	Revised section 2 based on review comments to correct problems and to help document flow. Added mappings for XMLSchema complexTypes, elements and attributes. Create new Appendix for all mappings.
2005/08/08	Jacek	added section 3.2.1 on correspondences between WSDL operations and WSMO transition rules
2005/7/22	Jacek	Updated to sync with newest choreography
2005/7/20	Matt, Adrian	Added transformation mappings for simpleType.
2005/6/25	Jacek	added appendix A defining the NFPs, added mention of triple spaces, mostly editorial changes
2005/6/11	Jacek	major revamp of grounding, especially including choreography
2005/6/03	Matt	Revised section 2 for consistency. Added walk-through example for Section 2.5 - Creating Mappings at the conceptual level. Revised figure 3 (prev. figure 1) to make more readable.
2005/6/01	Matt	Edited section 1 and added new subsection 1.1 to describe purpose of the deliverable.
2005/4/29	Jacek	Updated figure 1, clarified interaction of syntactic/semantic clients/services.
2005/4/23	Jacek	Removed section 2.3 with the direct mapping approach, that work has been discontinued in favor of the ontology-level mapping approach.
2005/4/23	Jacek	Finished section 3 according to WSMO (choreography) status quo.
2005/4/22	Matthew	Extended section 2.4, adding new sections for related work, translation examples and discussion points.
2005/3/11	Adrian, Matthew	Revised section 2.4, including references to previous work in mapping between XML schema and ontologies.
2005/3/11	Titi	split section 3.2, added pros in 3.2.2
2005/3/11	Jacek	enhanced WSDL description
2005/2/11	Titi	added example for wsdl types using wsmo ontologies
2005/2/9	Jacek	Adding Matt's approach as sec 2.4
2005/1/17	Jacek, Titi	Initial version



\$Date: 2005/09/05 14:42:23 \$

[webmaster](#)