DERI INNSBRUCK

XML parser extension for WSMO4J

Bachelor thesis

Hannes Innerhofer

csad5132@uibk.ac.at
matrnr. 0115721

14. October, 2007

Supervisor:
Reto Krummenacher
Digital Enterprise Research Institute
University of Innsbruck

Abstract

The Web Service Modeling Ontology (WSMO) defines a model which allows the semantical description of Web services. WSML is the description language for this model and introduces syntaxes and semantics to describe it. This thesis will introduce an API that allows the parsing and serialization of WSMO objects. It is based on the WSMO4J API which allows the creation and manipulation of WSMO objects.

The XML parser extension for WSMO4J provides the implementations for parsing XML syntax into WSMO4J objects and for serializing WSMO4J objects to XML syntax and extends WSMO4J in the means that WSMO4J itself did not support this serialization and parsing to and from XML syntax. The extension is intended to be easily adaptable and extensible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays the integration of different applications and services into one workflow takes a more and more important role. Businesses aim at automizing their workflows and need to integrate different applications on different platforms into one single process. The services used in that workflow do not have to be provided necessarily by the company itself, but may also be provided by third parties. However the detection and integration of such applications is always aligned with a high effort.

Semantic Web services aim at providing not only a Web service but also to give a semantic, machine-readable description of this service allowing third party software to detect

- What the service is good for, what results are achieved
- How the service may be used in a service composition in order to achieve the desired results
- How the service is invoked properly

Therefor the semantic Web is intended to be an extension of the current Web. In order to allow the processing of of requests based of its semantic meaning and not on its syntax, information must be enriched with semantic markupwhich is created by using ontologies. These ontologies provide a formal specified vocabulary to describe such semantic Web services. The Web Service Modeling Ontolgoy WSMO is such an ontology and since april 2005 a W3C submission. It contains the subproject WSML which defines a formal language for WSMO.

WSMO4J is the reference implementation of the WSMO-API which defines the programming interfaces of WSMO. It allows the creation of the objects defined by WSMO, their processing and parsing/serialization into WSML human readable syntax. However it does not support the serialization into

and the parsing from XML syntax. This thesis introduces the XML parser extension for WSMO4J which aims at adding that feature to WSMO4J.

The following sections will explain in detail the goal of the thesis and discuss the methodology to achieve the desired results. After an introduction of the used technologies - WSMO, the WSMO4J API and various XML parser APIs - the structure of the WSML parser API itself is explained in detail and some examples will illustrate how the extension can be used.

1.1 Goal

The XML parser extension for WSMO4J is designed to extend the WSMO4J API [1] in the term that it adds the feature of parsing the WSMO [3] objects from and serializing them into XML syntax.

The XML parser extension for WSMO4J consists of two parts: the parser and the serializer. The interfaces that both - the serializer and the parser - implement, are defined in the WSMO4J API. The implementation uses the Xerces-J API [6] for parsing XML documents and constructing the serializeable XML document tree, to be more precise the DOM parser [7].

The WSMO4J API provides the methods for the parsing and serialization of the WSML [2] human readable syntax as well as the creation and manipulation of the corresponding WSMO objects for a further processing of the WSMO models. However it did not support the parsing of these WSMO objects from XML syntax and the serialization of them into XML.

The motivation for adding that feature is the fact that - at the time of work - XML is a widely accepted language for interchanging data between different applications and platforms. To maintain great interoperability also for the WSMO4J API, allowing the use of it in collaboration with other products it is essential to add that feature of serializing/parsing the WSMO object representation into XML.[5]

XML is the most important language for data representation in the field of Web services. Remote Procedure Calls (RPC), SOAP[16], WSDL[17] are protocols taking an important place in the Web services area and they are based on XML. RDF[15] is as well an important language in this area but unlike the other protocols it is designed do give a semantic meaning to the Web. However it is also based on XML and therefor as being more or less a successor of RDF, the WSMO models should also be representable in XML syntax.

1.2 Methodology

After describing the goal of the project in the previous chapter, this section introduces the means that ensured a successful realization of that goal. Obviously the first step was the familiarizing with the subject. What is WSML, what is WSMO, for what are they used, what are their goals? In fact, WSMO (Web Service Modeling Ontology) defines a model for the semantic description of Web services. For example it allows the modeling of goals of a Web service from the user view, its interfaces, used concepts or relations, just to mention some of its capabilities. A detailed description that has also been a source for the research on this project is the final draft for WSMO. WSML is the description language for that model. It defines the syntax for describing the concepts of WSMO in a formal language. The final draft for WSML [2] contains the WSML syntax description but also rules how to transform its human-readable syntax into XML, RDF and OWL-DL.

After getting an overview over the main technologies and a deeper knowledge of the WSMO-Language WSML, the next step was the incorporation into WSMO4J. As explained previously, WSMO4J presents an API that allows the creation and modification of WSMO objects (Goals, WebServices, Ontologies and Mediators) and their serialization into WSML human readable syntax as well as the parsing of such objects out of the human readable syntax.

The extension should be implemented in a flexible and extensible way to guarantee an easy maintainment of the code. Due to the fact that the WSMO4J API was still under development this was imperative as changes to the WSMO4J API should not have had great impact to the extension. The documentation of WSMO4J consisted of a JAVADOC documentation which explained the internal technical details of the API and a paper explaining its general structure and behavior.

Before starting with the implementation, the system was modeled as shown in the class diagrams in Chapter 3 and Appendix A. It is abutted to the WSMO4J class model as it should be a part of that API.

After modeling the system, the implementation could begin. Nearly each object of the WSMO ontology is serialized and parsed by an own parser/serializer class. Each one of this parser/serializer classes signs responsible for the correct parsing/serialization of one type of WSMO object and for delegating the parsing/serialization of all its child objects to the corresponding parser/serializer class. A more detailed description of the structure will be given in Chapter 3.

Chapter 2

Technologies

As mentioned in the previous chapter, several technologies have been used to complete the project, namely WSMO, the Web Service Modeling Ontology, WSML, its description language and WSMO4J, an API for creating WSMO objects. These technologies will be introduced in the following sections. Finally various XML parser APIs will be discussed.

2.1 WSMO

There has been done a lot of research for making Web content more processable for machines. Semantic markup should make it possible to automate the discovery, composition and invocation of Web services, reducing the necessity of human intervention to a minimum.

The Web Service Modeling Ontology aims at describing Web services in a semantic, machine-readable way. This technology allows programs to retrieve information about Web services and be able to process it. As a consequence this makes it possible to dynamically search for Web services that meet certain criteria like the goal of the service, the capability or the interfaces it provides.

WSMO provides ontological specifications for the core elements of Semantic Web services. In fact Semantic Web services aim at an integrated technology for the next generation of the Web by combining Semantic Web technologies and Web services, thereby turning the Internet from an information repository for human consumption into a world-wide system for distributed Web computing. Therefore appropriate frameworks for Semantic Web services need to integrate the basic Web design principles, those defined for the Semantic Web, as well as design principles for distributed, service-oriented computing of the Web. [3]

There already exist protocols to describe Web services but none of them includes the semantic description of the functionality of a Web service. They are represented only syntactically and therefore do not depict the meaning of the information to be interchanged. With a semantic description of the Web services it will be possible to discover, composite, contract and execute Web services.

The Semantic Web and Web services are envisioned as the enabling technologies for the next generation of Web applications. The former aims at making Web content readable for machines, whereof ontologies have been identified as the key technical building block. The objective of Web services is to enable distributed computation over the Internet by automated and dynamic discovery, composition, and execution of services, thus providing a new technology for Web-based system engineering.

The ability to describe Web services in a machine-understandable way is expected to have a big influence in areas of e-Commerce and Enterprise Application Integration as it should make it possible to enable dynamic cooperation between different systems.

2.1.1 The WSMO elements

The top level elements of WSMO are the ontologies, Web services, goals and mediators. In the following the most important elements of the WSMO meta-model will be summarized.

Ontologies provide the terminology used by other WSMO elements. In the ontologies concepts, relations, instances and relation instances are defined. These elements defined in the ontology are then used by Web services, goals and mediators.

Concepts represent classes of objects of a real or abstract world. They provide attributes which the instances of the respective concept can have. The value given to the attribute by the instance is restricted also in the concept by defining logical constraints. A concept can have an arbitrary number of super concepts. Constraints and attributes that are defined in the superconcept are inherited to the concept.

Instances are objects that hold data as defined in their concepts. For example a concept "car" defines a set of objects. The instances are the objects themselves like a specific car with a certain color, construction year and type. Instances specified in an ontology are those which are shared together as part of the ontology. However most of the instances may reside outside the ontology definition in data stores (files, databases, et. etc.).

Relations tackle at the modeling of dependencies between different concepts (respectively instances of these concepts). A relation can have several parameters, each of them having a range restriction in form of a concept. It may also have an arbitrary number of super relations from which it inherits the signature and the parameter constraints.

Relation instances represent instances of relations. A relation instance is associated with a single relation and is part of an ontology.

Non-functional properties may occur in the definition of all WSMO elements. As the name already denotes they take care of describing the non-functional parts of an element like the creator, contributor or creation date.

Imported ontologies: Any of the four top level elements in the WSMO definition can import ontologies to define the terminology used by the element. By giving this possibility, ontologies can be designed in a modular way.

Used mediators: The import of ontologies might lead to mismatches between the different ontologies. Mediators are used for resolving these mismatches by making an alignment of the imported ontologies. Just like the imported ontologies also the used mediators may be used in any of the four top level elements.

Web services represent entities providing access to services that - in turn - provide some value in a domain. Web services contain capabilities and interfaces and describe the internal working of the service. The goal that the Web service should achieve is defined in the top-level element goal.

Capabilities describe the functionality of a Web service. They define itself by describing the state of the world before the Web service execution and after it. The capability descriptions are first of all used to determine if a Web service meets required needs. A capability defines an assumption, precondition, postcondition and an effect.

Preconditions define what information must be provided and what its state has to be like before the Web service execution. They specify what information the Web service needs to return as a result.

Assumptions define the state of the world before the Web service execution. While preconditions are always checked by a Web service, assumptions might also be ignored and not taken into account.

Postconditions describe the information data after the successful execution of the Web service.

Effects describe the state of the world after the successful Web service execution.

Shared variables represent the variables that are shared amongst preconditions, postconditions, assumptions and effects.

Interfaces provide a description of how the Web service is correctly invoked and what other functionality is required from other Web services. Interfaces are first of all meant for the description of the Web service, making it possible for software agents to discover the behavior of the Web service. The choreography and the orchestration are both defined in the interfaces of a Web service.

Choreography: It defines the protocol to interact with the Web service, to access and monitor it.

Orchestration: Web services will mostly not achieve a goal on their own but might use other Web services of other companies to meet the requirements for achieving the defined goal. The orchestration describes what other Web services are used by this one and how they are used.

Goals describe the goal of a service. They model the view of the user in the Web service usage process.

Mediators handle interoperability problems between different WSMO elements. They have to resolve incompatibilities on the data and protocol level and on the level of combining Web services (process level).

2.2 WSML

The Web Service Modeling Ontology WSMO proposes a conceptual model for the description of ontologies, Semantic Web services, goals, and mediators, providing the conceptual grounding for ontology and Web service descriptions. The conceptual model of WSMO is taken as a starting point for the specification of a family of Web service description and ontology specification languages.

The Web Service Modeling Language (WSML) aims at providing means to formally describe all the elements defined in WSMO. The different vari-

ants of WSML correspond with different levels of logical expressiveness and the use of different language paradigms. All WSML variants are specified in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model.

Ontologies and Semantic Web services need formal languages for their specification in order to enable automated processing. WSML aims at providing this set of formal languages in order to enable a formal description of the WSMO models.

2.2.1 The WSML syntax

The different elements of a WSMO model as introduced in the previous section can all be described with WSML. First of all the WSML variant is defined. The variant might be of the type WSML-Full, WSML-Core, WSML-DL, WSML-Flight or WSML-Rule or even unspecified. The variant can be used to set limitations to the complexity of models. While in WSML-Full all language constructs may be used, the other variants give limitation to the used language.

The variant definition is followed by the definition of any of the four top-level elements of WSMO: The Web services, the goals, ontologies and mediators. All WSMO elements are described in WSML syntax by the keyword that corresponds to the element, followed by an identifier.

Identifiers in WSML can be IRIs, data values or anonymous ids. WSML has some built-in data types. The most common are strings, integers and decimals. These can be used to construct more complex data types. Data type identifiers in a WSML document are either concept identifiers or data type wrappers. Table 2.1 lists all the built-in WSML data types.

Table 2.1: The built-in WSML data types

_string	_float	_time	_gday
_integer	_double	_date	_gmonth
_decimal	_boolean	_gyearmonth	_hexbinary
_iri	_duration	_gyear	_base64binary
_sqname	_dateTime	_gmonthday	

It is possible to add to all WSMO elements non-functional properties. These properties give information about the element, like a title, description, language, et etc. However they do not have any impact on the service func-

tionality. They only give additional information for the user of the Web service. These properties are specified by giving the name of the attribute (the properties of Dublin Core are recommended) followed by the keyword `hasValue` and the value of the attribute.

Further on a Web service - as well as the other three top level elements of WSMO - may import ontologies and use mediators (there is one exception: the ooMediator). This allows it to share WSMO definitions across Web services, enabling a modular programming approach. Further on a Web service can define a capability and interface. As well as all other WSMO elements, a capability may have a block of non-functional properties. Additionally it may define imported ontologies and used mediators, followed by an optional sharedVariables block and an arbitrary number of preconditions, postconditions, assumptions and effects. Interfaces may contain non-functional properties, imported ontologies and used mediators. Additionally a choreography and an orchestration may be specified.

The next top-level element, which is very similar to a Web service in WSMO is the goal. With the exception of the different keyword the goal in WSML is described with the same elements than the Web service.

The mediator is the next top-level element discussed. There exist four kinds of mediators to each of them mediating between different WSMO top-level elements. The ooMediator is used for importing ontologies and resolving heterogeneity. OoMediators are the single top-level elements that may not define a used service. In addition to imported ontologies and non-functional properties, an ooMediator may also define an arbitrary number of sources (which must be either ontologies or other ooMediators), a target (can be any top-level element) and a used service. While the type of the target and sources varies between the different kinds of mediators, the used service is always one of goal, Web service and ggMediator. The table below (table 2.2) shows the types of sources and targets of the four kinds of mediators.

Remember that all mediators may only specify one target and one source, only the ooMediator may specify multiple sources.

Finally the elements an ontology element may contain will be introduced. A concept definition starts with the concept keyword, optionally followed by the identifier of the concept. With the keyword `subConceptOf` its superconcepts may be specified - if there are any - followed by the non-functional properties. Finally an arbitrary number of concept attributes may be specified, defining also the type a corresponding attribute value in an instance must be of.

Table 2.2: Type of sources and targets of the four mediators

Mediator	Source	Target
ooMediator	Ontology ooMediator	Top-Level Element
wwMediator	WebService	WebService wwMediator
ggMediator	Goal ggMediator	Goal ggMediator
wgMediator	WebService wgMediator	Goal ggMediator

The concrete realizations of such concepts are called instances. Instances are defined giving the instance keyword, followed optionally by an identifier. The concept to which the instance belongs is expressed using the `memberOf` keyword, followed by the identifier of the respective concept. After it the values for the attributes defined in the concept are specified.

The last two elements which will be discussed here are the relation and the relationinstance. A relation has an identifier, followed optional by the arity of the relation and an arbitrary number of parameters (which must correspond to the defined arity). After them the superrelation - if there is any - may be specified, followed by a non-functional properties section. Relationinstances are the same for relations as the instances are for concepts. The `relationInstance` keyword is followed by an identifier, an identifier of the corresponding relation and - in brackets - the values for the parameter types as defined in the relation.

This element concludes the excursion into the WSML syntax. It was meant to give a very basic understanding of the WSML language as this will be needed to get familiar with WSMO4J and the XML parser extension for WSMO4J. It was not meant to explain the exact syntax or every element of WSML. This is covered by the "The Web Service Modeling Language WSML" final version [2]. It gives a detailed description of the syntax and the different WSML variants, the whole language production and grammar of the language and the exchange syntaxes to transform WSML from human-readable syntax into RDF and XML.

2.3 WSMO4J

WSMO4J is mainly a Java API for the creation of Semantic Web service applications compliant with the Web Service Modeling Ontology (WSMO).

Furthermore it also allows the serialization of these WSMO objects and their parsing. However the API provides only the serialization into the WSML human readable syntax and the parsing out of that syntax. WSMO4J is still under development, at the time the XML parser extension for WSMO4J was written it was at version 4.0.

WSMO4J consists of two parts. One is the WSMO-API which defines the interfaces for the WSMO objects allowing third parties to write their own implementation of WSMO. The other one is the reference implementation. WSMO4J is currently released with version 0.4, the documentation and source code can be found at http://wsmo4j.sourceforge.net/.

As being object-oriented WSMO4J contains classes for the main WSMO concepts:

- WebService, Goal, Capability, Choreography, Interface, Orchestration
- ggMediator, wwMediator, wgMediator, ooMediator
- Namespace, IRI
- Ontology, Attribute, Concept, Instance, Relation, RelationInstance, Parameter, Variable, DataValue
- Axiom, LogicalExpression

Nearly all classes extend the `Entity` class. This class contains the identifier of the WMSO object and the non functional properties (NFP). The `TopEntity` is one level below the `Entity` class - which it extends - and contains the used mediators and imported ontologies. The classes that extend the `TopEntity` class are `Ontology`, `ServiceDescription`, `Mediator`, `Capability` and `Interfaces`.

All the classes in the WSMO4J API cannot or at least should not be created by instantiating them directly but can be created by using the `Factory` class. The implementation takes use of the factory design pattern. A factory is the location in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base object. Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. The provided methods we are interested in are:

The `LogicalExpressionFactory` - as its name already denotes - allows the creation of logical expressions. Logical expressions take an important part in WSMO. For example they specify the capability of a Web service or the axioms of an ontology element.

```
static LogicalExpressionFactory
       createLogicalExpressionFactory( Map properties);

static WsmoFactory createWsmoFactory( Map properties );

static Parser createParser( Map properties );

static Serializer createSerializer( Map properties );
```

The `WsmoFactory` provides methods for the creation of all WSMO objects like Web services, goals, interfaces or axioms. The `createWsmoFactory` method expects in fact a map as parameter, nevertheless it can be also `null` which instructs the method to return the `WsmoFactory` of the reference implementation.

The `createParser` method creates a parser out of a class that implements the WSMO4J `Parser` interface. It expects a map containing at least one entry with the `Factory.PROVIDER_CLASS` property as key and the name of the class to be instantiated as its value.

The `creatSerializer` method creates a serializer out of a class that implements the WSMO4J `Serializer` interface. It expects a map containing at least one entry with the `Factory.PROVIDER_CLASS` property as key and the name of the class to be instantiated as its value.

For more detailed information on WSMO4J the project website [1] may be consulted.

2.4 XML parser API

There are several reasons why it is desired to transform WSMO ontologies from human readable into XML syntax. One and the probably most important fact is that XML is widely spread. Being a widely accepted W3C recommendation, there are offered many tools to handle XML. They facilitate the creation and processing of XML documents. Many of them are freely available.

Already a large number of organizations are also using XML based Web services to expose their back-end data systems and leverage their existing IT infrastructure by effectively unlocking data that has been tied into proprietary applications. Applications that use XML are for example SOAP, WSDL or XML-RPC. Therefore it is desirable to transform WSML human

readable syntax into XML.

For the parsing of XML syntax there already exist a lot of APIs that can read data from an arbitrary input source and provide the parsed data in an easy to handle manner. There exist two main models for representing the parsed data, the SAX- and the DOM model.

The SAX [4] parser allows the specification of handlers that will be called whenever the parser reaches a certain node, for example handlers for attributes, starting and ending elements or document type definitions (DTD). The SAX parser then walks through the data to parse and calls the respective handler for each element. However this makes it quite complex to access different elements of an XML document besides this walk-through.

The DOM [7] parser compiles an XML document into an internal tree structure, then allows an application to navigate that tree. It is memory-consumptive (the bigger the XML data to represent, the bigger the need for resources), but the data representation on hand of a DOM document tree is very comfortable. Another advantage of DOM is that the document may first be validated against a schema and is then parsed, so if the document contains a validation error the parsing does not take place. On the other hand a SAX parser validates the document during the parsing, so it might happen that nearly the whole document is parsed but at the end of it a parsing error occurs and the parsing was useless.

For our purposes the DOM parser is required. All the data can be passed at once between the different parser classes, for example the `WebServiceParser` receives the `<webService>` node with all its child nodes and can pass from the child nodes the `<capability>` node with all its children to the `CapabilityParser` making it much easier to handle the parsed data. Furthermore the DOM parser caches the elements and serializes them only when calling the respective method, allowing the manipulation of the elements. Using the SAX serializer the elements are not cached but written to the stream immediately making it impossible to manipulate the elements in an easy manner.

Another requirement is the possibility to validate the XML syntax. There exists an XML schema for WSML in XML syntax and the data to be parsed should be validated against it. There exist many parser APIs, some of the most common are listed here:

Xerces-J

This is a very complete, validating parser that has a good conformance

to XML 1.0 and namespaces in XML specifications. It fully supports the SAX 2.0 and DOM level 2 [12] APIs, though there are very few bugs in the DOM support. The latest versions feature experimental support for parts of the DOM level 3 [13] working drafts. Xerces-J is highly configurable and suitable for almost any parsing needs. Xerces-J is also notable for being the only current parser to support the W3C XML schema language, though that support is not yet 100% complete or bug-free. [6]

Oracle

The Oracle parser implements DOM level 1, and SAX 1.0 and 2.0; it has a partial implementation of DOM level 2 and includes APIs for XSLT. It supports schemas through the oracle.xml.parser.schema package [8].

Piccolo

Piccolo is a small, very fast XML parser for Java. It implements the SAX 1, SAX 2.0.1 and JAXP 1.1 (SAX parsing only) interfaces as a non-validating parser and attempts to detect all XML well-formedness errors. It reads the external DTD subset in order to apply default attribute values and resolve external entity references. Piccolo supports the SAX API exclusively. It does not have a DOM implementation nor does it support XML schemas. [9]

Crimson

Crimson, previously known as Java Project X, is the parser Sun bundles with the JDK 1.4. Crimson supports more or less the same APIs and specifications Xerces does - SAX 2.0, DOM level 2, JAXP, XML 1.0, namespaces in XML, etc. - with the notable exception of schemas. [10]

Ælfred

The GNU Classpath Extensions Project's Ælfred is actually two parsers, gnu.xml.aelfred2.SAXDriver and gnu.xml.aelfred2.XmlReader. SAXDriver aims for a small footprint rather than a large feature set. It supports XML 1.0 and namespaces in XML. However it does not validate and it does not make all the well-formedness checks it should make. It supports SAX but not DOM. Its small size makes it particularly well-suited for applets. For less resource constrained environments Ælfred provides XmlReader, a validating parser that supports both SAX and DOM. [11]

Table 2.3: Capabilities of the different XML parser APIs

	Sax1	Sax2	Dom1	Dom2	Dom3	Schema
Xerces-J	x	-	x	x	x	x
Oracle	x	x	x	x	-	x
Piccolo	x	x	-	-	-	-
Crimson	x	x	x	x	-	-
Ælfred	x	-	-	-	-	-

For the needs of the XML parser extension for WSMO4J the optimal tool is Xerces-J. It supports the checking for well-formedness, DOM parsing and the validation against XML schemas. Piccolo does not support DOM parsing, Crimson and Ælfred do not support validation against schemas. Only Oracle would also meet the requirements. However the problem is that Oracle is not freely available. You may only get a developers license for developing a prototype. But for the distribution of your application a license has to be acquired.

Chapter 3

Realization

WSMO4J comes along with a parser for the parsing from and the serialization into human-readable WSML syntax. However it did not support serialization into other formats. This should be made possible by this API which aims to add this feature to the WSMO4J API.

The WSMO-API provides a parser interface allowing the integration of third party parsers. This parser interface provides four methods as shown in table 3.1.

Table 3.1: WSMO-API parser interface methods

`org.wsmo.common.TopEntity[] parse(java.io.Reader src)` Parses the input data consisting of Ontology, Goal, Mediator or WebService definitions.
`org.wsmo.common.TopEntity[] parse(java.io.Reader src, java.util.Map options)` Parses the input data consisting of Ontology, Goal, Mediator or WebService definitions.
`org.wsmo.common.TopEntity[] parse(java.lang.StringBuffer src)` Parses the input data consisting of Ontology, Goal, Mediator or WebService definitions.
`org.wsmo.common.TopEntity[] parse(java.lang.StringBuffer src, java.util.Map options)` Parses the input data consisting of Ontology, Goal, Mediator or WebService definitions.

The Serializer interface provided by WSMO4J has a similar structure as shown in table 3.2. A parser/serializer implementing the methods as shown in the tables 3.1 and 3.2 can be instantiated using the WSMO4J ParserFactory.

Table 3.2: WSMO-API serializer interface methods

`void serialize(org.wsmo.common.TopEntity[] item, java.lang.StringBuffer target)` Parses the input data consisting of Ontology, Goal, Mediator or Web-Service definitions.
`void serialize(org.wsmo.common.TopEntity[] item, java.lang.StringBuffer target, java.util.Map options)` Serializes Ontology, Goal, Mediator or Webservice.
`void serialize(org.wsmo.common.TopEntity[] item, java.io.Writer target)` Serializes Ontology, Goal, Mediator or Webservice.
`void serialize(org.wsmo.common.TopEntity[] item, java.io.Writer target, java.util.Map options)` Serializes Ontology, Goal, Mediator or Webservice.

As in WSMO4J there exists an own interface for the parser and the serializer, also the XML parser extension for WSMO4J is divided into a parser (org\deri\wsmo4j\io\parser\xml) and a serializer package (org\deri\wsmo4j\io\serializer\xml). The two packages will we discussed in detail in the following sections.

3.1 The serializer

The main class for performing the serialization is the XMLSerializer class. As already mentioned the XMLSerializer class implements the WSMO4J Serializer interface. The reason why it has to implement that interface is the following: WSMO4J object creation may only be realized by using the WSMO4J factory classes. These factory classes have been introduced in section 2.3. One of these classes, namely the class Factory provides a method named `createSerializer`. It takes as parameter a java.util.Map which contains the name of the class which should be instanciated. This means the class which is instanciated and used as serializer is unknown at compile time. However that class has to implement the WSMO4J Serializer interface, otherwise the instantiation will cause a runtime exception.

An overview of the interaction between WSMO4J and the XML parser extension for WSMO4J is provided by the class diagram in figure 3.1. If a programmer wants to make use of the serializer of the XML parser extension for WSMO4J, an instance of the XMLSerializer using the Factory class

of WSMO4J has to be created. The instance is returned to the application and may be used to perform the required serialization. All other classes in the serializer package are not of interest for the programmer using the API.

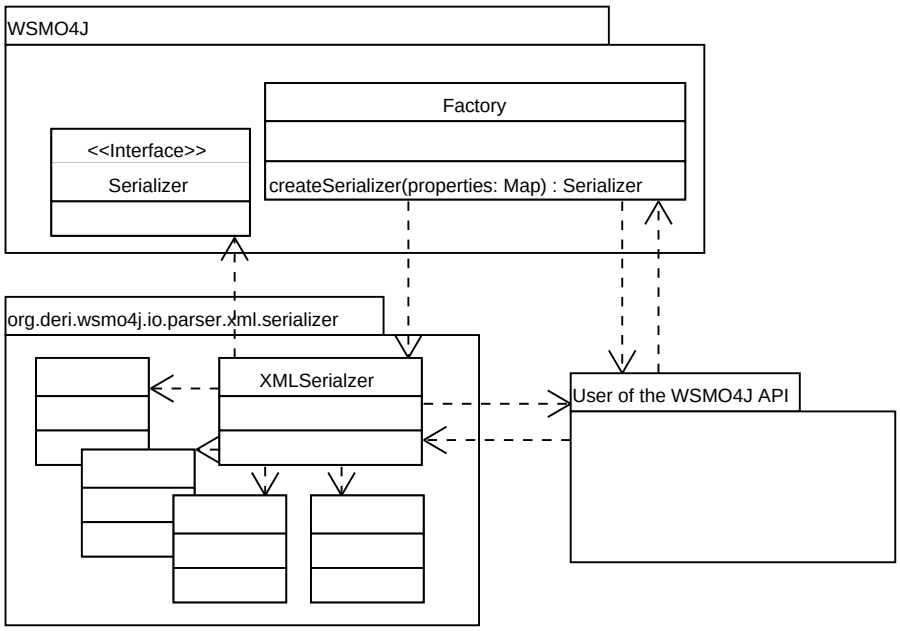

Figure 3.1: Interaction WSMO4J - XMLSerializer - WSMO4J user

After getting an overview of how the XML parser extension for WSMO4J interacts with WSMO4J now some words to the design of the implementation. Nearly all WSMO elements have an equivalent object representation in WSMO4J. This means in WSMO4J there exists for example a class Ontology representing the WSMO ontology element, a class WebService representing the WSMO webService element and so on. Now for nearly each class in WSMO4J the XML parser extension for WSMO4J serializer package contains a corresponding serializer class. For example a WSMO4J Ontology gets serialized by the OntologySerializer class, a WSMO4J WebService gets serialized by the WebServiceSerializer class. It can be said that in most cases one class of the serializer package takes over serializing one WSMO4J type of object.

The classes of the serializer package all have a similar constructor: it takes as first parameter the object to be serialized and as a second parameter a java.util.Map. The map is used to store objects or values which have to be passed along the different serializers. For example every serializer class needs a w3c.dom.Document object, where to add the serialized data to. This object is stored in the java.util.Map parameter. The other object

currently stored in the map is a java.util.HashTable wherein the namespace prefix binding is stored. This is needed for the IdSerializer when serializing ids to split them into namespace prefix and local identifier and replace the namespace prefix with its abbreviation. In the hash table the namespace is stored as key, the namespace abbreviation as value. The use of this map makes the implementation more flexible. If in further versions the need to pass other objects along the serializer classes arises, the constructors do not have to be adapted but the objects can simply be inserted into the map and taken from it where they are needed.

It has already been mentioned that the main class for the serialization is the XMLSerializer which implements the WSMO4J Serializer interface. The XMLSerializer provides four public methods, namely the ones of the interface it implements. Internally all of these four methods make use of the WsmlSerializer class. This class creates the XML `<wsml>` element and its attributes conforming to the top-level elements it receives in the constructor. Also it takes over serializing these top-level elements by detecting their type and passing them to one of WebService-, Goal-, Ontology- or MediatorSerializer as shown in figure 3.2.

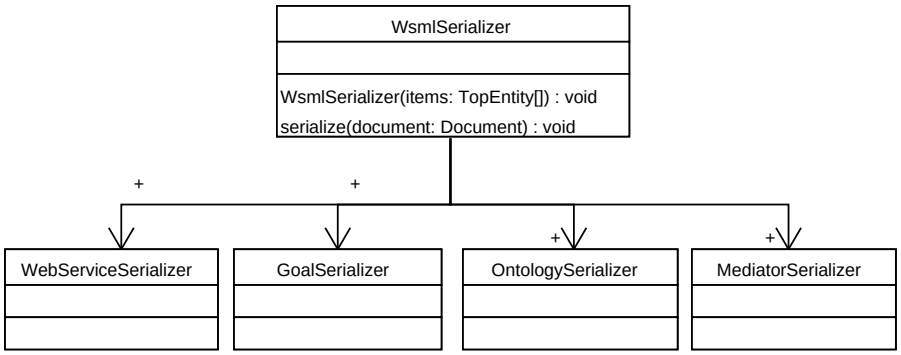

Figure 3.2: The WsmlSerializer class

The WsmlSerializer takes in its constructor an array of WSMO4J TopEntity objects. Besides the classes Ontology, WebService, Goal and Mediator, also the two classes Capability and Interface are of the type TopEntity by extending the TopEntity class. However these two classes may not me passed to and thus are not serialized by the WsmlSerializer.

As being the first class that makes any serialization in the serialization process, the WsmlSerializer takes in its serialize() method an empty org.w3c.dom.Document. This Document element is passed to every serializer class, each one of them fills in the serialized XML DOM nodes it is

responsible for. The WsmlSerializer for example is only responsible for setting the WSML variant and the prefix namespace binding. For serializing the top-level elements it calls the respective serializer.

3.1.1 IdSerializer

The reason why there exists an own class for serializing the ids is the possibility to define namespaces and to serialize them. The IdSerializer receives as parameter the id and a map containing the namespaces. It checks if the given id has a namespace prefix defined in the map and - if so - replaces it with the abbreviation for the namespace. If the id is of the type AnonymousId then it is not serialized as anonymous ids in WSMO4J are only created and used internally.

3.1.2 EntitySerializer

The Entity class is the super class of nearly all objects in WSMO4J. To be more precise it is the super class of those objects which can contain non functional properties and which have an identifier.

The EntitySerializer takes in his constructor a WSMO4J Entity. It retrieves the interface name of the WSMO4J object and takes it as the name of the respective XML DOM node. This is possible as nearly all objects in WSMO4J have the same name as its equivalent in XML syntax. So for example when serializing a WSMO4J WebService, the interface name would be "WebService", the respective DOM node would be "webService". However there is also provided a `setName` method which allows setting the name of the serialized node "manually". Furthermore the EntitySerializer gets the identifier of the Entity and serializes it using the IdSerializer. The serialized id is put as "name" attribute into the serialized node.

As already mentioned nearly all serializer classes extend the EntitySerializer. Therefore the EntitySerializer stores three important protected attributes as shown in figure 3.3: `entity`, `document` and `params`.

The `entity` attribute stores the serialized WSMO4J entity. Subclasses can add their own nodes to this entity node. So for example a WebServiceSerializer would append the serialized interfaces and capability as child nodes to the entity node `<webService>` created by the EntitySerializer. This node is returned by the serialize method which is defined in the EntitySerializer but inherited to all of its subclasses.

The `document` attribute is of type w3c.dom.Document and is responsible for creating XML nodes, elements and attributes. Such nodes, elements and

attributes have to be created in every subclass of the EntitySerializer. Also the elements created in the various serializer classes have to be created all by the same w3c.dom.Document instance. Therefore the DOM Document is instanciated in the XMLSerializer class and passed - as already explained in the beginning of this section - along the various serializer classes in a java.util.HashMap. The EntitySerializer extracts this Document instance from the map and stores it in the protected attribute `document`.

The third attribute is the map storing the namespaces and the mentioned w3c.dom.Document instance.

3.1.3 TopEntitySerializer

The TopEntitySerializer is the parent class for the serializer classes that need to serialize WSMO4J objects containing used mediators and imported ontologies. For this matter the TopEntitySerializer provides two methods, the `addOntology` method which adds the serialization of an imported ontology and the `addMediator` method, which allows adding the serialization of a used mediator.

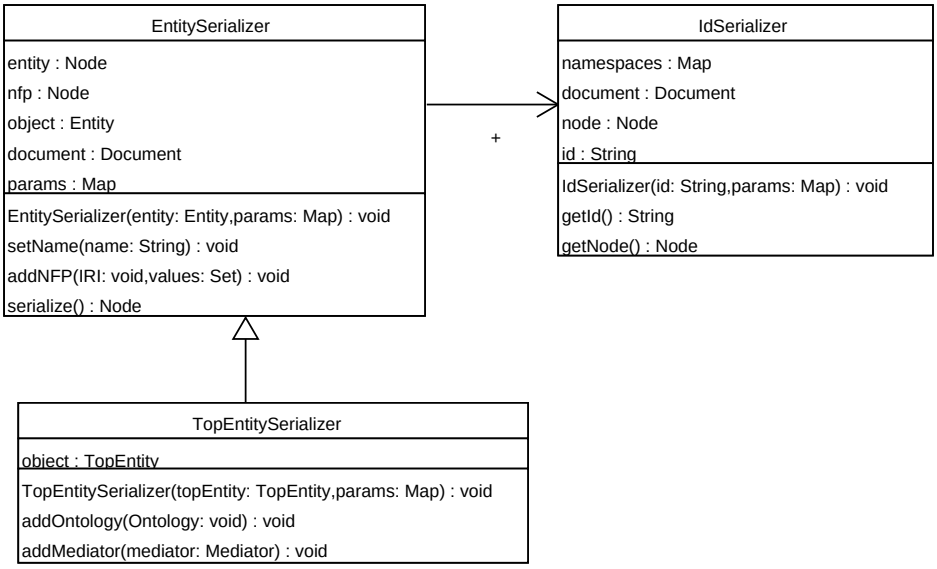

Figure 3.3: EntitySerializer, TopEntitySerializer and IdSerializer

3.1.4 WebService- and GoalSerializer

As Web services and goals are nearly the same in the WSMO specification, the classes for serializing them are also quite similar. Both, the GoalSe-

rializer and the WebServiceSerializer class extend a common super class - namely the ServiceDescriptionSerializer - without adding new methods or attributes.

The ServiceDescriptionSerializer takes over serializing the interfaces and capabilities of a goal or Web service and uses therefore internally the InterfaceSerializer and CapabilitySerializer respectively.

The class diagram for GoalSerializer and WebServiceSerializer is shown below in figure 3.4 and - in greater detail - in Appendix A. The Orchestration- and ChoreographySerializer are not yet in use as for now they are only represented in WSML by an IRI and therefore serialized in the InterfaceSerializer class directly. However these classes exist and can be used in the future as the definition of choreographies and orchestrations will grow.

The two remaining classes are the CapabilitySerializer and the AxiomSerializer. The CapabilitySerializer serializes the preconditions, assumptions, postconditions and effects of a capability by using the AxiomSerializer. In addition it serializes the shared variables between the four elements.

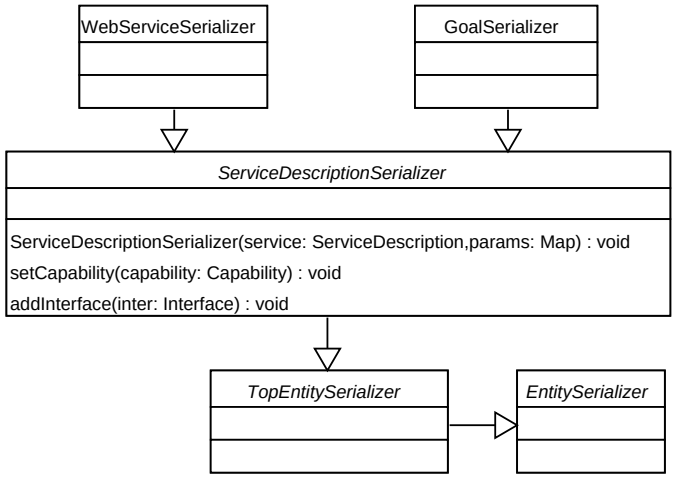

Figure 3.4: Classes for the serialization of Web services and goals

3.1.5 Ontologies

The class for serializing a WSMO4J Ontology is the OntologySerializer. It provides five methods for adding Relations, RelationInstances, Concepts, Instances and Axioms. In this five methods the respective WSMO4J object serializers are used, namely RelationSerializer, RelationInstanceSerializer, ConceptSerializer, InstanceSerializer and AxiomSerializer.

The RelationSerializer provides methods for adding the serialization of super relations and parameters, the ConceptSerializer allows adding super concepts and attributes. The RelationInstanceSerializer provides methods for adding relations to which the serialized instance belongs, the InstanceSerializer allows setting the concepts the serialized instance is a member of. The AxiomSerializer has already been introduced in the previous section.

Figure 3.5 shows the OntologySerializer class. The other classes for serializing an ontology are shown in Appendix A.

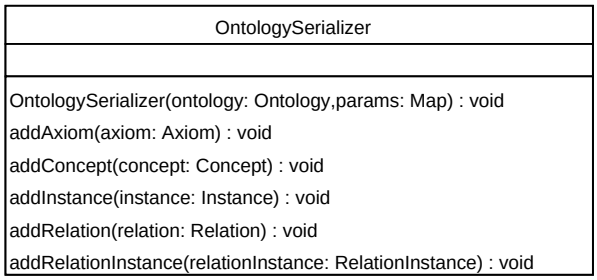

Figure 3.5: The OntologySerializer

3.1.6 Mediators

The MediatorSerializer signs responsible for the serialization of the four kinds of WSMO4J Mediators, the gg-, oo-, wg- and wwMediator. It provides three methods for adding a source to the mediator, setting its target and its mediation service. As a mediator is a top-level element in WSMO4J and therefore extends the TopEntity class, the MediatorSerializer in turn extends the TopEntitySerializer class discussed in Section 3.1.3.

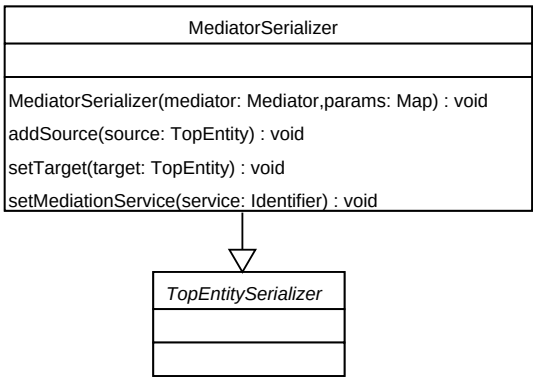

Figure 3.6: The MediatorSerializer

3.1.7 Values

Responsible for the serialization of values - this can be nfp values, attribute or parameter values - is the ValueSerializer. It takes a java.lang.Object as parameter. This object can be of type Integer, String, Float, Double, a WSMO4J DataValue or a WSMO4J IRI.

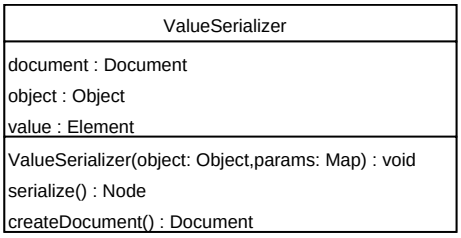

Figure 3.7: The ValueSerializer

The whole structure of the serializer is shown in Appendix A.

3.1.8 XmlDomSerializer

Finally some remarks about the XmlDomSerializer class. It is a workaround class as a problem with the Xerces-J API arose. It is impossible with the current version of Xerces-J to serializes self-defined entities, like for example `&wsml;` correctly. This is necessary, as it should be made possible to abbreviate namespaces with entities in a way not to have to write the whole namespace every time, but to write an abbreviation in form of the mentioned entity. So for example `&wsml;` could be an abbreviation of `http://www.wsmo.org/wsml/wsml-syntax#`. However Xerces-J does not recognize user defined entities as such, and therefore masks the `&` character as it is a special char in XML to `&`. The resulting entity after a Xerces-J serialization is in our example `&wsml;` which of course is not the result we wanted to achieve. The XmlDomSerializer class simply extends the Xerces-J XMLSerializer class, overriding the methods `printEscaped` and `printText`. The XmlDomSerializer is only a helper class and will be eliminated from the package, once this bug of Xerces-J is fixed.

3.2 The parser

The XML syntax to be parsed is first of all validated against an XMLSchema. This schema defines a valid structure for WSML in XML syntax and can be found on the official WSMO website (`http://www.wsmo.org/TR/d16/16.1/v0.2/20050320/xml-syntax/wsml-xml-syntax.xsd`). If the syntax is not valid a ParserException is thrown and the parsing is aborted.

The structure of the parser is quite similar to the structure of the serializer. The differences are that there exists a parser class for each of the four mediator types, as the parsing of the sources and targets is much more complicated then its serialization. An IdParser class does not exist as there is no need for it. The Xerces-J parser already substitutes the entities in the ids with their assigned value. There are two new helper classes that do not have an equivalent in the serializer package: the NodeMap and the Resolver. Their responsibility is discussed in the following sections.

3.2.1 NodeMap

The NodeMap is a helper class to organize the child nodes of an XML element in a better way. It allows retrieving all child nodes of a certain node with a certain name. This avoids looping through all child nodes, checking every node for its element name, as in the NodeMap the elements are structured like in a map. The element names are the keys of the key-value pairs and a vector containing all the element nodes with the respective element name is the value. The methods provided by the NodeMap class are shown in figure 3.8. The `list-` method returns a set of DOM nodes with the respective name passed as parameter. The `get-` method is called when there must exist only one node with the given name. The method checks if there exists only one node with the given name and throws an exception if there exist more.

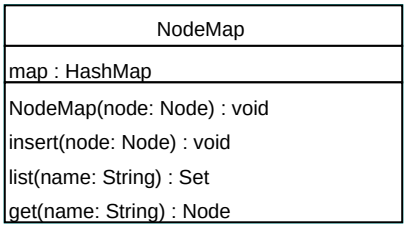

Figure 3.8: The NodeMap class diagram

3.2.2 Resolver

The Resolver takes over finding dependencies between WSMO4J objects. For example a WSMO4J object may import an ontology or use a mediator that is not defined at that point, but will be defined later in the document. Then that object cannot be resolved immediately but only at the end of the parsing. The Resolver class stores such dependencies and disbands them when its resolve method is called. To be exact, the Resolver disbands sources and targets of mediators, used mediators and imported ontologies of TopEntity objects, as well as inverse attributes.

To resolve the targets and sources of mediators and the imported ontologies and used mediators, all TopEntity objects must be registered in the resolver. This occurs in the TopEntity class. The registration is done by calling the `register-` method of the Resolver. Imported ontologies are added via the `addOntology` method, used mediators via the `addMediator` method. Targets and sources are added using the `addTarget` and `addSource` methods respectively.

If it is not possible to resolve one of these dependencies, a warning message is generated and stored. These warnings can be accessed using the `listWarnings` method. However a dependency might not be resolved by the Resolver in some cases even though the dependency is correct. This might happen when resources are defined externally. For example an ooMediator might be defined using the following statement:

```
ooMediator http://www.myexample.com#anOoMediator
source http://www.myexample.com#anotherOoMediator
target http://www.another-url.com#aTarget
```

In this example it is assumed that the target is not defined in the same document but points to another resource on the Web. Now there is no possibility to detect of what type the resource is, because the target of an ooMediator might be an ontology, a goal, a Web service or another mediator. And in WSMO4J it is not possible to create just an instance of type TopEntity. So to create the target, one of the seven Factory methods createOntology, createGoal, createWebService, createOOMediator, createWWMediator, createGGMediator or createWGMediator must be invoked. Which of them should be taken, when the type of the target is unknown?

To allow for now the parsing of XML documents also when externally defined resources are used (as explained previously), an arbitrary instance of an adequate type is created and a warning is generated. The warning is generated because the generated instance may be of an incorrect type. For the previous example of the ooMediator with an irresolvable target, an instance

of the Ontology class is created and used. As the target of a ooMediator may be any TopEntity (table 2.2), the ontology is just an arbitrary choice.

Table 3.3 shows what type is assumed for each mediator's targets and sources when they are defined externally. The class diagram of the Resolver is shown in Figure 3.9.

Table 3.3: Mediator type of sources and targets when externally defined

Mediator	Source	Target
ooMediator	Ontology	Ontology
wwMediator	WebService	WebService
ggMediator	WebService	Goal
wgMediator	Goal	Goal

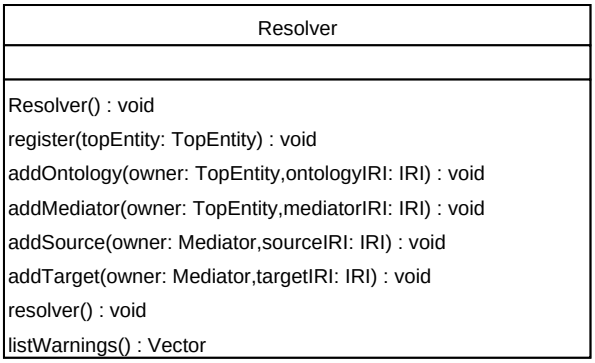

Figure 3.9: The Resolver class diagram

3.2.3 The EntityParser

After taking a look at the helper classes of the parser package, the main parser classes will be introduced, beginning with the EntityParser. Every class extending the EntityParser needs an instance of WsmoFactory to create the WSMO4J objects. Therefore this class creates such an instance. Originally the EntityParser class was designed to create the instances of the entities which are parsed. But this turned out to be not flexible enough, as e.g. Attribute objects are not created by the WsmoFactory but by the Concept class. Also, some methods for creating the WSMO elements need different types of parameters. Therefore the EntityParser class only defines a hook-method "createEntity" which must be implemented by the extending classes. The EntityParser calls this hook-method, gets from it the parsed entity and adds to the entity the non-functional properties.

Hook method design pattern

The Template Design Pattern is used for setting up the outline or skeleton of an algorithm, while the details are left to the specific implementations later. This way, subclasses are able to override some parts of the algorithm without changing its overall structure. This is particularly useful for the separation of the variant and the invariant behavior. It minimizes the amount of code that is written. The invariant part of the algorithm is placed in the abstract class (template) and any subclasses that inherit it may override the abstract methods defined in the superclass and implement the specifics which are needed in that context.[14]

In Appendix B an example of how this hook method design pattern is used in the implementation is shown. The following figure 3.10 shows the EntityParser class diagram.

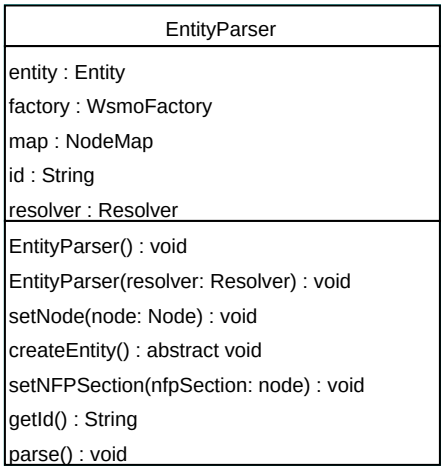

Figure 3.10: The EntityParser class diagram

The constructor of the EntityParser takes either a Resolver instance or no argument. The resolver is only needed by classes extending the Entity through the TopEntity class. Classes which extend the Entity class directly do not need a resolver as they contain no elements which have to be resolved at the end of the parsing.

For the initialization of the EntityParser, the `setNode` method has to be called. This method takes the w3c.dom.Node element which has to be parsed and initializes from its data the following class attributes:

- `id`: the id of the element is parsed out of the id attribute of the node

by calling the private method `getId`.

- `entity`: the WSMO element is created by calling the `createEnity` hook-method.
- `nfpSection`: the nfp section is parsed by calling the `setNFPSection` method.

The subclasses of the EntityParser can manipulate the created WSMO element - which is stored in the `entity` class attribute - by adding additional required attributes. For example the AttributeParser may cast the entity to a WSMO Attribute and add types to it, using the `addType` method. A ConceptParser may cast the entity to a WSMO4J Concept and add a super concept to it.

Also the `factory` as well as the `map` attribute can be used by its subclasses. The map property is a NodeMap instance and handles the child nodes of the node this EntityParser has to parse.

3.2.4 ValueParser

The ValueParser as the name already denotes takes over parsing the values. It takes a w3c.dom.Node object in his constructor representing the value and parses it.

3.2.5 WebService- and GoalParser

Just like in the serializer package, the classes for parsing Web services and goals are quite similar. They both extend a common super class, namely the ServiceDescriptionParser, without adding new methods or attributes.

The ServiceDescriptionParser handles the parsing of the interfaces and capabilities of a goal or Web service and uses therefore internally the InterfaceParser and CapabilityParser respectively.

The InterfaceParser parses the orchestration and choreography. In further version this work will be done by Orchestration- and ChoreographyParser. Their class definition exists, however they are not in use as - for now - they are only represented in WSML by a simple IRI.

The CapabilityParser handles the preconditions, assumptions, postconditions and effects of a capability, making use of the AxiomParser. It parses in addition the shared variables. The AxiomParser parses an axiom which may be of type effect, assumption, precondition or postcondition. The diagram below illustrates how the classes discussed in this section interact.

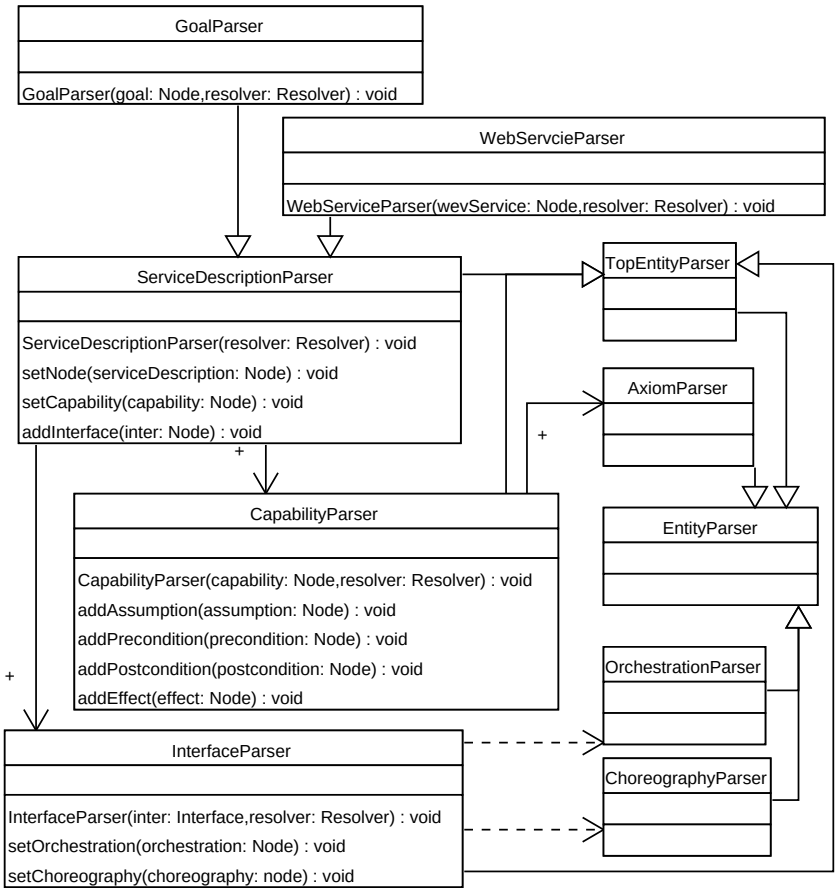

Figure 3.11: WebService- and GoalParser class diagram

3.2.6 Ontologies

The class for parsing ontologies is the OntologyParser. The five public methods provided take over parsing the five types of elements an ontology may contain: concepts, instances, axioms, relations and relation instances. How the parser classes for these elements depend on each other is shown in the diagram in figure 3.12.

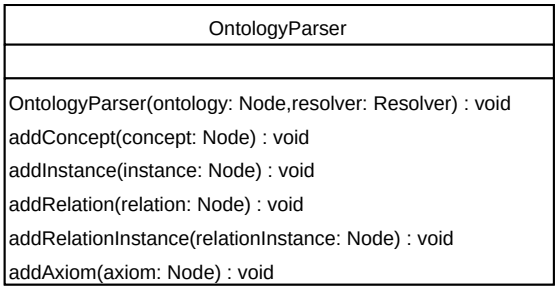

Figure 3.12: OntologyParser class diagram

3.2.7 Mediators

There exists a parser class for each type of mediator. All of these four classes extend a common base class - the MediatorParser - which provides the three methods for setting the mediation service and the target and for adding sources. The four classes do not add any new methods or attributes. However they implement the abstract hook method `createEntity` of the EntityParser class, each one of them differently.

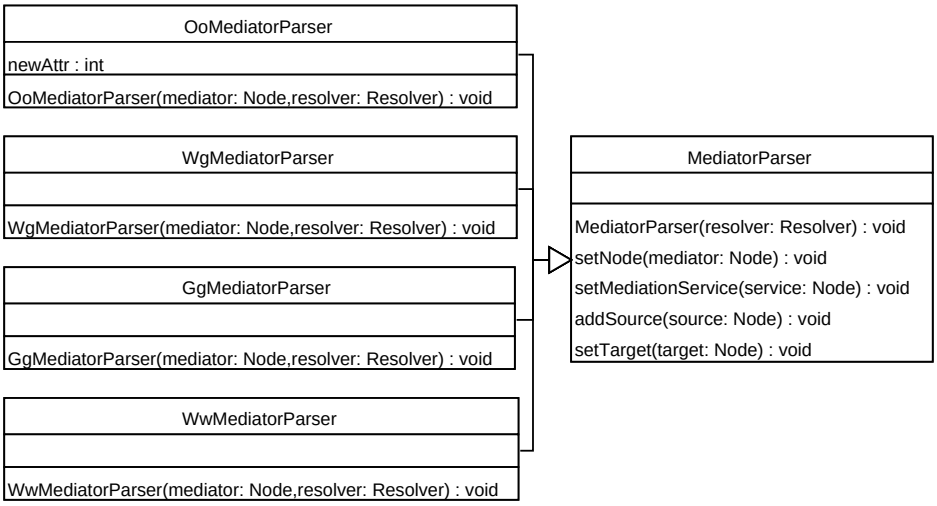

Figure 3.13: The classes for parsing mediators

3.2.8 Exception handling

Every class of the parser package may throw Parser- or InvalidModelExceptions which are defined in the WMSO-API. These exceptions are thrown whenever a parsing error occurs or the WSMO model constructed by the parser is not valid. Most of them are originally thrown by WSMO4J classes, but delivered through the parser classes. In some rare cases the parser classes themselves throw exceptions. Table 3.4 shows which exceptions are thrown in which classes and its cause.

Class	Exception	Cause
AxiomParser	ParserException	Invalid logical expression
NodeMap	ParserException	An element which may occur only once, occurs more times
Resolver	ParserException	Referenced "inverseOf" attribute irresolvable
XMLParser	ParserException	Invalid XML syntax
XMLParser	ParserException	XML schema validation error

Table 3.4: Thrown exceptions

These exceptions have to be caught and handled by the program that uses this extension.

3.3 Usage of the API

The XML parser extension for WSMO4J can be used the following way. An instance of the XMLParser or XMLSerializer can be created using the methods `createParser` or `createSerializer` of the static Factory class respectively.

The implementation to be used has to be specified in a hash map. The key for specifying the implementation is the constant `PROVIDER_CLASS` of the Factory class and the value is the path to the implementation.

```
createParams = new HashMap();
createParams.put( Factory.PROVIDER_CLASS,
org.deri.wsmo4j.io.parser.xml.XMLParser" );
xmlParser = Factory.createParser(createParams);

createParams = new HashMap();
createParams.put( Factory.PROVIDER_CLASS,
"org.deri.wsmo4j.io.serializer.xml.XMLSerializer" );
xmlSerializer = Factory.createSerializer(createParams);
```

Once having an instance of XMLSerializer/XMLParser the methods of the WSMO4J Serializer/Parser interface can be used to make the required serialization or parsing (see tables 3.1 and 3.2).

```
FileReader filereader = new FileReader( "example.xml" );
TopEntity entities[] = xmlParser.parse( filereader );

FileWriter filewriter = new FileWriter( "example.xml" );
xmlSerializer.serialize( entities, filewriter );
filewriter.close();
```

Chapter 4

Conclusion

This document presented the XML parser extension for WSMO4J, an extension for the WSMO4J API, allowing the parsing of XML to WSMO objects and the serialization of WSMO objects to XML. It gave an outline of the used technologies and the solution approach.

The extension is designed in a modular way. This means every serialized or parsed element is handled by an own parser or serializer class. Therefore changes to one WSMO element will not affect the whole extension but only the two classes that take over serializing and parsing the respective element.

WSMO4J is under constant development and changes to the API are made frequently. As a consequence this extension has to be adapted to those changes. However the modular design should make this extension flexible enough to implement those adaptions without too much effort also in the future.

Some problems in the implementation arose when using the WSMO4J API. The mediator sources and targets may be of various types (ontologies, other mediators, Web services, goals). The type of the target/source cannot be retrieved from the WSMO4J API. Therefore, when the target/source is an externally defined resource, it is not possible to detect its type. However this problem will hopefully be solved in further versions of WSMO4J.

In the beginning of the project it was planned to support also a serialization into and parsing from RDF syntax as this language plays an important role for the Semantic Web. However this task is not covered by this thesis but may be added in the future.

The development of the extension started with WSMO version 0.3. The implementation had to be adapted to the subsequent version 0.31 and 0.32

to work finally stable with WSMO version 0.4.

Glossary

DOM: Document Object Model (DOM) is a form of representation of structured documents as an object-oriented model. DOM is the official World Wide Web Consortium (W3C) standard for representing structured documents in a platform- and language-neutral manner.

DTD: A Document Type Definition (DTD for short) is a set of declarations that conform to a particular markup syntax and that describe a class or "type" of SGML or XML documents, in terms of constraints on the structure of those documents.

JAVA: Object oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages which are generally designed to be compiled to native code, Java is compiled to a byte code which is then run (generally using JIT compilation) by a Java virtual machine.

JAVADOC: JAVADOC is a tool for generating documentation in HTML format from comments in Java source code.

RDF: Resource Description Framework (RDF) is a metadata model that is often implemented as an application of XML. The RDF metadata model is based upon the idea of making statements about resources in the form of a subject-predicate-object expression, called a triple in RDF terminology.

SAX: SAX is a serial access parser API for XML and its name is acronymically derived from "Simple API for XML". A SAX parser handles XML information as a stream and is unidirectional, i.e. it cannot renegotiate a node without first having to establish a new handle to the document and reparse.

SOAP: SOAP is a standard for exchanging XML-based messages over a computer network, normally using HTTP.

W3C: The World Wide Web Consortium (W3C) is a consortium that pro-

duces the standards - called "recommendations" - for the World Wide Web.

WSDL: The Web Services Description Language (WSDL) is an XML format published for describing Web services.

WSML: The Web Service Modeling Language WSML provides a formal syntax and semantics for the Web Service Modeling Ontology WSMO.

WSMO: Ontology called Web Service Modeling Ontology (WSMO) for describing various aspects related to Semantic Web services.

WSMO-API: Java API for creating and manipulating WSMO objects.

WSMO4J: Reference implementation of the WSMO-API.

XML: The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages. It is a simplified subset of SGML, capable of describing many different kinds of data. Its primary purpose is to facilitate the sharing of data across different systems, particularly systems connected via the Internet.

XML-RPC: XML-RPC is a remote procedure call protocol encoded in XML.

XMLSchema: XMLSchema, published as a W3C recommendation in May 2001, is one of several XML schema languages. It was the first separate schema language for XML to achieve recommendation status by the W3C.

XSLT: XSL Transformations, or XSLT, is an XML-based language used for transforming XML documents.

Bibliography

[1] Ontotext Lab,
"WSMO4J",
`http://wsmo4j.sourceforge.net/`, last visited: October 14, 2007

[2] de Bruijn, Fensel, Keller, Kifer, Lausen, Krummenacher, Polleres, Predoiu
"Web Service Modeling Language (WSML)",
W3C Member Submission 3 June 2005,
`http://www.w3.org/Submission/WSML`, June 03, 2005

de Bruijn, Lausen, Polleres, Fensel,
"Web Service Modeling Language WSML: An Overview",
3rd European Semantic Web Conf., June 2006, 590-604

[3] de Bruijn, Bussler, Domingue, Fensel, Hepp, Keller, Kifer, König-Ries, Kopecky, Lara, Lausen, Oren, Polleres, Roman, Scicluna, Stollberg
"Web Service Modeling Ontology (WSMO)",
W3C Member Submission 3 June 2005,
`http://www.w3.org/Submission/WSMO`, June 03, 2005

Roman, Keller, Lausen, De Bruijn, Lara, Stollberg, Polleres, Feier, Bussler, Fensel
"Web Service Modeling Ontology",
Applied Ontology, 2005, volume 1, 77-106

[4] The Saxproject,
"SAX - Simple API for XML",
`http://www.saxproject.org/`, last visited: October 14, 2007

[5] Bray, Paoli, Sperberg-McQueen, Maler, Yergeau,
"Extensible Markup Language (XML) 1.0 (Fourth Edition)",
W3C Recommendation 16 August 2006, edited in place 29 September 2006,
`http://www.w3.org/TR/REC-xml`, September 29, 2006

[6] The Apache Software Foundation,
"Xerces Java Parser",
http://xerces.apache.org/xerces-j/, last visited: October 14, 2007

[7] Apparao, Byrne, Champion, Isaacs, Jacobs, Le Hors, Nicol, Robie, Sutor, Wilson, Wood,
"Document Object Model (DOM) Level 1 Specification",
W3C Recommendation 1 October, 1998,
http://www.w3.org/TR/REC-DOM-Level-1/, October 01, 1998

[8] Oracle,
"Oracle XML Developer's Kit",
http://www.oracle.com/technology/tech/xml/xdkhome.html,
last visited: October 14, 2007

[9] The Piccolo Project,
Oren,
"Piccolo XML Parser for Java",
http://piccolo.sourceforge.net/, last visited: October 14, 2007

[10] The Apache Software Foundation,
"Crimson",
http://xml.apache.org/crimson/, last visited: October 14, 2007

[11] Michael H. Kay,
"The Ælfred XML Parser",
http://saxon.sourceforge.net/aelfred.html, November 28, 2002

[12] Le Hors, Le Hégaret, Wood, Nicol, Robie, Champion, Byrne
"Document Object Model (DOM) Level 2 Core Specification",
W3C Recommendation 13 November, 2000,
http://www.w3.org/TR/DOM-Level-2-Core/, November 13, 2000

[13] Le Hors, Le Hégaret, Wood, Nicol, Robie, Champion, Byrne
"Document Object Model (DOM) Level 3 Core Specification",
W3C Recommendation 07 April 2004,
http://www.w3.org/TR/DOM-Level-3-Core/, April 07, 2004

[14] Bob Tarr,
"The Template Method Design Pattern",
http://www.research.umbc.edu/~
tarr/dp/lectures/Template-2pp.pdf, February 12, 2006

[15] Manola, Miller,
"RDF Primer",
W3C Recommendation 10 February 2004,
http://www.w3.org/TR/rdf-primer, February 10, 2004

[16] Mitra, Lafon,
"SOAP Version 1.2 Part 0: Primer (Second Edition)",
W3c Recommendation 27 April 2007
`http://www.w3.org/TR/2007/REC-soap12-part0-20070427`,
April 27, 2007

[17] Booth, Liu
"Web Service Description Language (WSDL) Version 2.0 Part 0: Primer",
W3c Recommendation 26 June 2007
`www.w3.org/TR/2007/REC-wsdl20-primer-20070626`, June 26, 2007

[18] "UDDI",
`www.uddi.org`, last visited: October 14, 2007

Appendix A

Additional class diagrams

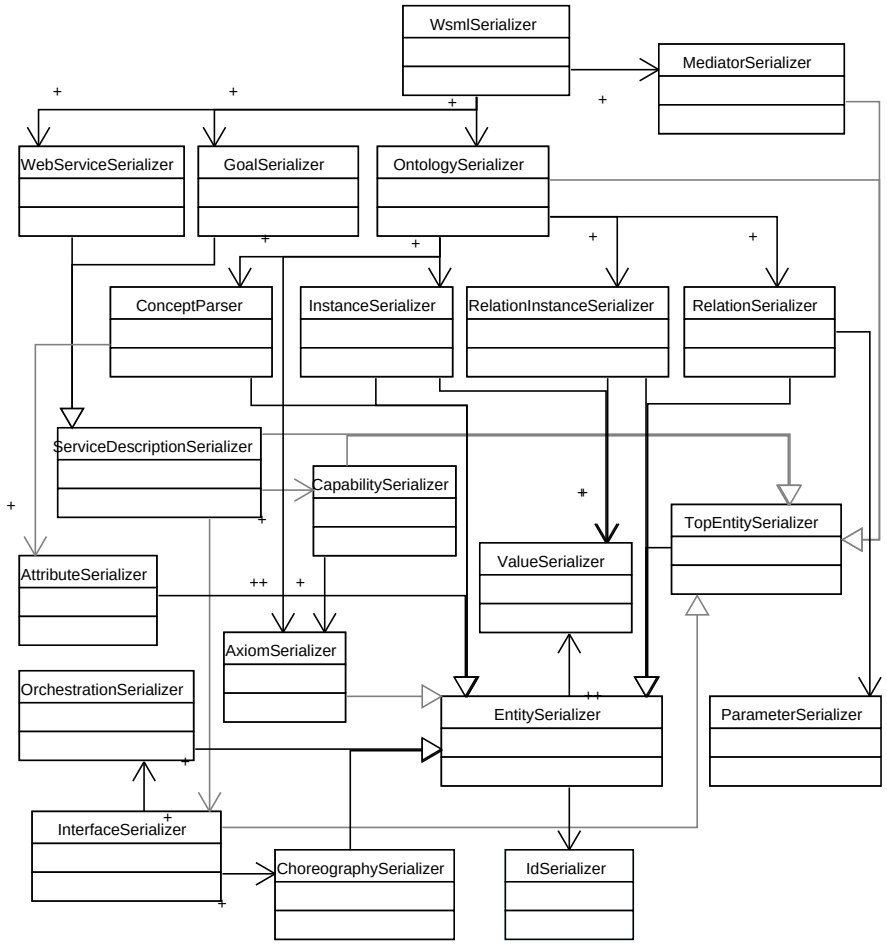

Figure A.1: The serializer full class diagram

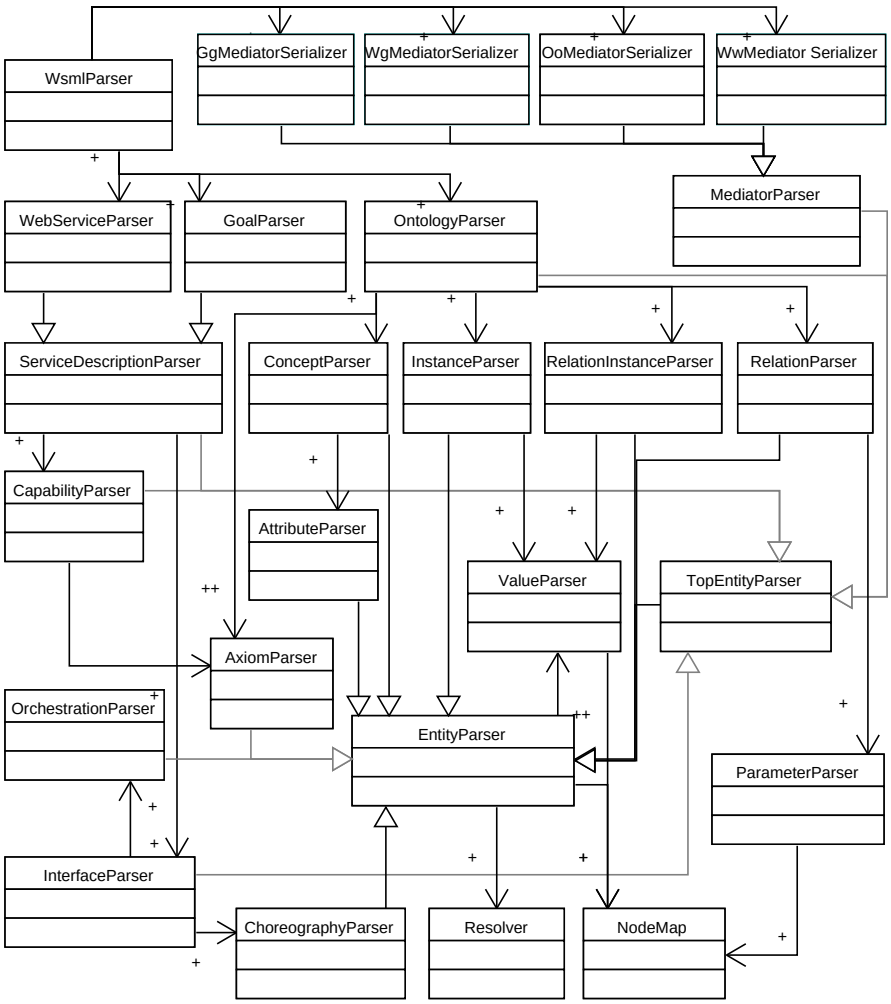

Figure A.2: The parser full class diagram

Appendix B

Hook method design pattern example

Source code extraction of file EntityParser.java:

```
public abstract class EntityParser
{
  ...

  protected Entity entity;

  /**
  * hook method
  * overwrite in subclasses for creating the entity
  **/
  protected abstract Entity createEntity();

  public void setNode( Node entity ) throws
                                     ParserException,
                                     InvalidModelException
  {
    /* build nodeMap */
    map = new NodeMap( entity );

    /* retreive identifier */
    id = getId( entity );
    if( null == id )
    id = factory.createAnonymousID().toString() +
         String.valueOf( ct++ );
    idIri = factory.createIRI( id );
```

```
    this.entity = createEntity();

    /* retreive nfp section */
    Node nfpSection =
         this.map.get( "nonFunctionalProperties" );
    setNFPSection( nfpSection );
  }

  ...
}
```

Source code extraction of file ConceptParser.java:

```
public class ConceptParser extends EntityParser
{
  ...

  /**
  * Hook method of Entity class
  * creates the required Entity object
  * @return returns the created entity
  **/
  protected Entity createEntity()
  {
    Entity e = factory.createConcept( idIri );
    return e;
  }
}
```