wsmo4j Programmers Guide
v. 2.01

Marin Dimitrov
marin.dimitrov@ontotext.com

Vassil Momtchev
vassil.momtchev@ontotext.com

Alex Simov
alex.simov@ontotext.com

Damyan Ognyanoff
damyan.ognyanoff@ontotext.com

Mihail Konstantinov
mihail.konstantinov@ontotext.com

November 24, 2006

Contents

List of Figures

List of Tables

Acknowledgements

wsmo4j is partly funded by the European Commission under the IST projects DIP[1] (FP6-507483), SemanticGov[2] (FP6-027517) and SUPER[3] (FP6-026850) as well as the FIT-IT project RW2[4]

The following people have directly contributed to the *wsmo4j* design and codebase:

- Marin Dimitrov (Ontotext Lab.)
- Holger Lausen (DERI Austria)
- Damyan Ognyanov (Ontotext Lab.)
- Vassil Momtchev (Ontotext Lab.)
- Reto Krummenacher (DERI Austria)
- Alex Simov (Ontotext Lab.)
- Nathalie Steinmetz (DERI Austria)
- Mick Kerrigan (DERI Austria)
- Thomas Haselwanter (DERI Austria)
- James Scicluna (DERI Austria)
- Mihail Konstantinov (Ontotext Lab.)
- Maciej Zaremba (DERI Ireland)

Additionally, many people contributed to *wsmo4j* by way of comments and suggestions for improvements, especially Jacek Kopecky (DERI Austria), Jan Jenke (DERI Austria), Gábor Nagypál (FZI) and Erel Sharf (Unicorn / IBM).

[1] http://dip.semanticweb.org

[2] http://www.semantic-gov.org

[3] http://www.ip-super.org

[4] http://rw2.deri.at/

Chapter 1

Introduction

wsmo4j is an API and a reference implementation for building Semantic Web Service applications based on the Web Service Modelling Ontology (WSMO).

wsmo4j is available under an open source licence, specifically LGPL[1].

The target audience of this guide is comprised of software architects and software developers providing end-user applications or infrastructure components.

The document is structured as follows:

- chapter 2 introduces the refactored and extended version of the WSMO API.
- chapter 4 introduces an extension of the WSMO API that provides functionality for describing WSMO centric service choreographies [10].
- chapter 3 describes an extension of the WSMO API that provides functionality for describing grounding of WSMO descriptions according to [1] and [6].

note: *More details are available in the JavaDoc at* ***http://wsmo4j.sourceforge.net/multiproject/wsmo-api/apidocs/index.html****. The sources are available online at* ***http://wsmo4j.sourceforge.net/multiproject/wsmo-api/xref/index.html***

[1] http://www.opensource.org/licenses/lgpl-license.php

Chapter 2

WSMO API

2.1 Introduction

This chapter introduces the core WSMO API interfaces. The chapter is organised as follows:

- section 2.4, section 2.6 and section 2.7 introduces the interfaces that correspond to various entities defined in the WSMO specification [9]
- section 2.5 introduces the Logical Expressions API[1] which is not part of the core WSMO specification but is elaborated in the WSML specification [4]
- section 2.3, section 2.8, section 2.9 and section 2.10 introduce various "helper" interfaces that are not part of the WSMO domain specification, but instead provide infrastructure functionality (e.g. parsers, validators, factories, etc.).

2.2 Common interfaces

Core interfaces and classes such as *Entity*, *Identifier*, *Namespace* and *NFP* are part of the `org.wsmo.common` package.

[1] The work on Logical Expressions is **not** part of DIP D6.14 funded effort and is briefly presented here only for the purpose of completeness, since parts of the WSMO API and the Choreography API refer to it.

2.2.1 Identifier

Identifier[2] (see Figure 2.1) is the base interface for identifiers and all WSMO entities have an identifier. There are two defined sub-interfaces of Identifier:

- *IRI*, representing an Internationalised Resource Identifier [5]
- *UnnumberedAnonymousID*, representing anonymous unnumbered identifiers, as defined in [4]

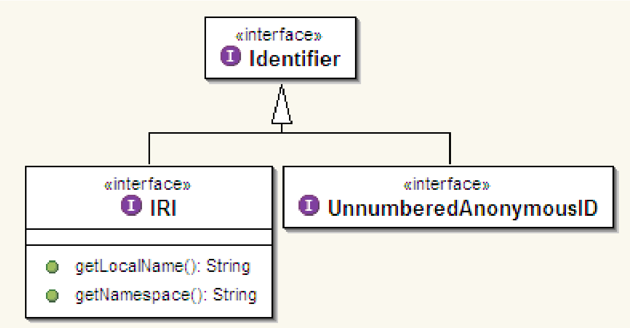


Figure 2.1: Identifier hierarchy

Identifiers are created by the *WsmoFactory* (see section 2.3).

2.2.2 Entity and TopEntity

The base interface for all WSMO entities is *Entity*. All WSMO objects that can be identified are entities:

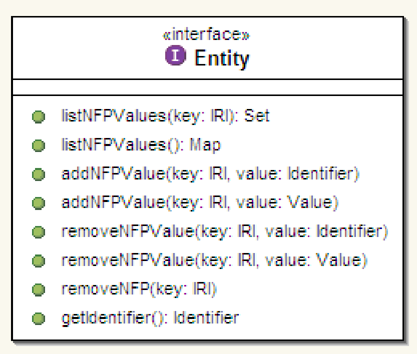
- Attributes, axioms, concepts, instances, relations (see section 2.4)
- Capabilities, goals, services and interfaces (see section 2.6)
- Mediators (oo-mediator, ww-mediator, wg-mediator, gg-mediator) (see section 2.7)

The *Entity* interface is quite generic and provides methods for accessing / modifying the identifier and the non-functional properties associated with the entity (see Figure 2.2). It is important to note that the identifier of an *Entity* is an **immutable** property, i.e. once specified, the *Entity* identifier cannot be changed[3], in order to reduce the possibility that

[2] More information on identifiers is available in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#ids

[3] If such a change is required, then a new *Entity*, with the desired identifier, should be created and the old *entity* should be removed

referential integrity is broken. Other Semantic Web frameworks, such as Jena[4], employ a similar approach toward identifier immutability.



```
«interface»
Entity

listNFPValues(key: IRI): Set
listNFPValues(): Map
addNFPValue(key: IRI, value: Identifier)
addNFPValue(key: IRI, value: Value)
removeNFPValue(key: IRI, value: Identifier)
removeNFPValue(key: IRI, value: Value)
removeNFP(key: IRI)
getIdentifier(): Identifier
```

Figure 2.2: Entity

The *TopEntity* interface (which extends *Entity*) represents the four building blocks in WSMO: services, goals, ontologies and mediators (see Figure 2.3). It provides common functionality for these four entity types:

- accessing and modifying the list of imported ontologies
- accessing and modifying the list of mediators referenced by the entity
- accessing and modifying the list of namespaces defined by the entity
- specifying the WSML variant of the entity description (see subsection 2.2.3)

2.2.3 Namespaces, WSML variants and NFPs

The `org.wsmo.common` package contains three additional classes:

- *Namespace* – represents a namespace binding (i.e. a (prefix, IRI) pair, see Figure 2.4)
- *NFP* – a placeholder for the non-functional property keys as defined by the Dublin Core set[5] [14].
- *WSML* – which contains the identifiers for the five WSML variants (WSML_CORE, WSML_DL, WSML_FLIGHT, WSML_FULL and WSML_RULE). Each TopEntity has a WSML variant specified.

[4] `http://jena.sourceforge.net/`

[5] Note that NFP keys in WSMO / WSML are not restricted to the Dublin Core set and may be extended in an arbitrary way by the user. Check out [13] for an overview of the application of non-functional properties in Web Services

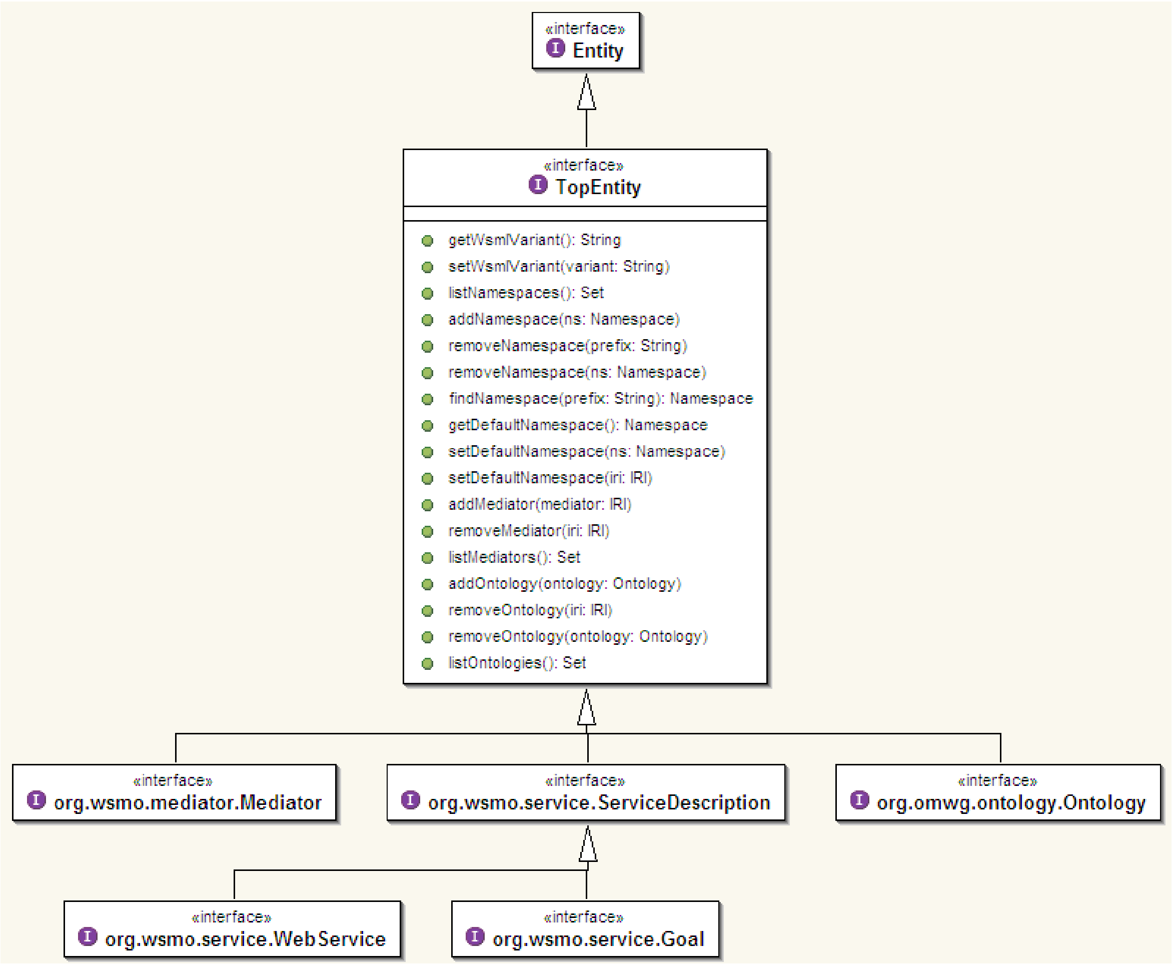

«interface»
Entity
«interface»
TopEntity
getWsmlVariant(): String
setWsmlVariant(variant: String)
listNamespaces(): Set
addNamespace(ns: Namespace)
removeNamespace(prefix: String)
removeNamespace(ns: Namespace)
findNamespace(prefix: String): Namespace
getDefaultNamespace(): Namespace
setDefaultNamespace(ns: Namespace)
setDefaultNamespace(iri: IRI)
addMediator(mediator: IRI)
removeMediator(iri: IRI)
listMediators(): Set
addOntology(ontology: Ontology)
removeOntology(iri: IRI)
removeOntology(ontology: Ontology)
listOntologies(): Set
«interface»
org.wsmo.mediator.Mediator
«interface»
org.wsmo.service.ServiceDescription
«interface»
org.omwg.ontology.Ontology
«interface»
org.wsmo.service.WebService
«interface»
org.wsmo.service.Goal

Figure 2.3: Top Level hierarchy

2.3 Factories

The WSMO API makes heavy use of the *Factory* design pattern (see [7] for details).

There are five factories defined at present[6]:

- *Factory* – this is a *meta-factory* responsible for creating other factories and service objects such as *Parsers*, *Serializers* (explained in section 2.8), *Datastores*, *Locators* (explained in section 2.9) and *Validators* (explained in section 2.10).

  See Figure 2.5 for the UML representation of *Factory*.

  Note that the *Factory* class is a *Singleton*, i.e. there is only one existing instance of the class at any time[7].

[6] Note that new factories may be added in the future as the API is extended with new functionality.

[7] More details on the *Singleton* pattern are available in [7]

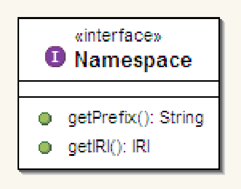


Figure 2.4: *Namespace* interface

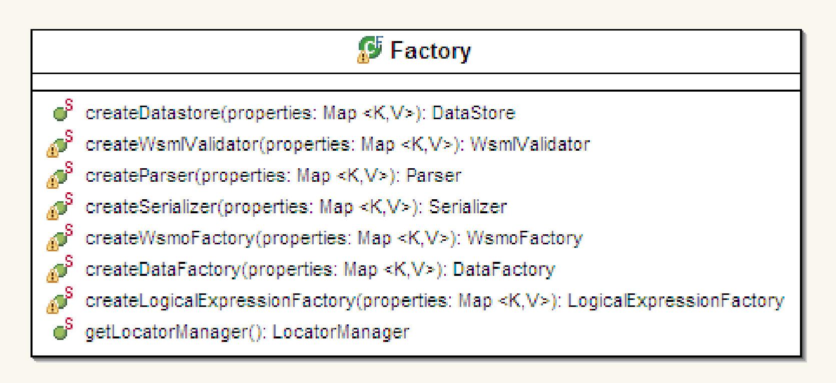

Factory
createDatastore(properties: Map <K,V>): DataStore
createWsmlValidator(properties: Map <K,V>): WsmlValidator
createParser(properties: Map <K,V>): Parser
createSerializer(properties: Map <K,V>): Serializer
createWsmoFactory(properties: Map <K,V>): WsmoFactory
createDataFactory(properties: Map <K,V>): DataFactory
createLogicalExpressionFactory(properties: Map <K,V>): LogicalExpressionFactory
getLocatorManager(): LocatorManager

Figure 2.5: *Factory* class

- *WsmoFactory* – this factory is responsible for:
  - creating identifiers and namespaces (see subsection 2.2.1 and subsection 2.2.3)
  - creating WSMO entities (see subsection 2.2.2)

  See Figure 2.6 for details.

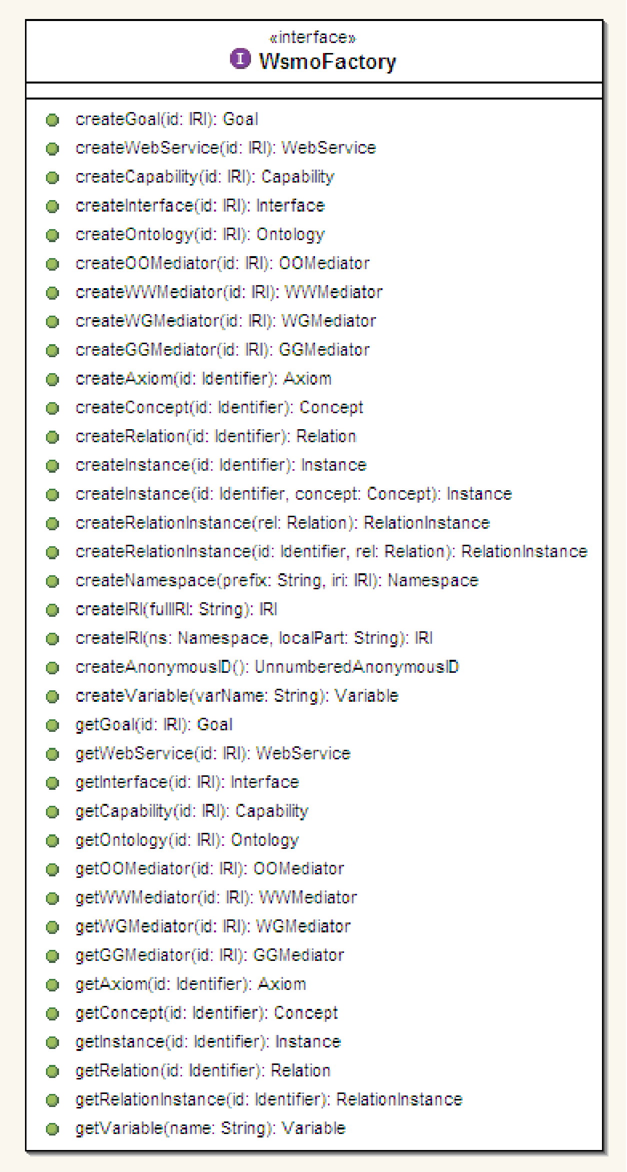
  note: *Note that the WsmoFactory provides both* `create*` *and* `get*` *methods for each WSMO element. The purpose of the former is to create new element descriptions, while the latter return references to existing elements. Note that* `get*` *will still create a new element if no existing element was found. In the future the* `create*` *methods will most probably be removed from the API.*

- *DataFactory* – this factory is responsible for creating instances of the built-in WSML datatypes. There are 19 built-in datatypes in WSML (see [4] for details[8]). The datatypes in the WSMO API are represented by the *SimpleDataType* and *ComplexDataType*, which will be presented in subsection 2.4.2.

  Figure 2.7 presents the *DataFactory* interface.

- *LogicalExpressionFactory* – this factory creates the various logical expression instances (*Atoms*, *Molecules*, *Rules*, etc., which are presented in section 2.5).

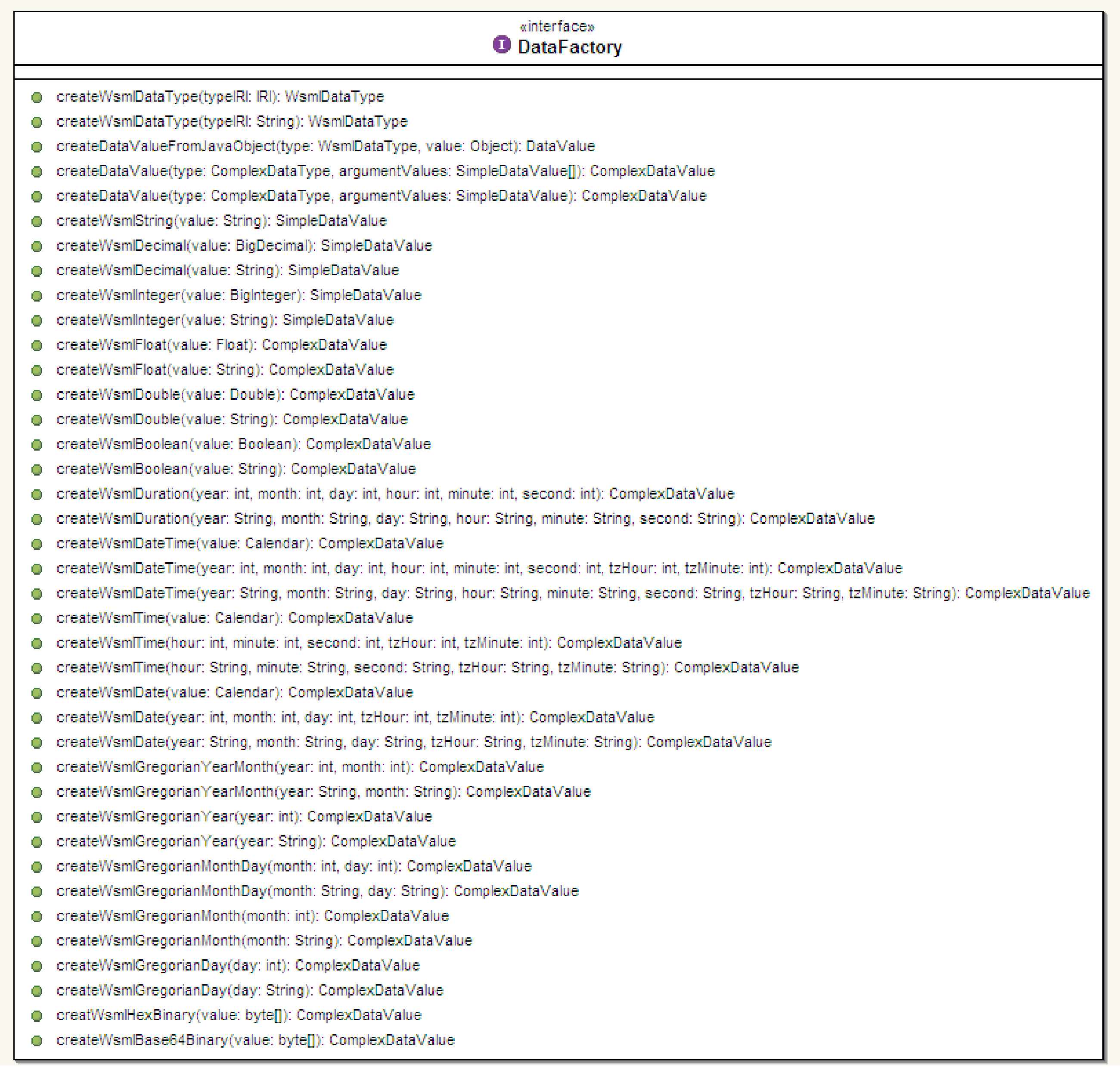
[8] Also available at `http://www.wsmo.org/TR/d16/d16.1/v0.3/#sec:wsml-builtin-datatypes`



```
«interface»
WsmoFactory

createGoal(id: IRI): Goal
createWebService(id: IRI): WebService
createCapability(id: IRI): Capability
createInterface(id: IRI): Interface
createOntology(id: IRI): Ontology
createOOMediator(id: IRI): OOMediator
createWWMediator(id: IRI): WWMediator
createWGMediator(id: IRI): WGMediator
createGGMediator(id: IRI): GGMediator
createAxiom(id: Identifier): Axiom
createConcept(id: Identifier): Concept
createRelation(id: Identifier): Relation
createInstance(id: Identifier): Instance
createInstance(id: Identifier, concept: Concept): Instance
createRelationInstance(rel: Relation): RelationInstance
createRelationInstance(id: Identifier, rel: Relation): RelationInstance
createNamespace(prefix: String, iri: IRI): Namespace
createIRI(fullIRI: String): IRI
createIRI(ns: Namespace, localPart: String): IRI
createAnonymousID(): UnnumberedAnonymousID
createVariable(varName: String): Variable
getGoal(id: IRI): Goal
getWebService(id: IRI): WebService
getInterface(id: IRI): Interface
getCapability(id: IRI): Capability
getOntology(id: IRI): Ontology
getOOMediator(id: IRI): OOMediator
getWWMediator(id: IRI): WWMediator
getWGMediator(id: IRI): WGMediator
getGGMediator(id: IRI): GGMediator
getAxiom(id: Identifier): Axiom
getConcept(id: Identifier): Concept
getInstance(id: Identifier): Instance
getRelation(id: Identifier): Relation
getRelationInstance(id: Identifier): RelationInstance
getVariable(name: String): Variable
```

Figure 2.6: *WsmoFactory* interface



«interface»
DataFactory

```
createWsmlDataType(typeIRI: IRI): WsmlDataType
createWsmlDataType(typeIRI: String): WsmlDataType
createDataValueFromJavaObject(type: WsmlDataType, value: Object): DataValue
createDataValue(type: ComplexDataType, argumentValues: SimpleDataValue[]): ComplexDataValue
createDataValue(type: ComplexDataType, argumentValues: SimpleDataValue): ComplexDataValue
createWsmlString(value: String): SimpleDataValue
createWsmlDecimal(value: BigDecimal): SimpleDataValue
createWsmlDecimal(value: String): SimpleDataValue
createWsmlInteger(value: BigInteger): SimpleDataValue
createWsmlInteger(value: String): SimpleDataValue
createWsmlFloat(value: Float): ComplexDataValue
createWsmlFloat(value: String): ComplexDataValue
createWsmlDouble(value: Double): ComplexDataValue
createWsmlDouble(value: String): ComplexDataValue
createWsmlBoolean(value: Boolean): ComplexDataValue
createWsmlBoolean(value: String): ComplexDataValue
createWsmlDuration(year: int, month: int, day: int, hour: int, minute: int, second: int): ComplexDataValue
createWsmlDuration(year: String, month: String, day: String, hour: String, minute: String, second: String): ComplexDataValue
createWsmlDateTime(value: Calendar): ComplexDataValue
createWsmlDateTime(year: int, month: int, day: int, hour: int, minute: int, second: int, tzHour: int, tzMinute: int): ComplexDataValue
createWsmlDateTime(year: String, month: String, day: String, hour: String, minute: String, second: String, tzHour: String, tzMinute: String): ComplexDataValue
createWsmlTime(value: Calendar): ComplexDataValue
createWsmlTime(hour: int, minute: int, second: int, tzHour: int, tzMinute: int): ComplexDataValue
createWsmlTime(hour: String, minute: String, second: String, tzHour: String, tzMinute: String): ComplexDataValue
createWsmlDate(value: Calendar): ComplexDataValue
createWsmlDate(year: int, month: int, day: int, tzHour: int, tzMinute: int): ComplexDataValue
createWsmlDate(year: String, month: String, day: String, tzHour: String, tzMinute: String): ComplexDataValue
createWsmlGregorianYearMonth(year: int, month: int): ComplexDataValue
createWsmlGregorianYearMonth(year: String, month: String): ComplexDataValue
createWsmlGregorianYear(year: int): ComplexDataValue
createWsmlGregorianYear(year: String): ComplexDataValue
createWsmlGregorianMonthDay(month: int, day: int): ComplexDataValue
createWsmlGregorianMonthDay(month: String, day: String): ComplexDataValue
createWsmlGregorianMonth(month: int): ComplexDataValue
createWsmlGregorianMonth(month: String): ComplexDataValue
createWsmlGregorianDay(day: int): ComplexDataValue
createWsmlGregorianDay(day: String): ComplexDataValue
creatWsmlHexBinary(value: byte[]): ComplexDataValue
createWsmlBase64Binary(value: byte[]): ComplexDataValue
```

Figure 2.7: *DataFactory* interface

- *ChoreographyFactory* – this factory creates elements related to the choreography interface of a service (check out [10] for a detailed overview of WSMO choreographies). The choreography elements such as *Modes*, *Rules* and *Containers* will be presented in chapter 4.

The general purpose factories are part of the `org.wsmo.factory` package, though some specific factories (such as the *ChoreographyFactory*) are part of the respective extension packages.

2.4 Ontologies

The `org.omwg.ontology` package is the core package in the WSMO API, that contains interfaces related to ontology modelling.

2.4.1 Ontology

This is the central interface in the package (see Figure 2.8). An *Ontology*[9] contains a set of related *Concepts*, *Relations*, *Instances* and *Axioms*.

Since an *Ontology* extends *TopEntity* (subsection 2.2.2), it can also define *Namespaces* and import *Mediators* or other *Ontologies*.

2.4.2 Types and Values

In the WSMO API there is a distinction between data types and concepts. *Type* is the root interface of the type hierarchy, with *Concept* and *WsmlDataType* being the only possible specialisations (see Figure 2.9).

WsmlDataType is the top level interface representing the built-in WSML types, as defined by [4], which are equivalent to the data types defined in the XML Schema specification [2]: String, Decimal, Integer, Float, Double, IRI, SQName, Boolean, Duration, DateTime, Time, Date, GYearMonth, GYear, GMonthDay, GDay, GMonth, HexBinary, Base64Binary.

The *SimpleDataType* interface represents types such as Boolean, String, Integer, Float, etc., while *ComplexDataType* (see Figure 2.10) represents composite types such as Date(year, month, day), Time(hour, minute, second), etc.

Data types are created by the *DataFactory*, which was already presented in section 2.3.

[9] See also the definition of Ontology in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#ontologies`

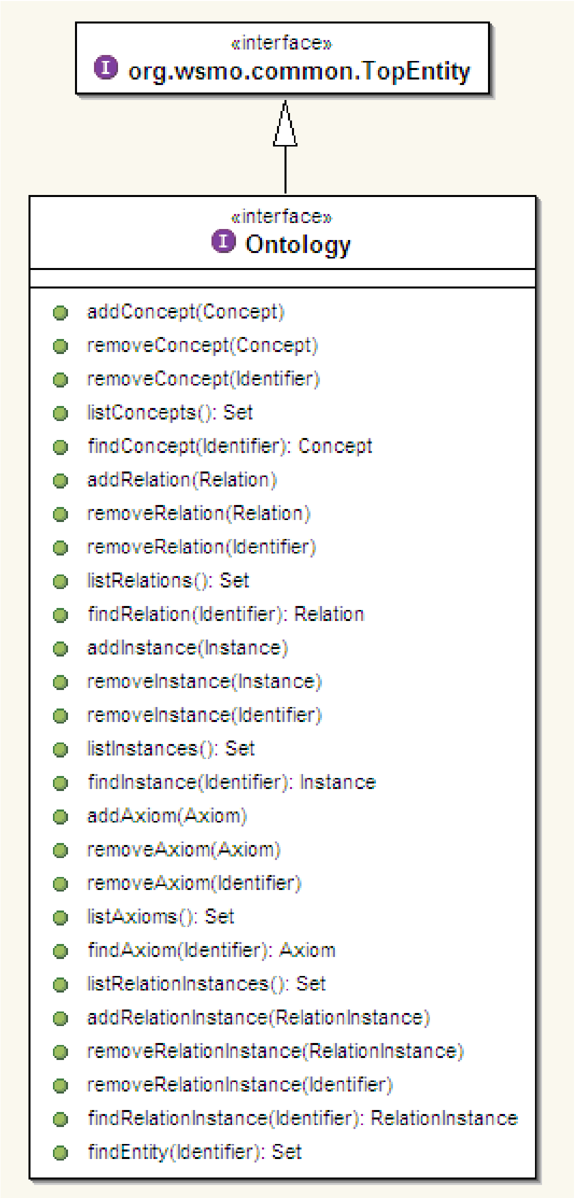

«interface»
org.wsmo.common.TopEntity
«interface»
Ontology
addConcept(Concept)
removeConcept(Concept)
removeConcept(Identifier)
listConcepts(): Set
findConcept(Identifier): Concept
addRelation(Relation)
removeRelation(Relation)
removeRelation(Identifier)
listRelations(): Set
findRelation(Identifier): Relation
addInstance(Instance)
removeInstance(Instance)
removeInstance(Identifier)
listInstances(): Set
findInstance(Identifier): Instance
addAxiom(Axiom)
removeAxiom(Axiom)
removeAxiom(Identifier)
listAxioms(): Set
findAxiom(Identifier): Axiom
listRelationInstances(): Set
addRelationInstance(RelationInstance)
removeRelationInstance(RelationInstance)
removeRelationInstance(Identifier)
findRelationInstance(Identifier): RelationInstance
findEntity(Identifier): Set

Figure 2.8: *Ontology* interface

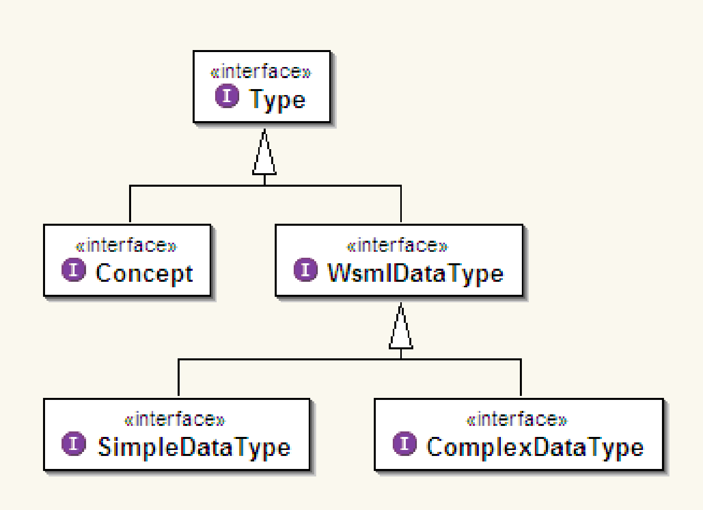


Figure 2.9: Type hierarchy

The *Value* interface provides a common abstraction over instances of concepts and values of the built-in WSML types. Figure 2.11 presents the data value hierarchy. *SimpleDataValues* and *ComplexDataValues* correspond to *SimpleDataTypes* and *ComplexDataTypes* respectively.

DataValues are created by the *DataFactory* (section 2.3)

2.4.3 Axioms

The *Axiom*[10] interface represents a WSML logical expression together with its non-functional properties (Figure 2.12)

Complex logical expressions can be created with the help of the interfaces and classes in the *org.omwg.logicalexpression* package (described in section 2.5)

2.4.4 Concepts, Instances and Attributes

The *Concept* interface (Figure 2.13) represents a concept[11] in a WSMO ontology. A concept may define attributes and may relate to several other super-concepts by an IS-A relation. Since a *Concept* is also an *Entity*, it has an *Identifier* and may define several non-functional properties.

[10] See also the definition of Axiom in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#axioms

[11] See also the definition of Concept in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#concepts

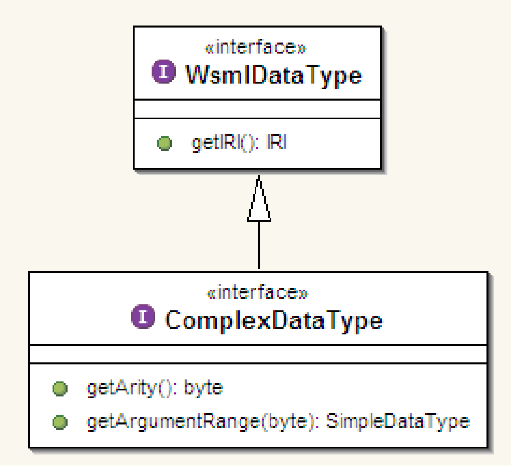


Figure 2.10: *ComplexType* interface

Concepts are created by the *WsmoFactory* (section 2.3). The *Concept* interface provides methods for navigating the concept hierarchy (i.e. listing super-concepts and sub-concepts) as well as listing the set of instances of the concept. *Concept* also serves as a factory for *Attributes*, i.e. attributes are created from the defining concept.

A *Concept* may define one or more *Attributes* (Figure 2.14) representing named slots for data values (for the concept instances). Attributes[12] may be reflexive (e.g. *partOf*), transitive (e.g. *hasAncestor*) or symmetric (e.g. *marriedTo*). Attributes may be associated with an inverse attribute (e.g. *hasParent* is the inverse of *hasChild*), and also specify cardinality constraints.

Note that *Attributes* are local, and thus are not created by the *WsmoFactory*. Instead, they are created by the defining *Concept* (see the *createAttribute(Identifier)* method).

The *Instance*[13] interface (Figure 2.15) represents an instance of a concept defined in an ontology. An instance may be associated with more than one concept (or with no concept at all). Instances may specify values for the attributes defined by the respective concepts.

Instances are created by the *WsmoFactory* (section 2.3).

[12] See also the definition of Attribute in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#concepts`

[13] See also the definition of Instance in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#instances`

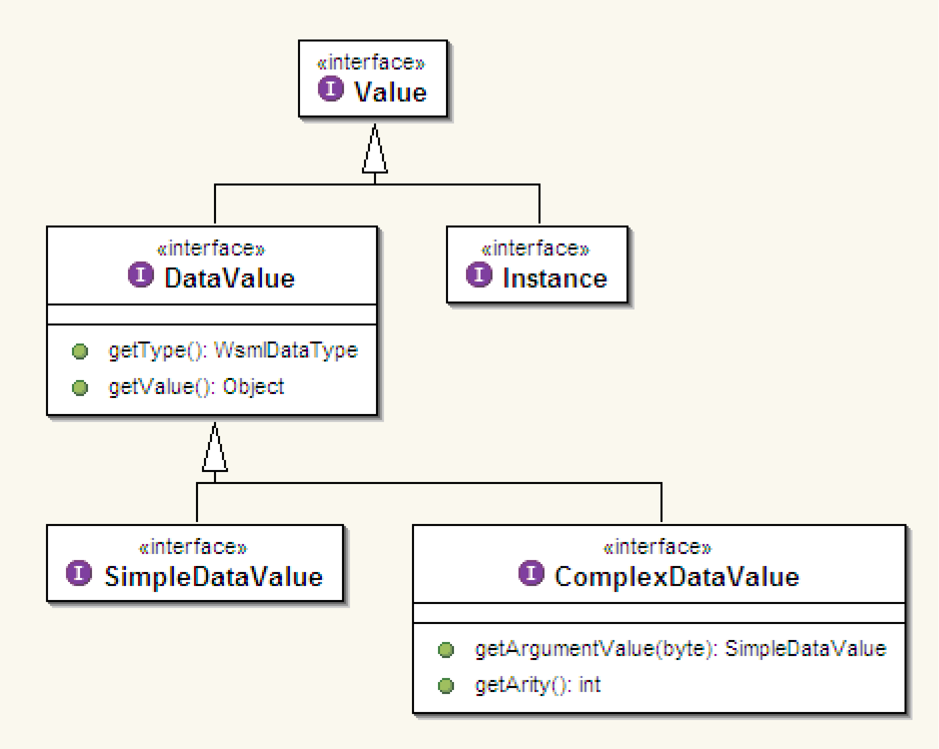


Figure 2.11: Data values

2.4.5 Relations, Relation Instances and Parameters

The *Relation*[14] interface (Figure 2.16) represents a relation definition in a WSMO ontology. Relations are used to model interdependencies between several concepts. A relation may be a specialisation of one or more super-relations.

A *Relation* may define zero or more *Parameters*[15] (Figure 2.17).

Relations are created by the *WsmoFactory* (section 2.3).

Note that *Parameters* are local, and thus are not created by the *WsmoFactory*. Instead, they are created by the defining *Relation* (see the *createParameter(byte)* method).

The *RelationInstance*[16] interface (Figure 2.18) represents instances of relations. A relation instance is associated with a **single** relation and may specify values for the parameters defined by the respective relation.

[14] See also the definition of Relation in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#relations

[15] See also the definition of Parameter in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#relations

[16] See also the definition of RelationInstance in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#instances

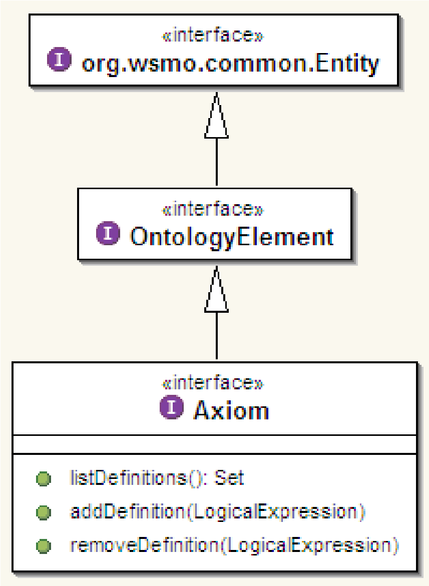


Figure 2.12: *Axiom* interface

RelationInstances are created by the *WsmoFactory* (section 2.3).

2.5 Logical Expressions

note: *The work on Logical Expressions is* ***not*** *part of DIP D6.14 and is briefly presented here only for the purpose of completeness, since parts of the WSMO API and the Choreography API refer to it.*

The packages `org.omwg.logicalexpression` and `org.omwg.logicalexpression.term` are used as object-oriented constructs of the WSML logical expressions. They are used within the *Axiom*, *Capability*, part of WSMO API, or *TransitionRule*, part of the Choreography API (chapter 4) to refine the WSMO elements using a logic language.

Each of the WSML syntaxes imposes different limitation over the following connectivity types: *and*, *or*, *implies*, *impliedBy*, *equivalent*, *neg*, *naf*, *forall*, *exists*, *(*, *)*, *[*, *]*, *,*, *=*, *!=*, *:=:*, *memberOf*, *hasValue*, *subConceptOf*, *ofType* and *impliesType*, as well as the symbols for Logical Programming Rules and database-style constraints: *:-*, *!-* (see [4] for details).

The basic construct of logical expressions are the *Terms* (Figure 2.19), which could be:

- *Constructed term* (function symbol) i.e. *john[age(2005) hasValue 25]*
- Logical expression *Identifier* is less restrictive then the *Identifiers* (introduced in subsection 2.2.1) may be also *NumberedAnonymousID*.

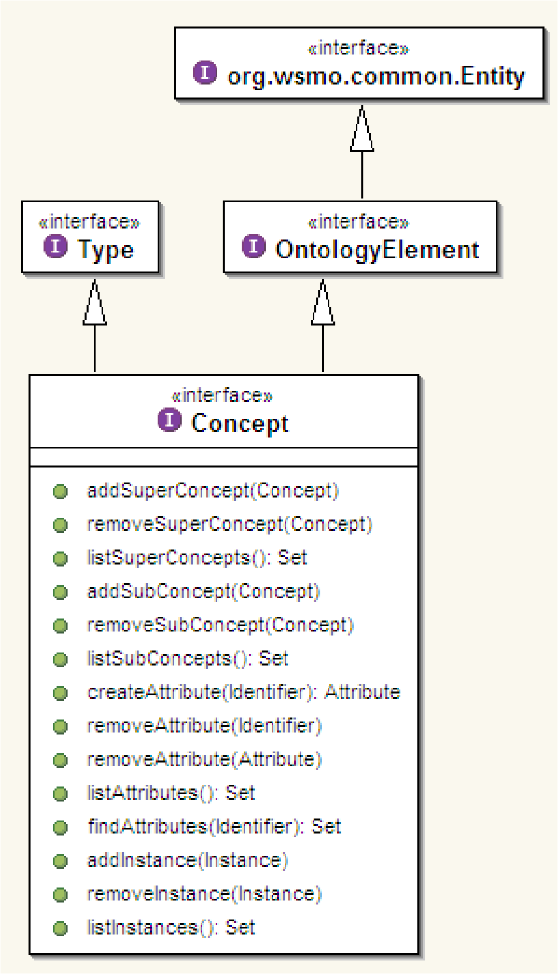


Figure 2.13: *Concept* interface

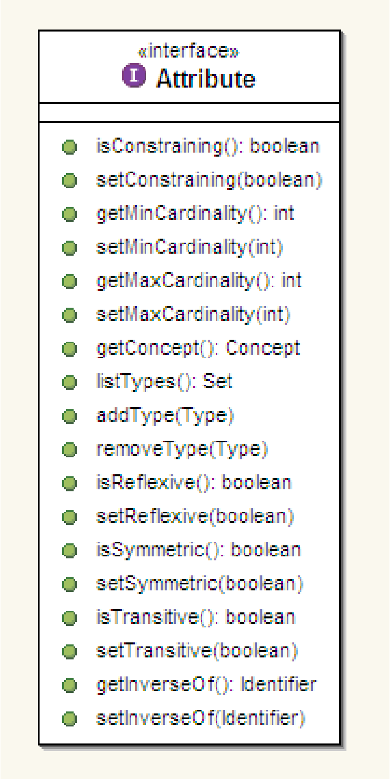


Figure 2.14: *Attribute* interface

- *DataValue john[age(2005) hasValue 12]* (see also subsection 2.4.2)
- Free or shared *Variable* john[age(2005) hasValue ?currentage]

The *LogicalExpression* interface is the super-interface to all logical expression connectivity types. It introduces methods to apply the *Visitor* design pattern (see [7]) and to serialize its content to a *String*. The implementations of *toString(TopEntity)* has to guarantee the correct usage of the namespace context when serializing.

Atom is a predicate symbol with number of parameters (terms) as arguments. *Molecule* is a special type of *Atom*, used to describe information from the conceptual model of the ontology. There are several types of *Molecules*:

- *SubConceptMolecule* π*(X1* ***subConceptOf*** *X2)*
- *MembershipMolecule* π*(X1* ***memberOf*** *X2)*
- *AttributeConstrainMolecule* π*(X1 [X2* ***ofType*** *X3])*
- *AttributeValueMolecule* π*(X1 [X2* ***hasValue*** *X3])*

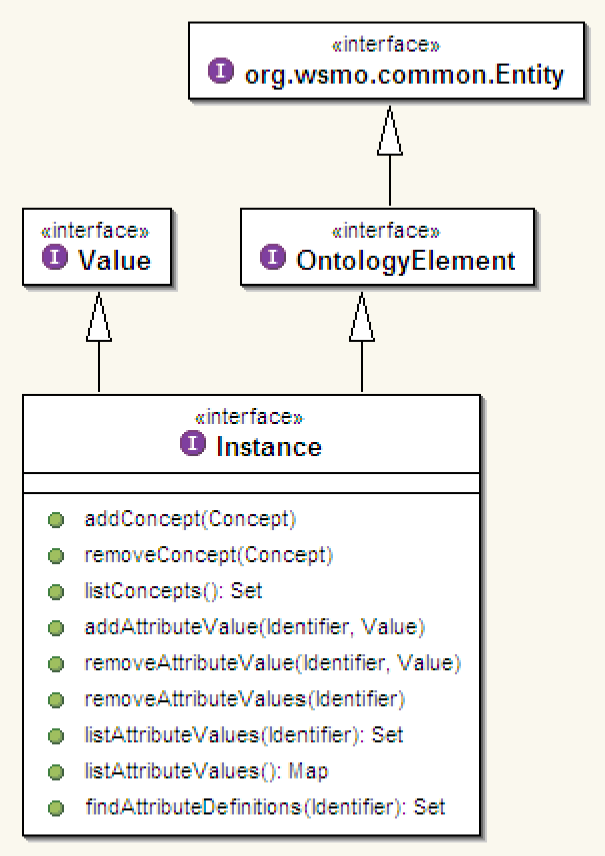

«interface»
org.wsmo.common.Entity
«interface»
Value
«interface»
OntologyElement
«interface»
Instance
addConcept(Concept)
removeConcept(Concept)
listConcepts(): Set
addAttributeValue(Identifier, Value)
removeAttributeValue(Identifier, Value)
removeAttributeValues(Identifier)
listAttributeValues(Identifier): Set
listAttributeValues(): Map
findAttributeDefinitions(Identifier): Set

Figure 2.15: *Instance* interface

- *AttributeInferenceMolecule* π*(X1 [X2* ***impliesType*** *X3])*

Figure 2.20 presents the hierarchy of atomic expressions.

The CompoundExpression interface (Figure 2.21) is a used to connect multiple Atom and/or Molecule expressions.

- *CompoundMolecule* – aggregates several *Molecules* (i.e. *john[age hasValue 25] memberOf Man*)
- *Unary* – defines unary logical expression operators
- *Binary* – defines binary logical expression operators

Logical expressions are created by the *LogicalExpressionFactory* (Figure 2.22).

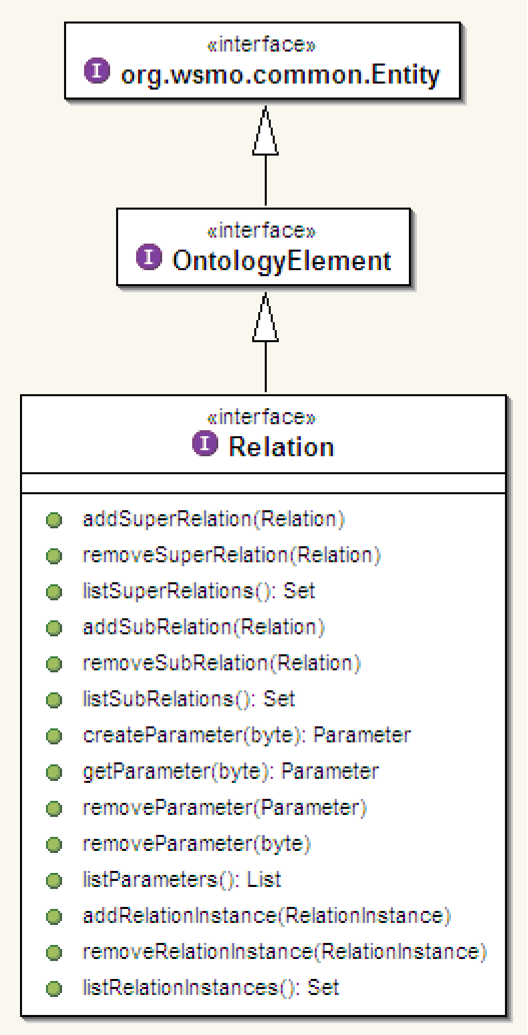

«interface»
org.wsmo.common.Entity
«interface»
OntologyElement
«interface»
Relation
addSuperRelation(Relation)
removeSuperRelation(Relation)
listSuperRelations(): Set
addSubRelation(Relation)
removeSubRelation(Relation)
listSubRelations(): Set
createParameter(byte): Parameter
getParameter(byte): Parameter
removeParameter(Parameter)
removeParameter(byte)
listParameters(): List
addRelationInstance(RelationInstance)
removeRelationInstance(RelationInstance)
listRelationInstances(): Set

Figure 2.16: *Relation* interface

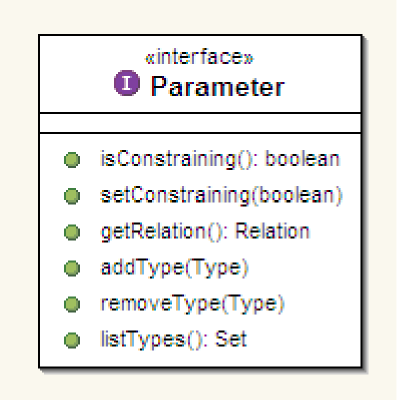


Figure 2.17: *Parameter* interface

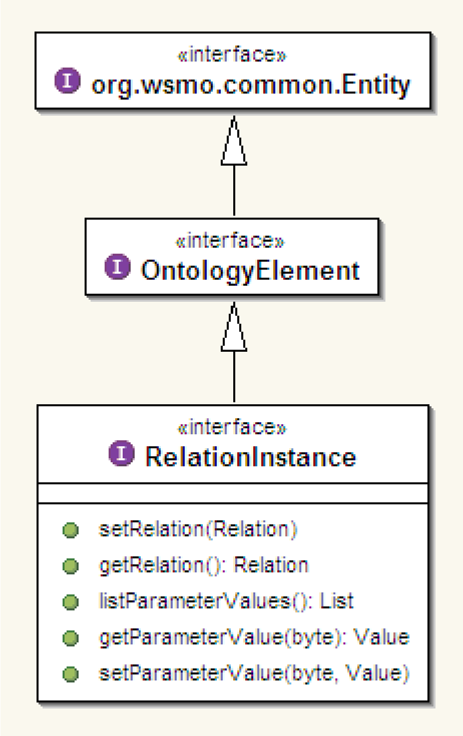


Figure 2.18: *RelationInstance* interface

2.6 Services & Goals

The `org.wsmo.service` package contains the interfaces related to service description in a WSMO centric way - goals, capabilities, services and service interfaces.

The *ServiceDescription* interface (Figure 2.23) is the common super-interface of *WebService*[17] and *Goal*[18]. A Web Service or a Goal in WSMO may be associated with a **single** *Capability* and zero or more *Interfaces*.

The *Capability*[19] interface (Figure 2.24) represents a WSMO capability definition. Capabilities in WSMO are used to formally define the functionality provided by a Web Service by means of the pre-conditions, assumptions, post-conditions and effects of the service (expressed as *Axioms*).

The *Interface*[20] interface (Figure 2.25) represents a WSMO interface (a description of the

[17] See also the definition of Web Service in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#services`

[18] See also the definition of Goal in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#goals`

[19] See also the definition of Capability in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#capability`

[20] See also the definition of Interface in the WSMO specification at `http://www.wsmo.org/TR/d2/v1.2/#interface`

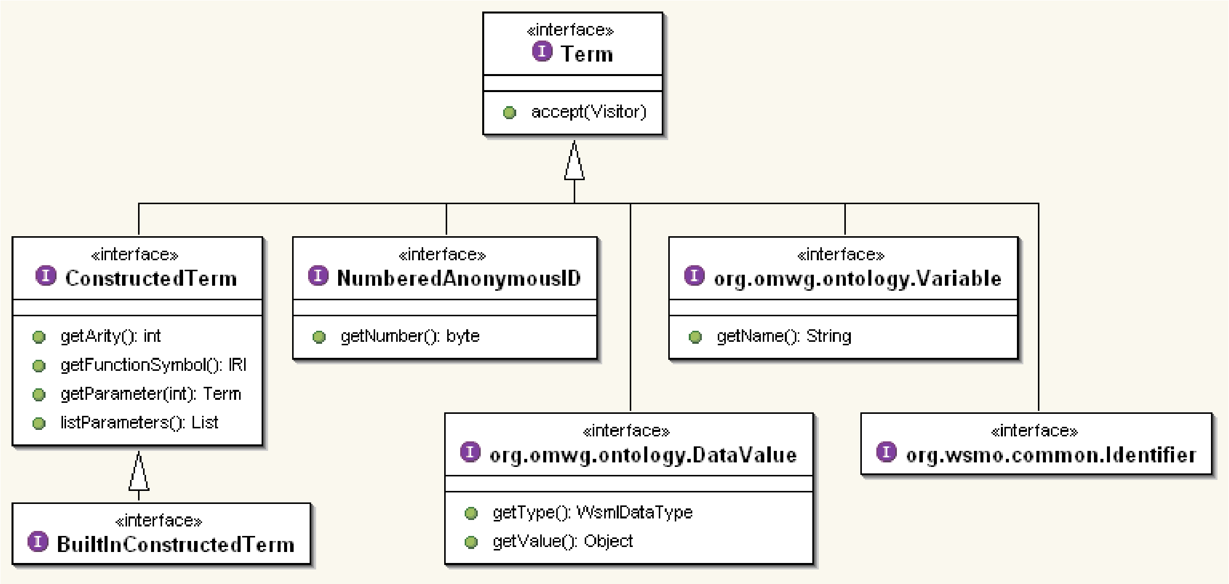


Figure 2.19: Logical expression *Terms*

web service / goal orchestration and choreography). Each *Interface* is associated with at most one *Choreography* and at most one *Orchestration*.

Note that in the WSMO API both *Capability* and *Interface* are top-level entities that can be reused by several Web Services or Goals. This is a slight deviation from the WSMO Specification [9] where capabilities and interfaces cannot be shared and reused. In our opinion, making capabilities and interfaces reusable is very important, since it is most likely that several services will provide functionality satisfying the same capability according to the same interface definition, and such a restriction in the WSMO specification induces unnecessary duplication of capability and interface definitions for each particular service.

2.7 Mediators

Mediators[21] in WSMO provide an abstraction for components that provide interoperability on the data, protocol or process level [9].

There are four types of mediators defined at present:

- *ggMeditors* that link two goals (i.e. state equivalence between goals, or refine source goal into the target goal)

[21] See also the definitions of mediators in the WSMO specification at http://www.wsmo.org/TR/d2/v1.2/#mediators

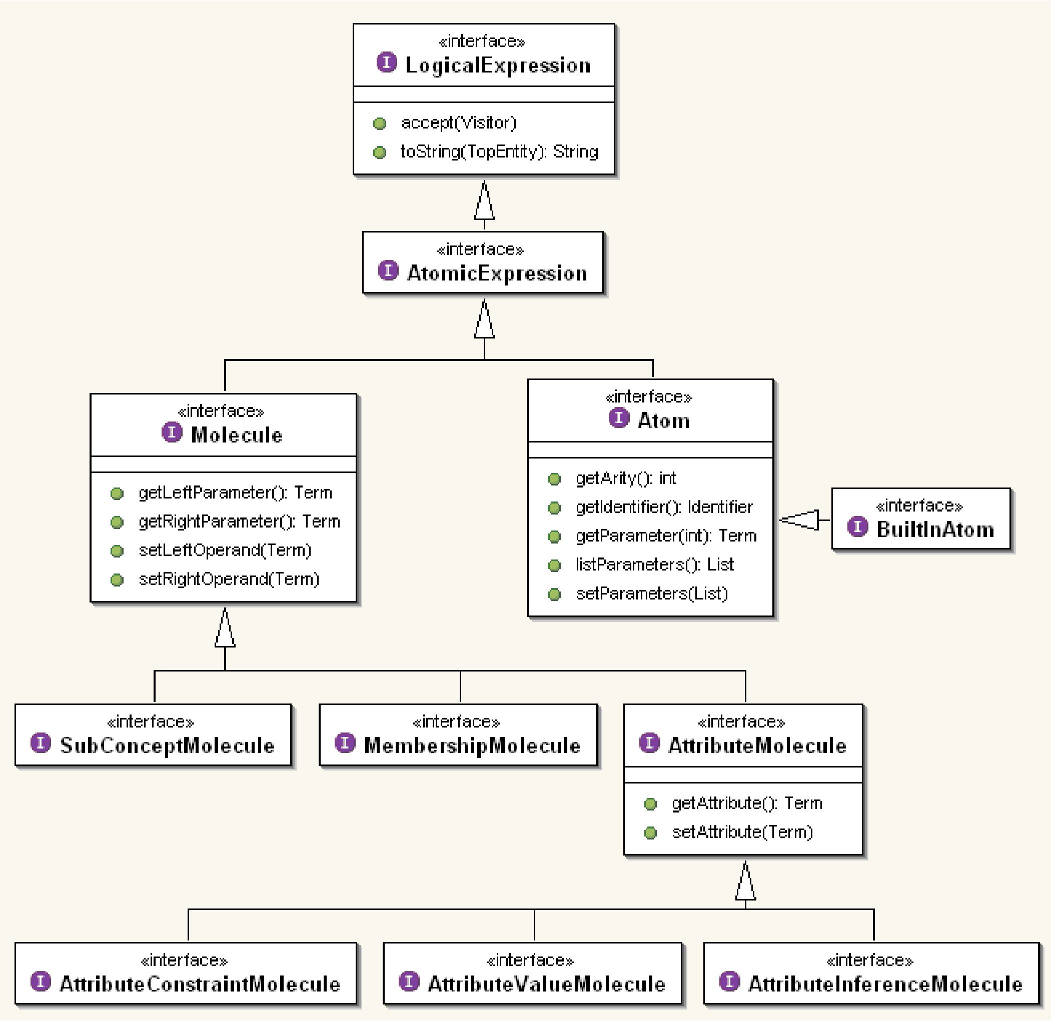


Figure 2.20: Atomic logical expressions

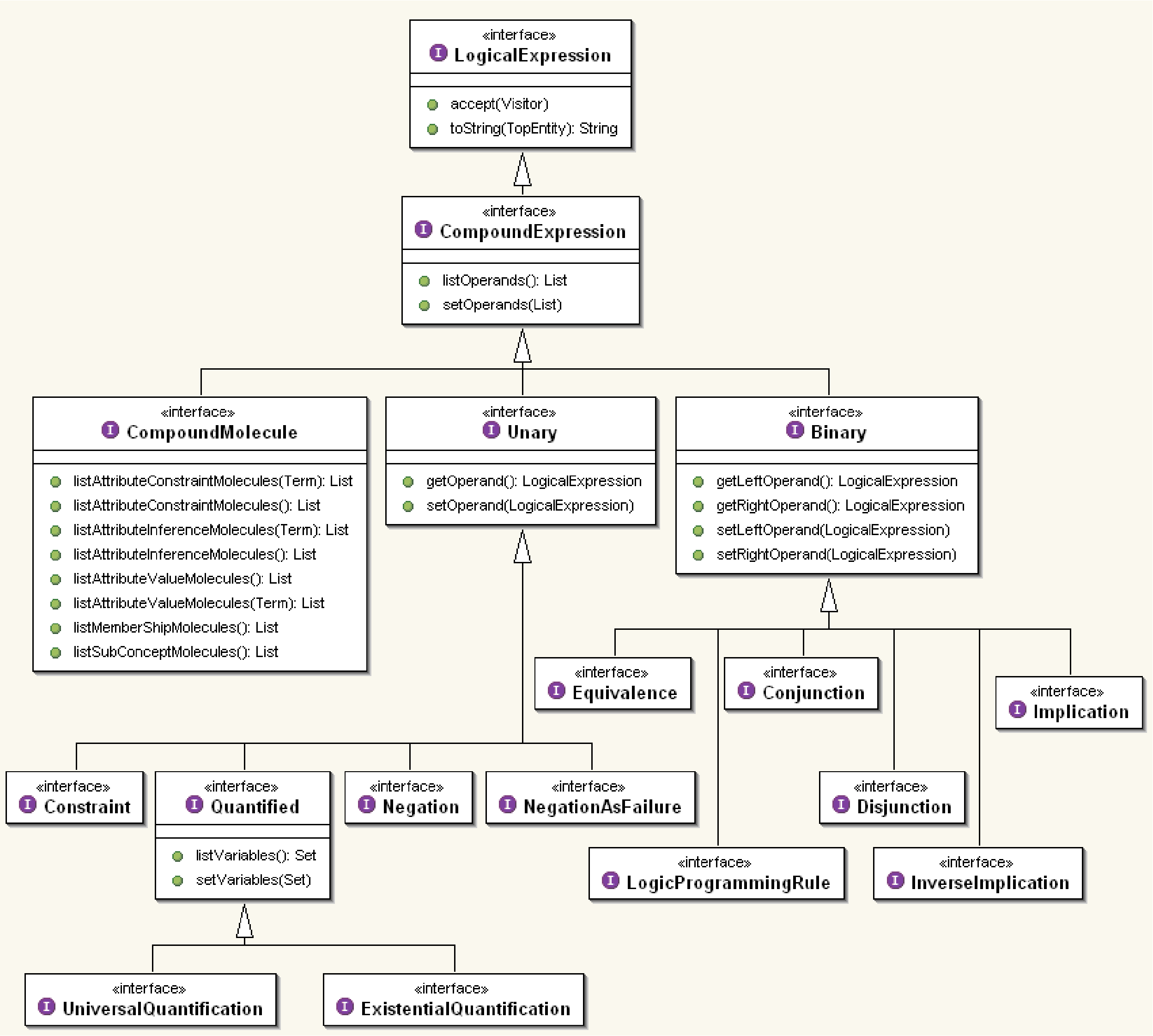
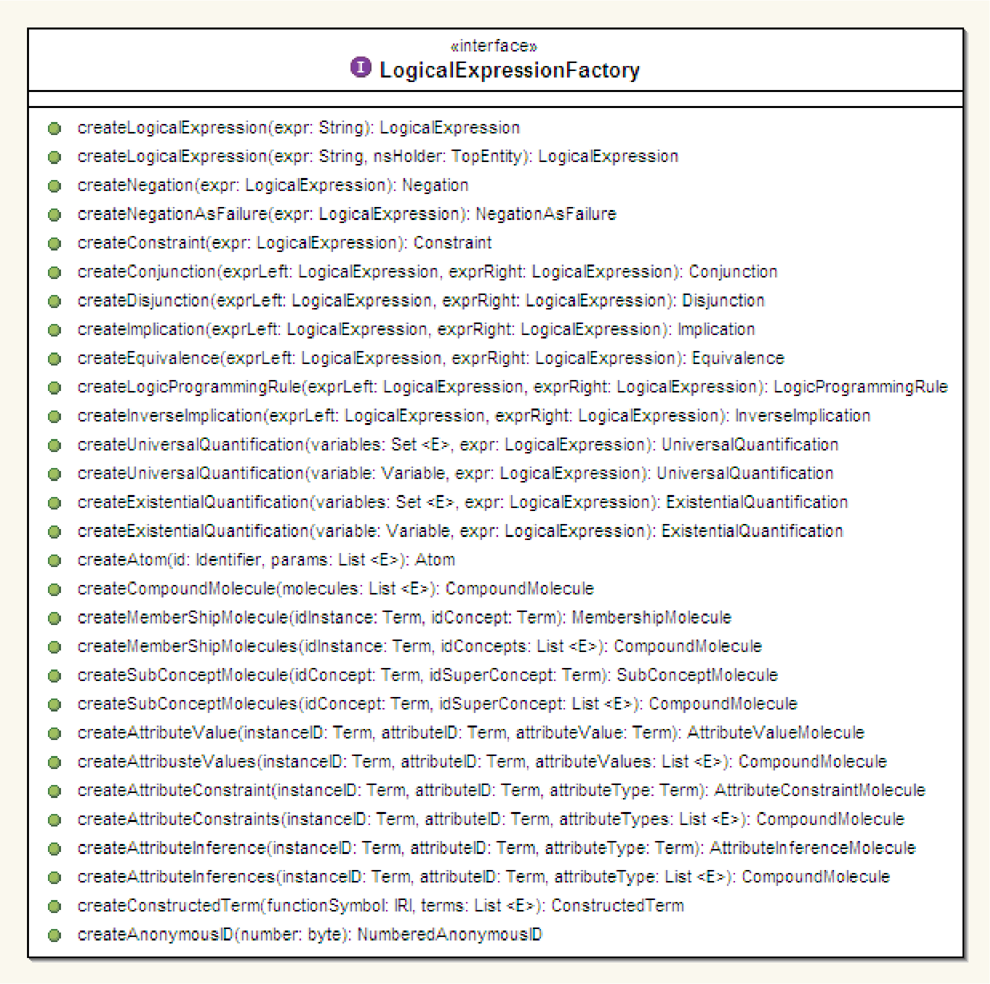

«interface»
LogicalExpression
accept(Visitor)
toString(TopEntity): String
«interface»
CompoundExpression
listOperands(): List
setOperands(List)
«interface»
CompoundMolecule
listAttributeConstraintMolecules(Term): List
listAttributeConstraintMolecules(): List
listAttributeInferenceMolecules(Term): List
listAttributeInferenceMolecules(): List
listAttributeValueMolecules(): List
listAttributeValueMolecules(Term): List
listMemberShipMolecules(): List
listSubConceptMolecules(): List
«interface»
Unary
getOperand(): LogicalExpression
setOperand(LogicalExpression)
«interface»
Binary
getLeftOperand(): LogicalExpression
getRightOperand(): LogicalExpression
setLeftOperand(LogicalExpression)
setRightOperand(LogicalExpression)
«interface»
Equivalence
«interface»
Conjunction
«interface»
Implication
«interface»
Constraint
«interface»
Quantified
listVariables(): Set
setVariables(Set)
«interface»
Negation
«interface»
NegationAsFailure
«interface»
Disjunction
«interface»
LogicProgrammingRule
«interface»
InverseImplication
«interface»
UniversalQuantification
«interface»
ExistentialQuantification

Figure 2.21: Compound logical expressions



```
«interface»
LogicalExpressionFactory

createLogicalExpression(expr: String): LogicalExpression
createLogicalExpression(expr: String, nsHolder: TopEntity): LogicalExpression
createNegation(expr: LogicalExpression): Negation
createNegationAsFailure(expr: LogicalExpression): NegationAsFailure
createConstraint(expr: LogicalExpression): Constraint
createConjunction(exprLeft: LogicalExpression, exprRight: LogicalExpression): Conjunction
createDisjunction(exprLeft: LogicalExpression, exprRight: LogicalExpression): Disjunction
createImplication(exprLeft: LogicalExpression, exprRight: LogicalExpression): Implication
createEquivalence(exprLeft: LogicalExpression, exprRight: LogicalExpression): Equivalence
createLogicProgrammingRule(exprLeft: LogicalExpression, exprRight: LogicalExpression): LogicProgrammingRule
createInverseImplication(exprLeft: LogicalExpression, exprRight: LogicalExpression): InverseImplication
createUniversalQuantification(variables: Set <E>, expr: LogicalExpression): UniversalQuantification
createUniversalQuantification(variable: Variable, expr: LogicalExpression): UniversalQuantification
createExistentialQuantification(variables: Set <E>, expr: LogicalExpression): ExistentialQuantification
createExistentialQuantification(variable: Variable, expr: LogicalExpression): ExistentialQuantification
createAtom(id: Identifier, params: List <E>): Atom
createCompoundMolecule(molecules: List <E>): CompoundMolecule
createMemberShipMolecule(idInstance: Term, idConcept: Term): MembershipMolecule
createMemberShipMolecules(idInstance: Term, idConcepts: List <E>): CompoundMolecule
createSubConceptMolecule(idConcept: Term, idSuperConcept: Term): SubConceptMolecule
createSubConceptMolecules(idConcept: Term, idSuperConcept: List <E>): CompoundMolecule
createAttributeValue(instanceID: Term, attributeID: Term, attributeValue: Term): AttributeValueMolecule
createAttribusteValues(instanceID: Term, attributeID: Term, attributeValues: List <E>): CompoundMolecule
createAttributeConstraint(instanceID: Term, attributeID: Term, attributeType: Term): AttributeConstraintMolecule
createAttributeConstraints(instanceID: Term, attributeID: Term, attributeTypes: List <E>): CompoundMolecule
createAttributeInference(instanceID: Term, attributeID: Term, attributeType: Term): AttributeInferenceMolecule
createAttributeInferences(instanceID: Term, attributeID: Term, attributeType: List <E>): CompoundMolecule
createConstructedTerm(functionSymbol: IRI, terms: List <E>): ConstructedTerm
createAnonymousID(number: byte): NumberedAnonymousID
```

Figure 2.22: *LogicalExpressionFactory* interface

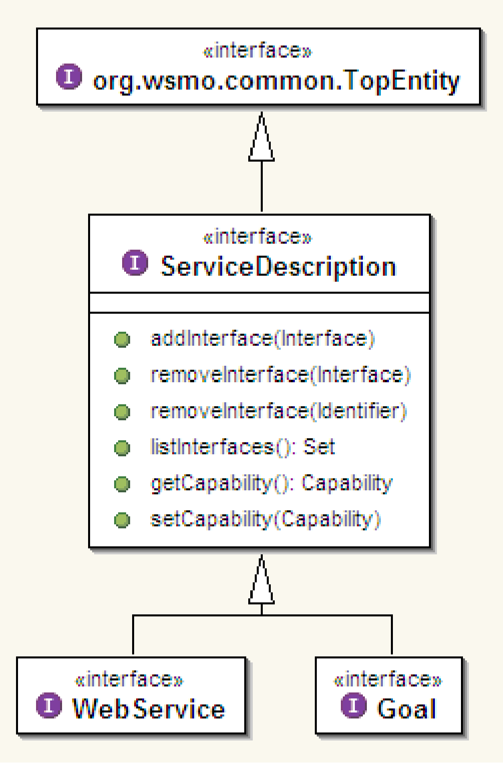

«interface»
org.wsmo.common.TopEntity
«interface»
ServiceDescription
addInterface(Interface)
removeInterface(Interface)
removeInterface(Identifier)
listInterfaces(): Set
getCapability(): Capability
setCapability(Capability)
«interface»
WebService
«interface»
Goal

Figure 2.23: *WebService* and *Goal* interfaces

- *ooMeditors* that provide interoperability between two ontologies
- *wgMeditors* that link services to goals (i.e. specify that the source service fully or partially fulfils the target goal)
- *wwMeditors* that mediate between two services

The corresponding Java interfaces, part of the WSMO API, are presented on Figure 2.26.

2.8 Parsers and Serializers

The `org.wsmo.wsml` package contains interfaces related to import and export of WSML definitions from and into various formats.

At present parsers / serializers for the following languages and formats are available:

- WSML

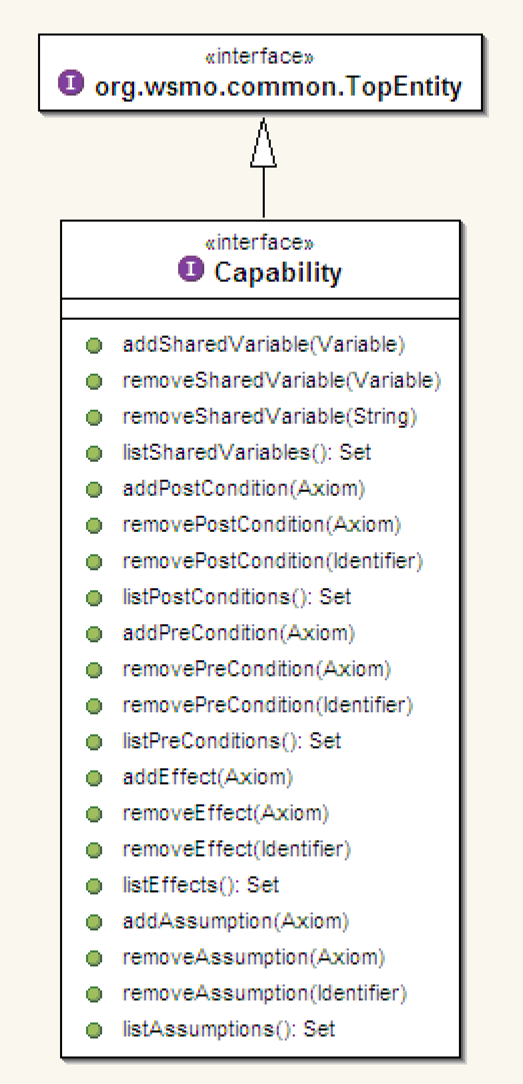

«interface»
org.wsmo.common.TopEntity
«interface»
Capability
addSharedVariable(Variable)
removeSharedVariable(Variable)
removeSharedVariable(String)
listSharedVariables(): Set
addPostCondition(Axiom)
removePostCondition(Axiom)
removePostCondition(Identifier)
listPostConditions(): Set
addPreCondition(Axiom)
removePreCondition(Axiom)
removePreCondition(Identifier)
listPreConditions(): Set
addEffect(Axiom)
removeEffect(Axiom)
removeEffect(Identifier)
listEffects(): Set
addAssumption(Axiom)
removeAssumption(Axiom)
removeAssumption(Identifier)
listAssumptions(): Set

Figure 2.24: *Capability* interface

- WSML-XML, the XML representation of WSML[22]
- a subset of OWL-DL[23] (import only)
- RDF[24] (import only)

Figure 2.27 and Figure 2.28 present the *Parser* and *Serializer* interfaces respectively. *Parsers* and *Serializers* are created by the *Factory* (section 2.3).

[22] See `http://www.wsmo.org/TR/d16/d16.1/v0.3/#sec:wsml-xml` for details

[23] See `http://www.wsmo.org/TR/d16/d16.1/v0.3/#sec:wsml-owl-mapping` for details

[24] See `http://www.wsmo.org/TR/d16/d16.1/v0.3/#sec:wsml-rdf` for details

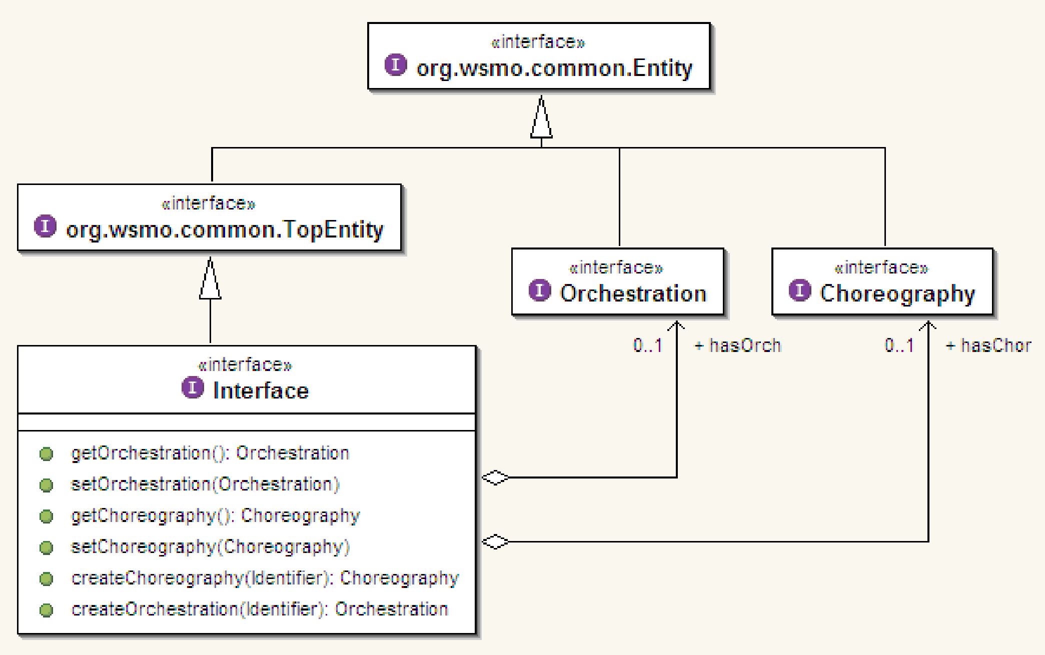

«interface»
org.wsmo.common.Entity
«interface»
org.wsmo.common.TopEntity
«interface»
Orchestration
«interface»
Choreography
0..1
+ hasOrch
0..1
+ hasChor
«interface»
Interface
getOrchestration(): Orchestration
setOrchestration(Orchestration)
getChoreography(): Choreography
setChoreography(Choreography)
createChoreography(Identifier): Choreography
createOrchestration(Identifier): Orchestration

Figure 2.25: *Interface*, *Orchestration* and *Choreography* interfaces

2.9 Datastores and Repositories

The `org.wsmo.datastore` package contains interfaces related to interacting with datastores and repositories for storing WSMO descriptions of ontologies, services, goals and mediators.

The *DataStore* interface (Figure 2.29) provides a simple abstraction of a persistent storage that can be used to store and load WSMO descriptions. The *WsmoRepository* interface further refines *DataStore* by providing specific methods for each top-entity (*Ontology*, *WebService*, *Goal* and *Mediator*).

DataStores and *WsmoRepositories* are created by the *Factory* (section 2.3).

2.10 Validators

The `org.wsmo.validator` package contains interfaces that assist validation of the WSML descriptions created by the WSMO API.

The need for some validation mechanism emerges from the fact that WSMO provides several variants, namely WSML-Core, WSML-DL, WSML- Flight, WSML-Rule and WSML-Full (see [4] for details), and since the variants are based on different logical formalisms a description of a WSMO element created with the WSMO API (e.g. a *TopEntity*) may be valid

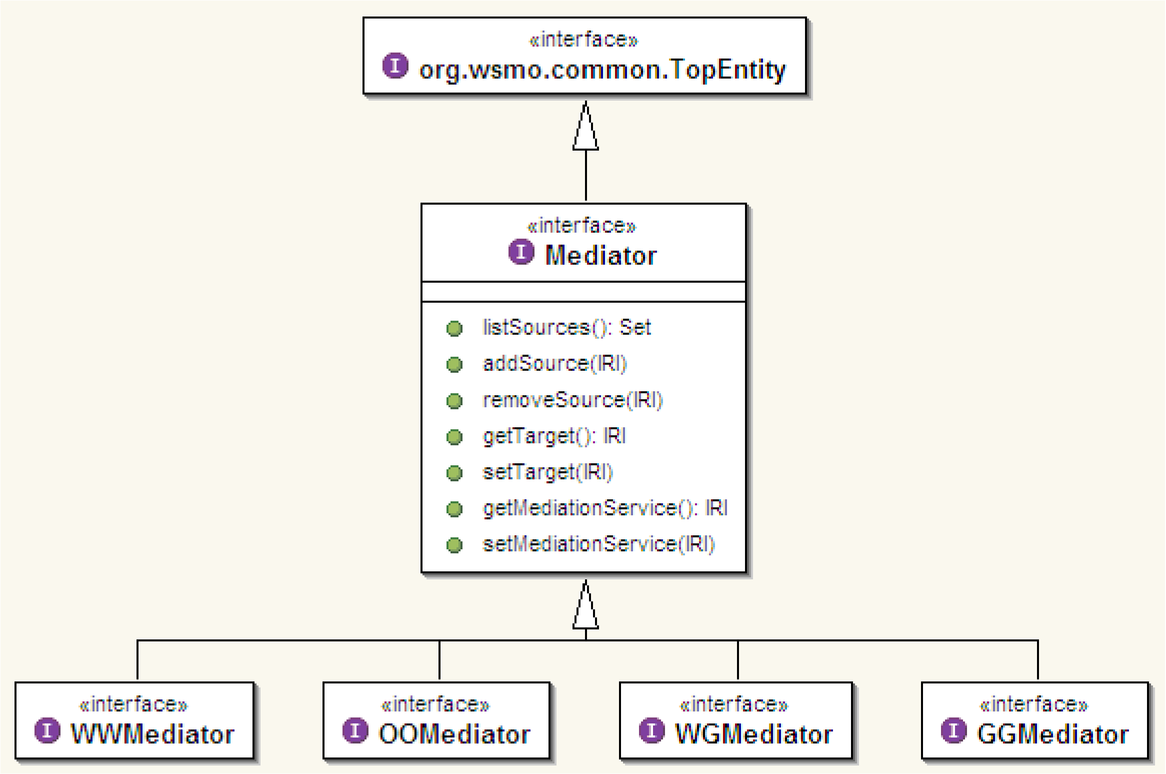


Figure 2.26: Mediator interfaces

in certain variant but not valid in another.

The *Validator* interface (Figure 2.30) provides means for checking the validity of a *TopEntity* according to its specified WSML variant. The validation process produces a list of warnings (i.e. non-critical problems) and errors (critical problems) identified. [11] presents more details about the validation process.

Validation warnings and errors are facilitated by the *ValidationWarning* and *ValidationError* interfaces (Figure 2.31), which are further subclasses into *AttributeError* and *LogicalExpressionError*.

Validators are created by the *Factory* (section 2.3).

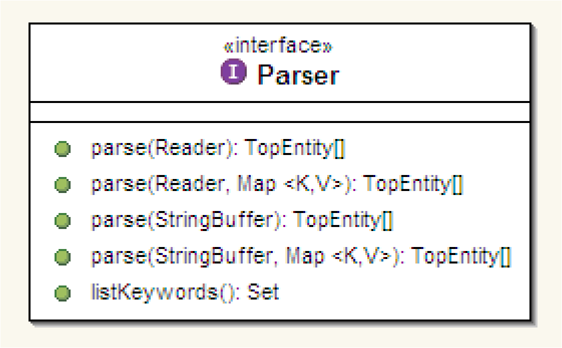

«interface»
Parser
parse(Reader): TopEntity[]
parse(Reader, Map <K,V>): TopEntity[]
parse(StringBuffer): TopEntity[]
parse(StringBuffer, Map <K,V>): TopEntity[]
listKeywords(): Set

Figure 2.27: *Parser* interface

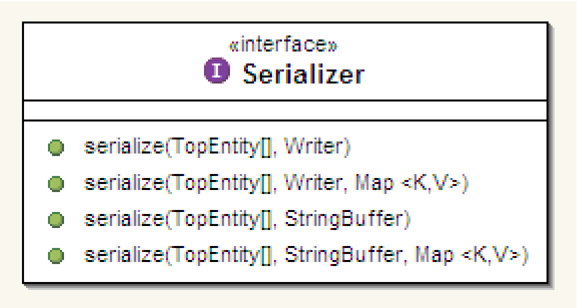

«interface»
Serializer
serialize(TopEntity[], Writer)
serialize(TopEntity[], Writer, Map <K,V>)
serialize(TopEntity[], StringBuffer)
serialize(TopEntity[], StringBuffer, Map <K,V>)

Figure 2.28: *Serializer* interface

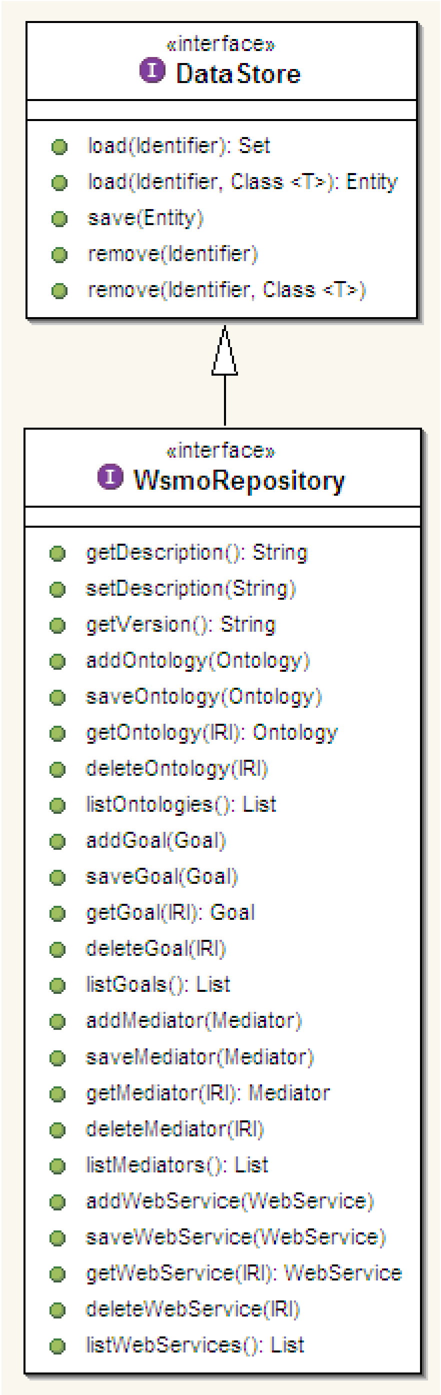


Figure 2.29: *DataStore* and *Repository* interfaces

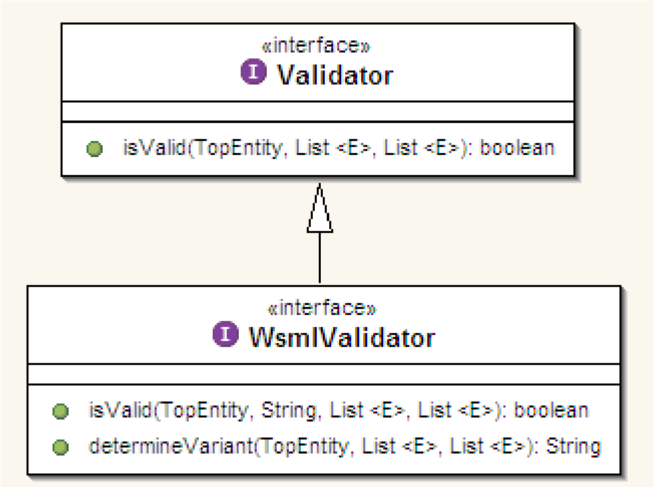

«interface»
Validator
isValid(TopEntity, List <E>, List <E>): boolean
«interface»
WsmlValidator
isValid(TopEntity, String, List <E>, List <E>): boolean
determineVariant(TopEntity, List <E>, List <E>): String

Figure 2.30: *Validator* interface

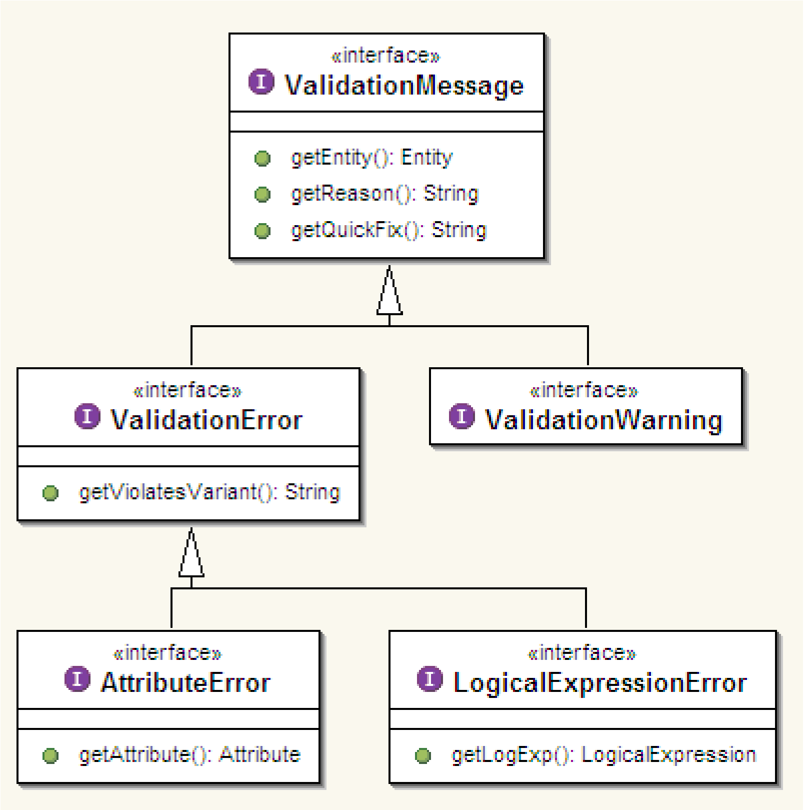


Figure 2.31: Validation warnings and errors

Chapter 3

Grounding API

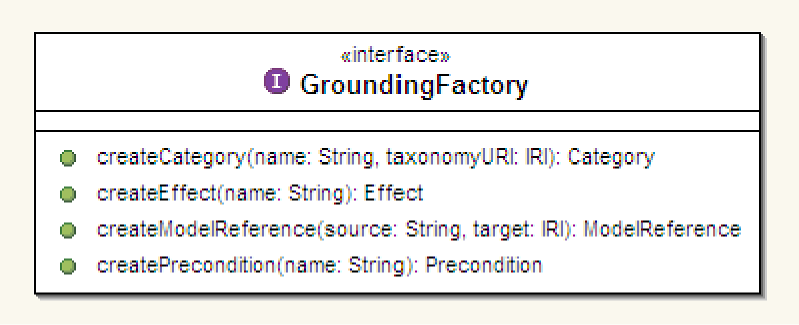
3.1 Introduction

The Grounding API is an extension of the WSMO API, which provides functionality for attaching semantic annotations to WSDL descriptions according to the SAWSDL [6] specifications.

The grounding related interfaces are part of the `org.wsmo.grounding` package.

3.2 Grounding Factory

The *GroundingFactory* interface (see Figure 3.1) is provides the factory[1] for creating grounding related elements (e.g. groundings, model references, assertions, categories, etc.). The *GroundingFactory* itself is created by the *Factory* (see section 2.3).



«interface»
GroundingFactory

```
createCategory(name: String, taxonomyURI: IRI): Category
createEffect(name: String): Effect
createModelReference(source: String, target: IRI): ModelReference
createPrecondition(name: String): Precondition
```

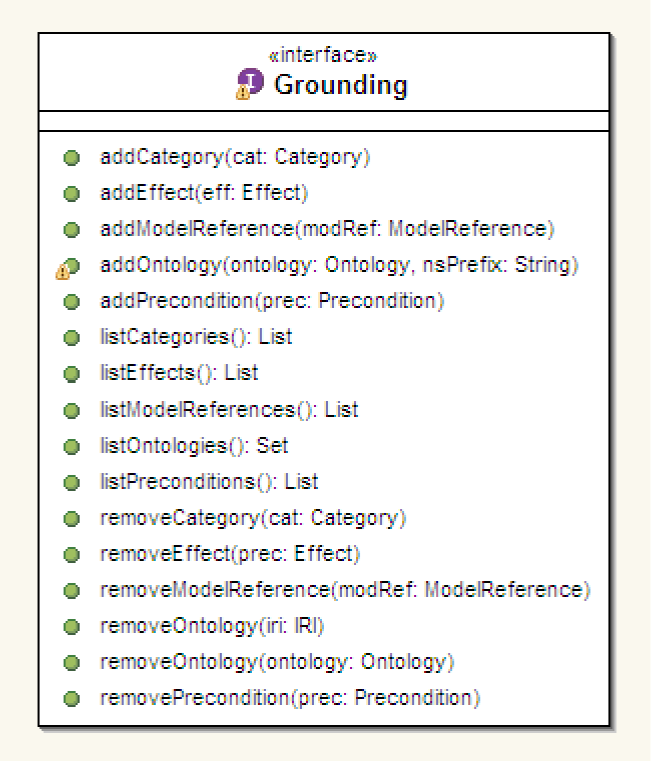
Figure 3.1: *GroundingFactory* interface

[1] See [7] for details on the *Factory* pattern

3.3 Grounding

The *Grounding* interface (Figure 3.2) is the main grounding description, i.e. it contains the mappings between WSDL elements (such as operations and XML types) and WSMO elements (such as concepts and axioms). A grounding description is comprised of:

- zero or more Categories (section 3.4)
- zero or more Effects and Preconditions (section 3.5)
- zero or more Model References (section 3.6)



«interface»
Grounding

```
addCategory(cat: Category)
addEffect(eff: Effect)
addModelReference(modRef: ModelReference)
addOntology(ontology: Ontology, nsPrefix: String)
addPrecondition(prec: Precondition)
listCategories(): List
listEffects(): List
listModelReferences(): List
listOntologies(): Set
listPreconditions(): List
removeCategory(cat: Category)
removeEffect(prec: Effect)
removeModelReference(modRef: ModelReference)
removeOntology(iri: IRI)
removeOntology(ontology: Ontology)
removePrecondition(prec: Precondition)
```

Figure 3.2: *Grounding* interface

3.4 Model References

ModelReferences (Figure 3.3) represent mappings between elements in the two domains (WSDL and WSMO), for example a correspondence between an XML type in the WSDL file and a concept from a WSMO ontology may be specified.

ModelReferences are further divided into *OperationModelReference* (for mappings to WSDL operations), *MessageModelReference* (for mappings to WSDL messages), *FaultModelReference* (for mappings to WSDL faults) and *TypeModelReference* (for mappings to XML types) since there are specific restrictions on the different types of mappings.

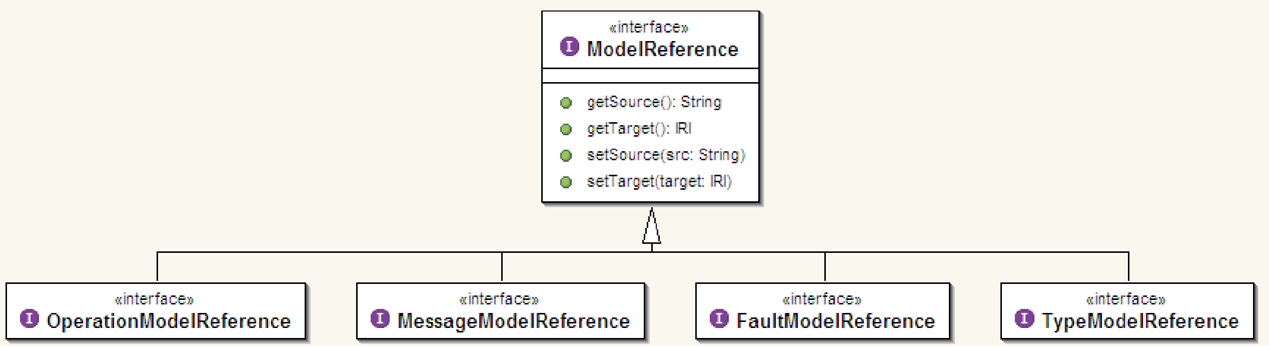


Figure 3.3: *ModelReference* interface

3.5 Assertions

Preconditions or *Effects* (Figure 3.4) may be associated with certain WSDL operations in order to specify assertions that must/will hold before/after a web service operation is invoked. The assertions are either described by means of a *ModelReference* or by means of a logical expression.

3.6 Category

A grounding may be associated with zero or more *Categories* (Figure 3.5), which refer to a specific taxonomy.

3.7 Parsers and Serializers

The *Parser* and *Serializer* interfaces provide ways to import and export the grounding descriptions into the formats specified by WSDL-S [1] and SA-WSDL [6].

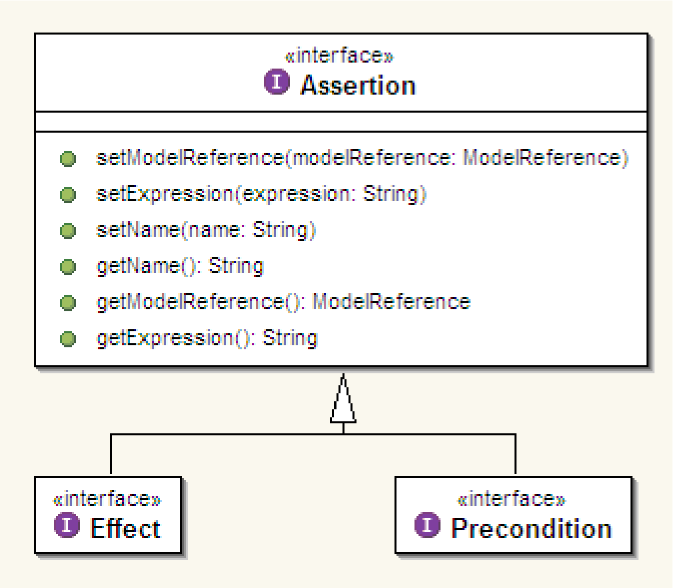


Figure 3.4: *Precondition* and *Effect* interfaces

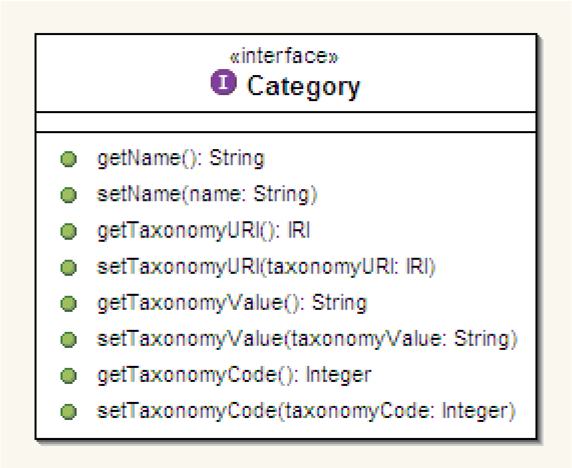

«interface»
Category
getName(): String
setName(name: String)
getTaxonomyURI(): IRI
setTaxonomyURI(taxonomyURI: IRI)
getTaxonomyValue(): String
setTaxonomyValue(taxonomyValue: String)
getTaxonomyCode(): Integer
setTaxonomyCode(taxonomyCode: Integer)

Figure 3.5: *Category* interfaces

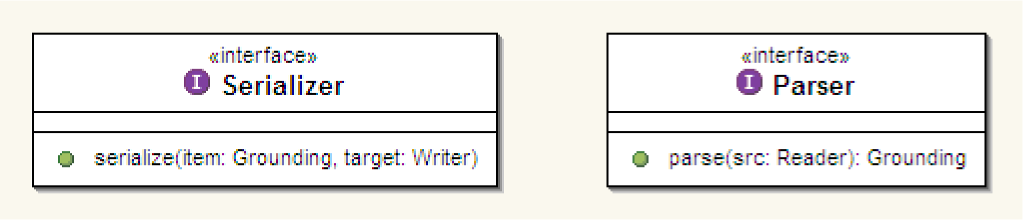

«interface»
Serializer
serialize(item: Grounding, target: Writer)
«interface»
Parser
parse(src: Reader): Grounding

Figure 3.6: *Parser* and *Serializer* interfaces for Grounding

Chapter 4

Choreography API

4.1 Introduction

The Choreography API is an optional extension of the WSMO API, that provides the java interfaces for modelling WSMO centric choreographies based on Abstract State Machines, as specified by [10].

The Choreography API is still evolving and thus is not integrated with the main WSMO API. Besides, keeping the choreography extension separate from the WSMO API core, makes it easier to plug into the WSMO API other choreography modelling approaches such as Cashew ([8]) or ADO ([8]).

The Choreography API interfaces are part of the `org.wsmo.service.choreography` package.

The Choreography API provides a core conceptual model to deal with the description of the service invocation. A state-based approach is used and is inspired from the Abstract State Machine methodology is used. A key extension to the traditional ASM is that the definition of the machine signature is defined in the terms of WSMO ontologies and logical language to dynamically modify the underlying ontologies (see [10] for details). The Choreography is composed of a *state signature* and *transition rules*.

4.2 Choreography

There are two choreography interfaces: `org.wsmo.service.Choreography` (Figure 2.25) is part of the WSMO API and provides no additional methods beside the methods of *Entity* interface. The `org.wsmo.service.choreography.Chorepgraphy` (Figure 4.1), referred in this chapter simply as *Choreography*, is part of the Choreography API and is extended with the support of the state signature and transition rules.

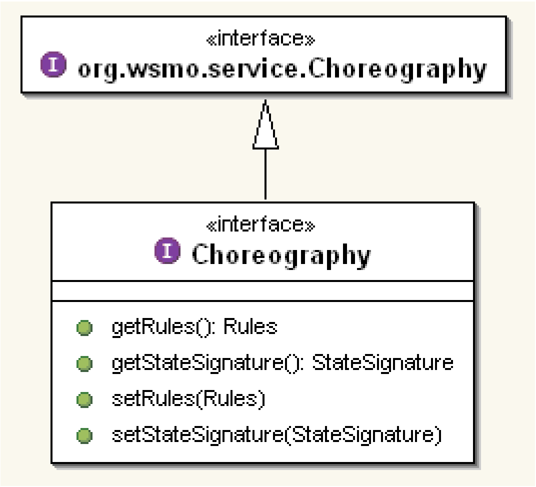

«interface»
org.wsmo.service.Choreography
«interface»
Choreography
getRules(): Rules
getStateSignature(): StateSignature
setRules(Rules)
setStateSignature(StateSignature)

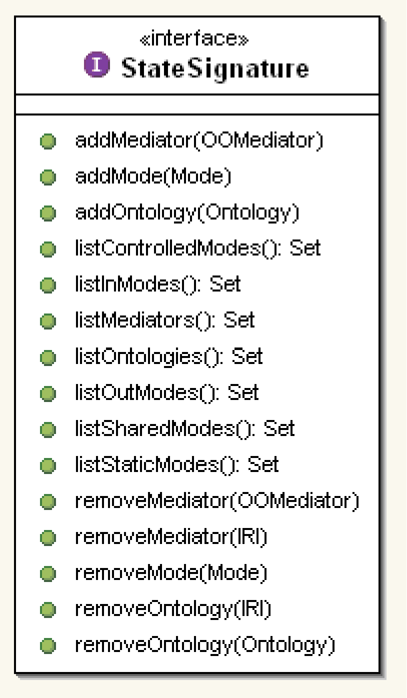
Figure 4.1: *Choreography* interface

4.3 State signature

The *StateSignature*[1] (Figure 4.2) is a container for the *Mode* objects (which define how the ontology instances are interchanged between the client and the web service interface) and the imported ontologies (see [12]).

There are five different mode types, as defined by [10] (see Figure 4.3):

- *Static* – extension of the concept cannot be changed (default mode).
- *In* – extension of the concept or relation can only be changed by the environment and read by the choreography execution; a grounding mechanism for this item, that implements write access for the environment, must be provided
- *Out* – extension of the concept or relation can only be changed by the choreography execution and read by the environment; a grounding mechanism for this item, that implements read access for the environment, must be provided.
- *Shared* – the extension of the concept or relation can be changed and read by the choreography execution and the environment; a grounding mechanism for this item, that implements read/write access for the environment and the service, may be provided
- *Controlled* – the extension of the concept is changed and read only by the choreography execution


«interface»
StateSignature

```
addMediator(OOMediator)
addMode(Mode)
addOntology(Ontology)
listControlledModes(): Set
listInModes(): Set
listMediators(): Set
listOntologies(): Set
listOutModes(): Set
listSharedModes(): Set
listStaticModes(): Set
removeMediator(OOMediator)
removeMediator(IRI)
removeMode(Mode)
removeOntology(IRI)
removeOntology(Ontology)
```

Figure 4.2: *StateSignature* interface

Grounding is upper interface for any grounding descriptions. The current Choreography API supports only *WSDLGrouding*, which defines methods for getting information about IRI pointing to the input/output parameter of some WSDL ([3]).

4.4 Transition rules

Transition rules[2] define a formal algebra to model the changes of the state in the ASM. The web service interface execution is described by a finite set of transition rules, which are executed in parallel by the ASM agent (i.e. their order is not important). Rules express the changes of the state by modifying set of instances (adding, removing and updating instances to the signature ontology).

The available rules are:

- **if** *condition* **then** *rules* **endIf**
- **forall** *variables* **with** *condition* **do** *rules* **endForall**

[1] See also the definition of State Signature in the WSMO specification at http://www.wsmo.org/TR/d14/v0.4/#chorSig

[2] See also the definition of Transition rules in the WSMO specification at http://www.wsmo.org/TR/d14/v0.4/#chorGt

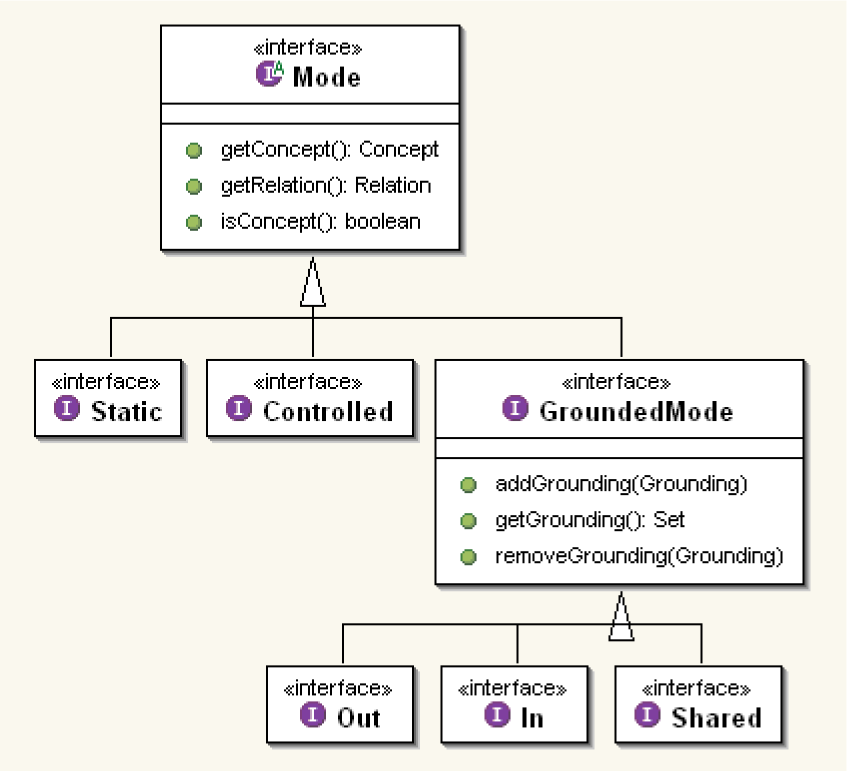


Figure 4.3: Choreography modes

- **choose** *variables* **with** *condition* **do** *rules* **endChoose**
- **add**(*fact*)
- **delete**(*fact*)
- **update**($fact_{old} \rightarrow fact_{new}$), or **update**($fact_{new}$)
- $Rule_1 \mid Rule_2 \mid Rule_3$

The corresponding Java interfaces (*IfThen*, *ForAll*, *Choose*, *Add*, *Delete*, *Update* and *PipedRules* respectively) are presented on Figure 4.4

4.5 Choreography factory

The choreography modelling elements are created by the *ChoreographyFactory*, which implements the *Factory* design pattern [7].

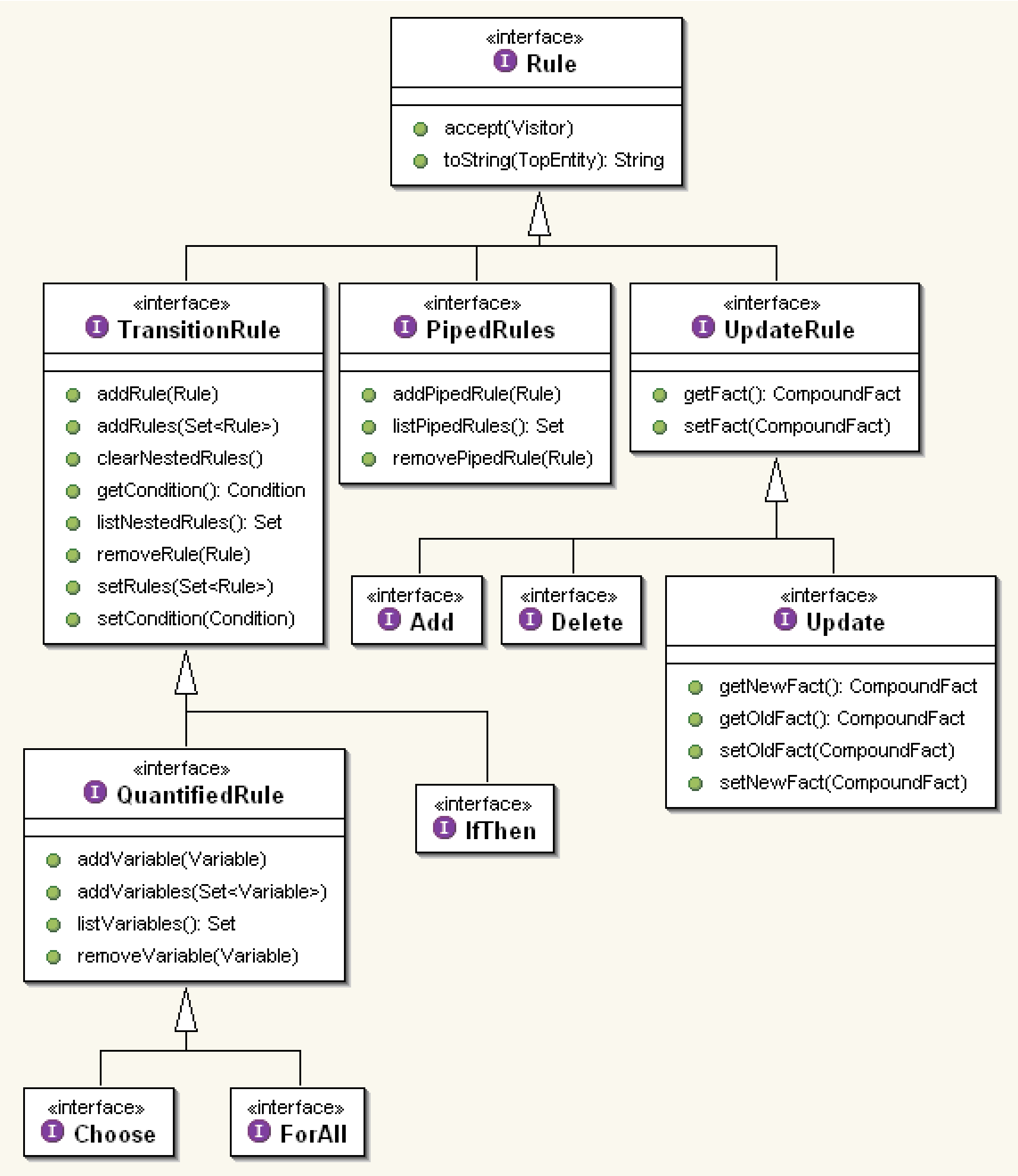


Figure 4.4: Choreography rules

Bibliography

[1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S. W3C Member Submission, November 2005.

[2] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. W3C Recommendation, W3C, October 2004.

[3] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3C Candidate Recommendation, March 2006. Available at http://www.w3.org/TR/wsdl20/.

[4] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. D16.1: WSML family of representation languages. WSML working draft, DERI, October 2005. Available at http://www.wsmo.org/TR/d16/d16.1/v0.3/.

[5] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). IETF RFC3987, IETF, 2005. Available at http://www.ietf.org/rfc/rfc3987.txt.

[6] J. Farrell and H. Lausen. Semantic Annotations for WSDL (SA-WSDL). Technical report, W3C, June 2006.

[7] E. Gamma, R. Helm, . Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1st edition, January 1995.

[8] C. Pedrinaci L. Henocque J. Lemcke, B. Norton. D3.8 ontology for web services choreography and orchestration v2. Dip deliverable, EU IST FP6 507483, June 2006.

[9] D. Roman, H. Lausen, U. Keller, J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, and M. Stollberg. Web Service Modeling Ontology, v1.2. WSMO working draft, DERI, April 2005. Available at http://www.wsmo.org/TR/d2/v1.2/.

[10] D. Roman, J. Scicluna, D. Fensel, A. Polleres, and J. de Bruijn. D14v0.4: Ontology-based choreography and orchestration of WSMO services. WSMO working draft, DERI, May 2006.

[11] N. Steinmetz and H. Lausen. Parser and validator manual. Available online at http://wsmo4j.sourceforge.net/doc/validator.pdf, January 2006.

[12] M. Stollberg, S. Galizia, J. Kopecký, M. Moran, J. Scicluna, A. Polleres, J. Lemcke, L. Henoque, B. Norton, E. Kilgarriff, and M. Kleiner. Dip interface description ontology. Dip deliverable, EU IST FP6 507483, January 2006. Available at http://dip.semanticweb.org/documents/DIO-Annex-to-D3.4-and-D3.5.pdf.

[13] I. Toma and D. Foxvog. D28.4v0.1 Non-functional properties in Web Services. WSMO working draft, DERI, June 2006. Available at http://www.wsmo.org/TR/d28/d28.4/v0.1/.

[14] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Rfc 2413 - Dublin Core metadata for resource discovery. Technical report, September 1998.

Chapter 5

Appendix A – Examples

Examples demonstrating the usage of WSMO API are available online at http://wsmo4j.sourceforge.net/examples.html

Chapter 6

Appendix B – Changelog

Version	**Date**	**Author(s)**	**Changes**
2.01	2006/11/24	marin	first public draft