

The Concepts of WSMO

In the case of complex systems, one cannot make sense of the constituent parts without first understanding the whole. Parts can only make sense in the context of a greater whole.

Our hope is that the previous chapter has provided the reader with some sort of conceptual understanding of WSMO (although only a general overview at best), at least enough to be able to understand the contention that WSMO does not fall prey to the problems of the family of complex systems. Nevertheless, the present chapter will follow the “grandfatherly” advice of complexity theory just the same, attempting to “make sense of the constituent parts”.

In this chapter, we describe further the core elements of WSMO: we present ontologies in Section 6.1, Web services in Section 6.2, goals in Section 6.3, and mediators in Section 6.4. Section 6.5 presents an informal discussion of the nonfunctional properties that are part of the WSMO conceptual model.

6.1 Ontologies

Ontologies were introduced in Chapter 3 and although there are currently several standardizations efforts for ontology languages [60, 32, 64], none of them have the desired expressivity and computational properties that are required to describe Web services at a sufficient level of granularity. In the following, we shall define an epistemological model which is general enough to intuitively capture existing languages. In the next chapter of this book we shall present a concrete language specifically designed to express this metamodel.

Now we present the conceptual model and introduce the elements that constitute an ontology using the MOF notation, defining the class “ontology” with its attributes. The following listing specifies the building blocks for an ontology. Note that all attributes are optional. Thus an ontology can have multiple instances of all elements; however, this is not a requirement.

```

Class ontology
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  usesMediator type ooMediator
  hasConcept type concept
  hasRelation type relation
  hasFunction type function
  hasInstance type instance
  hasAxiom type axiom

```

6.1.1 Nonfunctional Properties

Nonfunctional properties are applicable to all the definitions of WSMO elements. Note that in some cases WSMO specifically recommends that certain properties should be applicable to an element, but this does not impact the general considerations presented here. Nonfunctional properties are mainly used to describe nonfunctional aspects such as the creator and the creation date, and to provide natural-language descriptions, etc. The elements defined by the Dublin Core Metadata Initiative [134] are taken as a starting point. Dublin Core is a set of attributes that define a standard for cross-domain information resource description. In WSMO, we use URIs for identification of elements, which are reused. WSMO proposes several extensions to this set by introducing new attributes, such as the version element which can contain information about the particular version of an element. The set of nonfunctional properties that WSMO recommends are informally presented in Section 6.5.

6.1.2 Imported ontologies

Building an ontology for a particular problem domain can be a rather cumbersome and complex task. One standard way to deal with the complexity is modularization. Imported ontologies allow a modular approach to ontology design and can be used as long as no conflicts need to be resolved between the ontologies. When ontologies are imported, all statements of the imported ontology will be virtually included in the importing ontology. Every WSMO top-level entity may use this import facility to include the logical definition of the vocabulary used.

6.1.3 Mediators

When ontologies are imported in realistic scenarios, some steps are needed for aligning, merging and transforming the imported ontologies in order to resolve ontology mismatches. For this reason and in line with the basic design principles underlying the WSMF, ontology mediators (OO mediators), which are described in detail in Section 6.4, are used when an alignment of an imported ontology is necessary. Such an alignment can be done by simply renaming concepts, attributes or the like. Just like the *importsOntology* statement, the *usesMediator* statement is applicable to all top-level elements, however depending on the element, different mediators may be used.

6.1.4 Concepts

Concepts constitute the basic elements of the agreed terminology for a problem domain. From a high-level perspective, a concept - described by a concept definition - provides attributes with names and types. Specified more formally in MOF, a concept is made up of the following elements:

```

Class concept
  hasNonFunctionalProperties type nonFunctionalProperties
  hasSuperConcept type concept
  hasAttribute type attribute
  hasDefinition type logicalExpression multiplicity = single-valued

```

Furthermore, a concept can be a subconcept of several (or possibly none) direct superconcepts, as specified by an “is-a” relation. In the WSMO model, each *concept* can have a finite number of *concepts* that serve as a superconcept for some other concept. Being a subconcept of some other concept means, in particular, that a concept inherits the signature of this superconcept and the corresponding constraints.

A concept provides a (possibly empty) set of attributes that represent named slots for the data values of instances. An attribute specifies a slot of a concept by fixing the name of the slot and a logical constraint on the possible values filling that slot, which, in a simple case, can be another concept. Hence, this logical expression can be interpreted as a typing constraint.

Additionally, a concept is defined by a logical expression. This means that axioms can be asserted about a concept that refine its meaning, for example with nuances that are not expressible by attributes or an “is-a” hierarchy. A logical expression can be used to refine the semantics of the concept. More precisely, the logical expression defines (or restricts) the extension (i.e. the set of instances) of the concept. More details of the concrete language in which this is done are given in Chapter 7.

6.1.5 Relations

Relations are used in order to model interdependencies between several concepts (or instances of these concepts). The arity of relations is not limited. Specified more formally in MOF, a relation is made up of the following elements:

```

Class relation
  hasNonFunctionalProperties type nonFunctionalProperties
  hasSuperRelation type relation
  hasParameter type parameter
  hasDefinition type logicalExpression multiplicity = single-valued

```

Every *relation* can have a finite set of *relations*, of which a defined relation is declared as being a subrelation. Being a subrelation of some other relation means, in particular, that a relation inherits the signature of this superrelation and the corresponding constraints. Furthermore, the set of tuples belonging to the relation (and the extension of the relation) is a subset of each of the

extensions of the superrelations. In the example given earlier, we can define airline distance as a subrelation of the general distance relation.

Similarly to attributes for concepts, each relation has a possibly empty set of named parameters. If no named parameters are given, a unnamed, ordered list is assumed. Each parameter is single-valued and can have a range restriction in the form of a concept. As for concepts, a logical expression defining the set of instances (n -ary tuples, if n is the arity of the relation) can be specified.

6.1.6 Functions

A function is a special relation, with a unary range and an n -ary domain (the parameters inherited from relations), where the range specifies the return value. Functions can be used to represent and exploit built-in predicates of common datatypes. Their semantics can be captured externally by means of an oracle or the semantics can be formalized by assigning a logical expression to a particular relation. The logical representation of functions is almost the same as for relations. A typical example of the use of oracles is provided by “built-ins”. These are functions such as “round” that are evaluated outside the logical theory.

6.1.7 Instances

Instances are defined either explicitly or by a link to an instance store, i.e. an external storage of instances and their values. Specified more formally in MOF, an instance is made up of the following elements:

```

Class instance
  hasNonFunctionalProperties type nonFunctionalProperties
  hasType type concept
  hasAttributeValues type attributeValue

```

Instances of relations (with arity n) can be seen as n -tuples of instances of the concepts that are specified as the parameters of the relation.

In general, instances do not need to be specified using an explicit notation of the kind used above. In particular, when a huge number of instances exist, a link to a data store can be used [73]. Basically, the approach is to integrate large sets of instances which already exist on storage devices by means of sending queries to external storage devices or oracles.

6.1.8 Axioms

An axiom is a logical expression together with its nonfunctional properties. A detailed discussion of axioms can be found in Chapter 7.

6.2 Web Services

The *Web service* element of WSMO provides a conceptual model (a meta-model in MOF terms) for describing, in an explicit and unified manner, all aspects of a service, including its nonfunctional properties, its functionality, and the interfaces needed to obtain it. An unambiguous model of services with a well-defined semantics can be processed and interpreted by computers without human intervention, enabling the automation of the tasks involved in the usage of Web services, for example discovery, selection, composition, mediation, execution, and monitoring.

As observed in [111], the word “service” can be understood in several different ways, with slightly different meanings: as the provision of value in some domain, as a software entity able to provide something of value, and as a means of interacting online with a service provider.

WSMO provides a unified view of a service; the value that the service can provide is captured by its *capability*, and the means to interact with the service provider to request the actual performance of the service, or to negotiate aspects of its provision, is captured by its *interfaces*. The software entity that provides the service is transparent to us. We are concerned only with its style of interaction and with what other services are used in order to provide the value described in the capability.

Notice that in WSMO, the interaction with a service can be realized by using Web services in the WSDL sense. However, we are not restricted to WSDL for the grounding of services. The elements identified in WSMO are equally applicable to other underlying technologies, such as Triple Space computing (Section 12.4). The directions that we have taken for the grounding to WSDL are described in Section 9.5.

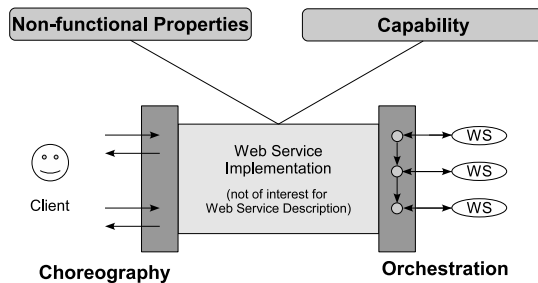


Fig. 6.1. WSMO Web service – general description

The main elements of a service description, depicted in Fig. 6.1, are a capability, describing the value that the service can provide, and one or more interfaces, in which the choreography and the orchestration of the service are described. The choreography specifies how the service achieves its capability

by means of interaction with the user – i.e. the communication with the user of the service. The orchestration specifies how the service achieves its capability by making use of other services – i.e. the coordination of other services.

More precisely, the WSMO Web service element is defined as follows:

```

Class service
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  usesMediator type {ooMediator, wwMediator}
  hasCapability type capability multiplicity = single-valued
  hasInterface type interface

```

- *Nonfunctional properties.* The nonfunctional properties of a Web service are the aspects of the Web service that are not directly related to its functionality. Besides the nonfunctional properties already presented, they consist of Web-service-specific elements such as *accuracy* (the error rate generated by the service), *financial* (the cost-related and charging-related properties of a service [105]), *network-related QoS* (QoS mechanisms operating in the transport network that are independent of the service), *owner* (the person or organization to which the service belongs), *performance* (how fast a service request can be completed), *reliability* (the ability of a service to perform its functions, i.e. to maintain its service quality), *robustness* (the ability of the service to function correctly in the presence of incomplete or invalid inputs), *scalability* (the ability of the service to process more requests in a certain time interval), *security* (the ability of a service to provide authentication, authorization, confidentiality, traceability/auditability, data encryption, and nonrepudiation), *transactional* (the transactional properties of the service), *trust* (the trustworthiness of the service), and *version*. The nonfunctional properties are to be mainly used for the discovery and selection of services; however, they contain information that is also suitable for negotiation.
- *Imported ontologies.* Imported ontologies are used to import the explicit, formal vocabulary used in the specification of a Web service.
- *Use of mediators.* A Web service uses mediators in the following situations. (1) When heterogeneous terminologies are used and conflicts between them arise, a service can import ontologies using ontology mediators, as explained in Section 6.1.3. (2) When the service needs to cope with process and protocol heterogeneity when interacting with other services. In this case, a *wwMediator* is used. For a more detailed description of mediators, see Section 6.4.
- *Capability.* The capability describes the real service provided for example booking of train tickets. A more detailed description of capabilities is given in Section 6.2.1.
- *Interface.* An interface describes the interface of the Web service to be used to achieve the service described. Further details are given in Section 6.2.2.

6.2.1 Capability

The functionality offered by a given service is described by its capability. This is expressed by the state of the world before the service is executed and the state of the world after successful service provision. The capability of a service can be used for discovery and selection purposes, i.e. the capability is used by the requester to determine whether the service meets its needs.

Specified more formally in MOF, a capability is made up of the following elements:

```

Class capability
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  usesMediator type ooMediator
  hasPrecondition type axiom
  hasAssumption type axiom
  hasPostcondition type axiom
  hasEffect type axiom

```

The set of nonfunctional properties that can be attached to a capability is presented in Section 6.5. The attributes of *imported ontologies* and *mediators* play the same role for ontologies.

- *Preconditions.* Preconditions in the description of a capability specify the required state of the information space before the execution of the service, i.e. they specify what information a Web service expects in order to provide the service. Preconditions constrain the set of states of the information space such that any state satisfying these constraints can serve as a valid starting state (in the information space) for executing the service in a defined manner.
- *Assumptions.* Assumptions in the description of a capability describe the state of the world which is assumed before the execution of the service. Otherwise, the successful provision of the service is not guaranteed. As opposed to preconditions, assumptions are not necessarily checked by the service. We make this distinction in order to allow the explicit idea of conditions on the state of the world that are outside the information space.
- *Postconditions.* Postconditions in the description of a capability describe the state of the information space that is guaranteed to be reached after the successful execution of the service; they also describe the relation between the information that is provided to the service and its results.
- *Effects.* Effects in the description of a capability describe the state of the world that is guaranteed to be reached after the the successful execution of the service, i.e. if the preconditions and the assumptions of the service are satisfied.

6.2.2 Interfaces

An interface describes how the functionality of a service can be achieved (i.e. how the capability of a service can be fulfilled) by providing a twofold view

of the operational competence of the service: (1) *choreography* decomposes a capability in terms of interaction with the service, and (2) *orchestration* decomposes a capability in terms of the functionality required from other services.

This distinction reflects the difference between communication and cooperation. The choreography defines how to communicate with the service in order to consume its functionality. The orchestration defines how the overall functionality is achieved by the cooperation of more elementary service providers.

The Web service interface is meant primarily for the description of the behavior of Web Services and is presented in a way that is suitable for software agents to determine the behavior of the service and reason about it. It might be also useful for discovery and selection purposes, and in this description, a connection to existing Web service specifications, for example WSDL, could also be specified.

The definition of an interface is given below:

```
Class interface
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  usesMediator type ooMediator
  hasChoreography type choreography
  hasOrchestration type orchestration
```

Choreography

The choreography interface describes the behavior of the service from the client's point of view; this definition is in accordance to the one given in the W3C Glossary:¹ Web service choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web services, applications, or human beings.

The state-based mechanism for describing WSMO choreography and orchestration interfaces is based on the abstract state machine (ASM) [58] methodology. An ASM is used to abstractly describe the behavior of a service with respect to an invocation instance of that service. We have chosen an ASM model for the description of such interfaces since such a service invocation (e.g. the purchase of a book from Amazon.com) may consist of a number of interaction steps. These interactions can be described by a stateful abstract machine.

ASMs have been chosen as the underlying model for the following three reasons:

- *Minimality.* ASMs provide a minimal set of modeling primitives, i.e. they enforce minimal Ontological commitments. Therefore, they do not intro-

¹ <http://www.w3.org/2003/glossary>.

duce any ad hoc elements that would be questionable if they were to be included into a standard proposal.

- *Maximality.* ASMs are expressive enough to model any aspect around computation.
- *Formality.* ASMs provide a rigid framework to express dynamics.

In the following, we present a short overview of ASMs and then address the core conceptual model for WSMO choreographies.

Basic Abstract State Machines

Abstract state machines, formerly known as evolving algebras [58], provide a means to describe systems in a precise manner using a semantically well-founded mathematical notation. The core principles are the definition of ground models and the design of systems by refinements. Ground models define the requirements and operations of the system, expressed in mathematical form. Refinements allow the expression of the classical divide-and-conquer methodology for system design in a precise notation, which can be used for abstraction, validation, and verification of the system at any given stage in the development process.

Abstract state machines can be divided into two main categories, namely basic ASMs and multiagent ASMs. The former express the behavior of a system within a given environment. Multiagent ASMs express the behavior of the system in terms of multiple entities that are collaborating to achieve a functionality. For the description of the behavior of a single party, we are interested only in basic ASMs.

A basic ASM is defined in terms of a state signature plus a finite set of transition rules, which are executed in parallel. It may involve nondeterministic behavior. Finite state machines can be viewed as a special case of such basic ASMs. The state signature for a classical ASM usually consists simply of a set of static and dynamic functions, which can be locally updated by update rules. Dynamic functions can be classified into four other categories, namely, *controlled*, *monitored* (or *in*), *interaction* (or *shared*), and *out*. Controlled functions are directly updatable by the rules of a machine M only. Thus, they can be neither read nor updated by the environment. Monitored functions can only be updated by the environment and read by the machine M , and hence constitute the externally controlled part of the state. Shared functions can be read and updated by both the environment and the rules of the machine M . Out functions can be updated and read by the environment, but remain unreadable by M . Furthermore, ASMs define *derived functions*. These are functions that are updatable by neither the machine nor the environment but are instead defined in terms of other static and dynamic (and derived) functions.

WSMO Choreography Model

WSMO choreography deals with the interactions of the Web service from the client's perspective. We base the description of the behavior of a single service exposed to its client on the basic ASM model. WSMO choreography interface descriptions inherit the core principles of such ASMs, which can be summarized as follows: (1) they are state-based, (2) they represent a state by a signature, and (3) they model state changes by transition rules that change the values of functions and relations defined by the signature of the algebra (which, in our context, is a nonempty set of ontologies).

In order to define the signature we use a WSMO ontology, i.e. definitions of concepts, their attributes, and relations and axioms over these. Instead of dynamic changes of function values as represented by dynamic functions in ASMs, we allow the dynamic modification of instances and attribute values in the state ontology. Note that the choreography interface describes the interaction with respect to a single instance of the choreography. The key extension compared with the basic ASMs described above is that the machine signature is defined in terms of a WSMO ontology (or possibly more than one), and the logical language used for expressing conditions is WSML. This leads us to the notion of evolving ontologies (derived from the notion of evolving algebras, the name initially used for ASMs), since the choreography ASM in fact changes the values of concepts and relations within ontologies.

Taking the ASM methodology as a starting point, a WSMO choreography consists of three elements, which are defined as follows:

```

Class choreography
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasTransitionRules type transitionRules

```

The set of *nonfunctional properties* that can be attached to a capability is presented in Section 6.5. The *state signature* defines the state ontology used by the service, together with the definition of the types of modes that the concepts and relations may have.

State Signature

The signature of the machine is defined by (1) importing an ontology which defines the state signature over which the transition rules are executed, (2) an optional set of OO mediators if the imported state ontologies are heterogenous, (3) a set of statements defining the modes of the concepts and relations, and (4) a set of update functions. The default mode for concepts of the imported ontologies which are not listed explicitly in the mode statements is static. Note it is not allowed to assign either “in” or “out” to concepts which have explicitly defined any instance data in the imported ontologies by the state signature. The definition of the state signature is expressed as follows:

```

Class stateSignature
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type ooMediator
  hasStatic type mode
  hasIn type mode
  hasOut type mode
  hasShared type mode
  hasControlled type mode

Class mode sub—Class {concept, relation}
  hasGrounding type grounding

```

The state for a given signature of a WSMO choreography is defined by all legal WSMO identifiers, concepts, relations, and axioms. The elements that can change and are used to express different states of a choreography, are instances of concepts and relations, which are used similarly to locations in ASMs. These changes are expressed in terms of the creation of new instances or changes of attribute values.

In a similar way to the classification of locations and functions in ASMs, the concepts and relations of an ontology are marked to support a particular role (or mode). These roles are of five different types: static, in, out, shared, and controlled.

Since Web services deal with actual instance data, the classification is inherited by instances of the correspondingly classified concepts and relations. That is, instances of controlled concepts and relations can only be created and modified by the choreography interface; instances of “in” concepts can only be read by the choreography interface; instances of “out” concepts can only be created by the choreography interface, and cannot be read or modified after their creation. Instances of shared concepts and relations can be read and written by both the choreography interface and, possibly, the environment; they can also be modified after creation. In the future, we expect shared concepts to be particularly important for grounding alternatives to WSDL which do not rely on strict message passing, such as semantically enabled Tuple Spaces [39].

Transition Rules

Unlike basic ASMs, the most basic forms of rules are not assignments, but instead deal with basic operations on instance data, for example adding instances to the signature ontology, removing instances, and updating instances. To this end, we define atomic update functions to add, delete, and update instances. This in turn allows us not only to add and remove instances to/from concepts and relations, but also to add and remove attribute values for particular instances. In WSMO choreography, these basic updates are defined as a set of fact modifiers, which are of four different types:

- `add(fact);`
- `delete(fact);`

- `update(factnew);`
- `update(factold \rightarrow factnew).`

More complex transition rules are defined recursively, analogously to classical ASMs, by “if-then”, “forall” and “choose” rules:

```

if Condition then Rules endif
forall Variables with Condition do Rules endforall
choose Variables with Condition do Rules endchoose

```

Orchestration

Orchestration describes how a service makes use of other services in order to achieve its capability. In many real scenarios, a service is provided by using and interacting with services provided by other applications or businesses. For example, the booking of a trip might involve the use of another service for validating a credit card and charging it with the correspondent amount, and the user of the booking service may want to know which other business organizations they are implicitly going to deal with.

WSMO introduces the orchestration element into the description of a service to reflect such dependencies. WSMO orchestration allows the use of statically or dynamically selected services. In the former case, a concrete service will be selected at design time. In the latter case, the service will only describe the goal that has to be fulfilled in order to provide the service. This goal will be used to select at runtime an available service that fulfills it, and the service user could influence this choice.

It is envisioned that orchestration should make use of the multiagent asynchronous ASM model to describe the interactions between Web services and goals. These aspects are still to be investigated further, and will be defined in future publications.

6.3 Goals

Goals are used in WSMO to describe a user’s desires. They provide the means to specify the requester-side objectives when a Web service is consulted, describing at a high level a concrete task to be achieved.

Goals are representations of objectives for which fulfillment is sought through the execution of Web services. They can be descriptions of services that would potentially satisfy the user’s desires.

Note that WSMO completely decouples the objectives that a requester has, i.e. his/her goal, from the services that can actually fulfill such a goal. Specified more formally in MOF, a goal is made up of the following elements:

```

Class goal
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  usesMediator type {ooMediator, ggMediator}
  requestsCapability type capability multiplicity = single-valued
  requestsInterface type interface

```

- *Nonfunctional properties.* Given the fact that a goal can represent a service that would potentially satisfy the user's desires, the set of nonfunctional properties that can be attached to a goal is similar to that attached to a Web services (see Section 6.2). An extra nonfunctional property can be attached to a goal, namely *type of match*, which represents the type of match desired for a particular goal (under the assumption of set-based modeling, this can be an exact match, a match where the goal description is a subset of the Web service description or a match where the Web Service description is a subset of the goal description). For a detailed discussion, see Section 9.2.
- *Imported ontologies.* A goal uses imported ontologies as a terminology to define the other elements that are part of the goal, as long as no conflicts need to be resolved.
- *Nonfunctional properties.* A goal uses mediators in the following situations. (1) When heterogeneous terminologies are used, conflicts between them may arise; in these cases, a service can import ontologies using ontology mediators (*OO mediators*). (2) When a goal reuses already existing goals, for example by refining them, *GG mediators* are used (these are explained in more detail in Section 6.4).
- *Requested Capability.* The requested capability in the definition of a goal describes the capability of the services that the user would like to have.
- *Interface.* The interface in the definition of a goal describes the interface of the service that the user would like to have and interact with.

6.4 Mediators

Mediation is concerned with handling heterogeneity, i.e. resolving possible mismatches between resources that ought to be interoperable. Heterogeneity naturally arises in open and distributed environments, and thus in the application areas of Semantic Web services, WSMO defines the concept of mediators as a top-level concept.

Mediator-orientated architectures, as introduced in [135], specify a mediator as an entity for establishing interoperability of resources that are not compatible prior to resolving mismatches between them at run time. The approach to mediation that is aspired to relies on declarative description of resources, whereupon mechanisms for resolving mismatches work on a structural and semantic level. This allows the defining of generic, domain-independent mediation facilities and the reuse of mediators. Concerning the need for mediation within Semantic Web services, WSMO identifies three levels of mediation:

1. **Data-level mediation:** mediation between heterogeneous data sources and transfer protocols; within ontology-based frameworks such as WSMO, this is mainly concerned with ontology integration.
2. **Functional-level mediation:** mediation between heterogeneous functionalities requested and provided; in WSMO, this relates to capability descriptions of goals and Web services that are similar but do not match precisely.
3. **Process-level mediation:** mediation between heterogeneous communication protocols and business processes; in WSMO, this relates mainly to choreographies of Web services that ought to interact with the mismatch handling on the business logic level of Web services (which is related to the orchestration of Web Services).

WSMO mediators realize a mediation-orientated architecture for Semantic Web services, thereby providing an infrastructure for handling heterogeneities that may possibly arise between WSMO components and realizing the design concepts of strong decoupling and strong mediation. We explain the WSMO mediation architecture in further detail in Section 9.3.

6.4.1 Description of a Mediator

A WSMO mediator connects WSMO components and resolves mismatches between them. The following gives the general definition:

```

Class mediator
  hasNonFunctionalProperty type nonFunctionalProperty
  importsOntology type ontology
  hasSource type {ontology, goal, webService, mediator}
  hasTarget type {ontology, goal, webService, mediator}
  hasMediationService type {webService, goal, wwMediator}

```

- *Nonfunctional properties.* As a mediator can be provided as a service, the same nonfunctional properties as for Web services are used (see Section 6.2 for what these nonfunctional properties consist of).
- *Imported ontologies.* Imported ontologies are used to import the explicit, formal vocabulary used in the specification of a mediator; they play the same role as in the case of ontologies or any other element defined by WSMO.
- *Source.* The source component of a mediator defines the resources for which heterogeneities are resolved; a mediator can have several source components.
- *Target.* The target component of a mediator is the component that receives the mediated source components.
- *Mediation service.* The mediation service defines the mediation facility applied for resolving mismatches. This can be defined in various ways: directly (i.e. explicitly linking to a mediation service); via a goal that

specifies the desired mediation facility which is then detected by a discovery mechanism; or via another mediator when a mediation service is to be used that is not interoperable with the first mediator.

6.4.2 WSMO Mediator Types

In order to allow the resolving of heterogeneities between different WSMO components, WSMO defines several different types of mediators for connecting different WSMO components and overcoming the heterogeneities that can arise between those components: OO mediators, GG mediators, WG mediators, and WW mediators. All mediators are subclasses of the general WSMO mediator class defined above, where the prefix indicates the components connected by the particular type of mediator. The following explains the various WSMO mediator types.

OO Mediators

OO mediators resolve mismatches between ontologies and provide mediated domain knowledge specifications to the target component. The source components are ontologies or other OO mediators that are heterogeneous and able to be integrated, and the target component is any WSMO top-level concept that applies to the integrated ontologies. The following shows the description specialization of an OO mediator:

```
Class ooMediator sub—Class mediator
  hasSource type {ontology, ooMediator}
```

OO mediators are used to import the terminology required for a resource description whenever there is a mismatch between the ontologies to be used. The mediation technique used by OO mediators is mainly ontology integration, i.e. merging, aligning, and mapping ontology definitions in order to retrieve integrated, homogeneous terminology definitions.

GG Mediators

A GG mediator connects goals, allowing one to create a new goal from existing goals and thus defining goal ontologies. GG mediators are defined as follows:

```
Class ggMediator sub—Class mediator
  usesMediator type ooMediator
  hasSource type {goal, ggMediator}
  hasTarget type {goal, ggMediator}
```

A GG mediator might use an OO mediator to resolve terminology mismatches between the source goals. Mediation services for GG mediators reduce or combine the descriptions of the source goals into the newly created target goal.

WG Mediators

A WG mediator links a Web service to a goal, resolves terminological mismatches, and states the functional difference (if any) between the two. WG mediators are defined as follows:

```
Class wgMediator sub—Class mediator
  usesMediator type ooMediator
  hasSource type { service , wgMediator }
  hasTarget type { goal , ggMediator }
```

WG mediators are used to prelink services to existing goals, and for the handling of partial matches within Web service discovery. As with GG mediators, WG mediators can be applied for resolving terminological mismatches.

WW Mediators

A WW mediator is used to establish interoperability between Web services that are otherwise not interoperable. The definition in MOF is as follows:

```
Class wwMediator sub—Class mediator
  usesMediator type ooMediator
  hasSource type { service , wwMediator }
  hasTarget type { service , wwMediator }
```

A WW mediator mediates between the choreographies of Web services that ought to interact with one another, where mediation might be required on the data, protocol, and process levels. As with the other WSMO mediator types, OO mediators can be applied for resolving terminological mismatches.

6.5 Nonfunctional Properties

Nonfunctional properties are used in the definition of WSMO elements. Which nonfunctional properties apply to which WSMO element is specified in the description of each WSMO element. In the following, we informally describe the nonfunctional properties recommended to be attached to WSMO elements.

- *Accuracy*. This represents the error rate generated by a Web service. It can be measured by the numbers of errors generated in a certain time interval.
- *Contributor*. An entity responsible for making contributions to the content of the element. Examples of the property `dc:contributor` include a person, an organization, and a Web service. The Dublin Core specification recommends that typically, the name of a `dc:contributor` should be used to indicate the entity.²

² In order to point unambiguously to a specific resource, we recommend the use of an instance of `foaf:Agent` as the value type [21].

- *Coverage*. The extent or scope of the content of the element. Typically, dc:coverage will include spatial location (a place name or geographic coordinates), a temporal period (a period label, date, or date range), or a jurisdiction (such as a named administrative entity).³
- *Creator*. The entity primarily responsible for creating the content of the element. Examples of dc:creator include a person, an organization, and a Web service. The Dublin Core specification recommends that, typically, the name of a dc:creator should be used to indicate the entity.
- *Date*. The date of an event in the life cycle of the element. Typically, dc:date will be associated with the creation or availability of the element.⁴
- *Description*. An account of the content of the element. Examples of dc:description include, but are not limited to, an abstract, a table of contents, a reference to a graphical representation of the content and a free-text account of the content.
- *Financial*. This represents the cost-related and charging-related properties of a Web service [105]. This property is a complex property, which includes charging styles (e.g. per request or delivery, per unit of measure, granularity, etc.), aspects of settlement such as the settlement model (transactional vs. rental) and a settlement contract, payment obligations, and payment instruments.
- *Format*. The physical or digital manifestation of the element. Typically, dc:format may include the media type or dimensions of the element. The format may be used to identify the software, hardware, or other equipment needed to display or operate the element. Examples of dimensions include size and duration.
- *Identifier*. An unambiguous reference to the element within a given context. Recommended best practice is to identify the element by means of a string or number conforming to a formal identification system. In Dublin Core, formal identification systems include, but are not limited to, the Uniform Resource Identifier (URI) (including the Uniform Resource Locator (URL)), the Digital Object Identifier (DOI), and the International Standard Book Number (ISBN).
- *Language*. The language of the intellectual content of the element. The language tags are defined in ISO Standard 639 [43], for example “en-GB”. In addition, the logical language used to express the content should be mentioned; for example this could be OWL.

³ For more complex applications, consideration should be given to using an encoding scheme that supports an appropriate specification of information, such as DCMI Period, DCMI Box, or DCMI Point.

⁴ We recommend using an encoding defined in ISO Standard 8601:2000 [44] for date and time notation. A short introduction to the standard can be found at <http://www.cl.cam.ac.uk/~mgk25/iso-time.html>. This standard is also used by the XML Schema definition (YYYY-MM-DD) [16], and thus one is automatically compliant with XML Schema too.

- *Network-related QoS.* This represents the QoS mechanisms operating in the transport network that are independent of the Web service. They can be measured by network delay, delay variation, and/or message loss.
- *Owner.* A person or organization to which a particular WSMO element belongs.
- *Performance.* This represents how fast a Web service request can be completed. According to [112], performance can be measured in terms of throughput, latency, execution time, and transaction time. The response time of a Web service can also be a measure of the performance. High-quality Web services should provide higher throughput, lower latency, lower execution time, faster transaction time and faster response time.
- *Publisher.* The entity responsible for making the element available. Examples of dc:publisher include a person, an organization, and a Web service. The Dublin Core specification recommends that, typically, the name of a dc:publisher should be used to indicate the entity.
- *Relation.* A reference to a related element. Recommended best practice is to identify the referenced element by means of a string or number conforming to a formal identification system.
- *Reliability.* This represents the ability of a Web service to perform its functions (that is, to maintain its Web service quality). It can be measured by the number of failures of the Web Service in a certain time interval.
- *Rights.* Information about rights held in and over the element. Typically, dc:rights will contain a rights management statement for the element, or reference a Web service providing such information. Rights information often encompasses intellectual property rights (IPR), copyright, and various property rights. If the rights element is absent, no assumptions may be made about any rights held in or over the element.
- *Robustness.* This represents the ability of the Web service to function correctly in the presence of incomplete or invalid inputs. It can be measured by the number of incomplete or invalid inputs for which the Web service still functions correctly.
- *Scalability.* This represents the ability of the Web service to process multiple requests in a certain time interval. It can be measured by the number of requests resolved in a certain time interval.
- *Security.* This represents the ability of a Web service to provide authentication (entities (users or other Web services) that can access a Web service, and their data, should be authenticated), authorization (entities should be authorized so that only they can access the protected Web services), confidentiality (data should be treated properly so that only authorized entities can access or modify the data), traceability/auditability (it should be possible to trace the history of a Web service after a request has been serviced), data encryption (data should be encrypted), and nonrepudiation (an entity cannot deny requesting a Web service or data after the fact).
- *Source.* A reference to an element from which the present element is derived. The present element may be derived from an element referenced in

dc:source in whole or in part. Recommended best practice is to identify the referenced element by means of a string or number conforming to a formal identification system.

- *Subject*. A topic of the content of the element. Typically, dc:subject will be expressed as keywords, key phrases, or classification codes that describe a topic of the element. Recommended best practice is to select a value from a controlled vocabulary or a formal classification scheme.
- *Title*. The name given to an element. Typically, dc:title will be a name by which the element is formally known.
- *Transactional*. This represents the transactional properties of the Web service.
- *Trust*. This represents the trustworthiness of a Web service or an ontology.
- *Type*. The nature or genre of the content of the element. The dc:type includes terms describing general categories, functions, genres, or aggregation levels of content.
- *Version*. As many properties of an element might change over time, an identifier of the element at a certain moment in time is needed.

6.6 Summary

In this chapter we have exhaustively introduced all elements relevant to the WSMO framework. We have done this in a semiformal fashion using the UML-based Meta-Object Facility. By separating the model from the language used to implement it, this provides significantly more freedom than other competing approaches (see Chapter 8).

Among the guiding principles we have presented Web compliance, which will be very prominent in the next chapter, which introduces the concrete language for WSML. Aspects of the Semantic Web are represented mainly by introducing an epistemological model for ontologies and by using elements of this model for refining other aspects such as capabilities through logical expressions.

We have not yet made the relationship between the WSMO model and Web service technology in the sense of WSDL explicit. The reason is that (1) WSMO, used as a conceptual model to describe services, is independent of the concrete technology used to implement them, and (2) this relationship (grounding) is, in our context, only relevant to actual Web service execution and not essential for an overall understanding. Information about grounding can be found in Section 9.5.

Enabling Semantic Web Services

The Web Service Modeling Ontology

Fensel, D.; Lausen, H.; Polleres, A.; de Bruijn, J.;

Stollberg, M.; Roman, D.; Domingue, J.

2007, XIV, 188 p., Hardcover

ISBN: 978-3-540-34519-0