A Formal Semantic Model of the Semantic Web Service Ontology (WSMO)

Hai H. Wang Nick Gibbins Terry Payne Ahmed Saleh
University of Southampton
{hw, nmg, trp, amms}@ecs.soton.ac.uk

Jun Sun
National University of Singapore
sunj@comp.nus.edu.sg

Abstract

Semantic Web Services, one of the most significant research areas within the Semantic Web vision, has attracted increasing attention from both the research community and industry. The Web Service Modelling Ontology (WSMO) has recently been proposed as an enabling framework for the total/partial automation of the tasks (e.g., discovery, selection, composition, mediation, execution, monitoring, etc.) involved in both intra- and inter-enterprise integration of Web Services. To support the standardization and tool support of WSMO, a formal semantics of the language is highly desirable. As there are a few variants of WSMO and it is still under development, the semantics of WSMO needs to be formally defined to facilitate easy reuse and future development. In this paper, we present a formal Object-Z semantics of WSMO. Different aspects of the language have been precisely defined within one unified framework. This model not only provides a formal unambiguous model which can be used to develop tools and facilitate future development, but as demonstrated in this paper, can be used to identify and eliminate errors presented in existing documentation.

1 Introduction

The next generation of the Web, the Semantic Web (SW) [2] provides computer-interpretable markup of both content and services on the Web, thus enabling automation of many tasks currently performed by humans. Among the most important Web resources are those that provide services. Web services are Web-accessible programs that proliferate the Web by providing user access to applications supporting tasks such as e-commerce, entertainment, etc. Semantic Web Service research [14] has attracted more and more attention from both research communities and industries, and several different approaches have been studied to achieve the vision of Semantic Web service [1, 17]. The Web Service Modelling Ontology (WSMO), one of the most significant Semantic Web Service framework proposed to date [17], complements the existing syntactic Web service standards by providing a conceptual model and language for the semantic markup of all relevant aspects of general services, which are accessible through a Web service interface. The ultimate goal of such markup is to enable the total/partial automation of the tasks (e.g., discovery, selection, composition, mediation, execution, monitoring, etc.) involved in both intra- and inter-enterprise integration of Web Services.

The syntax and semantics of WSMO are defined in terms of its metamodel. The language has been described from three different aspects: syntax, static semantics and dynamic semantics. One of the major problems with the current WSMO definition is that the three aspects of WSMO have been separately described in various formats (mainly in natural language, i.e., English, complemented with some XML schemas and simple axioms). These different descriptions contain redundancy and sometimes contradiction in the information provided. Furthermore, with the continuous evolution of WSMO it has been very difficult to consistently extend and revise these descriptions. More importantly, the use of natural language is ambiguous and can be interpreted in different ways. This lack of precision in defining the semantics of WSMO can result in different users, Web service providers and tool developers having different understandings of the same WSMO model. To support common understanding and facilitate standardization[1] and tool development for WSMO, a formal semantics of its language is highly desirable. Also, being a relatively young field, research into Semantic Web services and WSMO is still ongoing, and therefore a semantic representation of WSMO needs to be reusable and extendable in a way that can accommodate this evolutionary process.

The aim of our work is to define a complete formal denotational semantics of the WSMO language using Object-Z (OZ) [10] . A denotational approach has been proved to be one of the most effective ways to define the semantics of a language, and has been used to give formal semantics for many programming and modeling languages [13, 20]. Object-Z has been used to provide one single formal model

[1]http://www.w3.org/Submission/WSMO

for the syntax, the static semantics and the dynamic semantics of WSMO. Also, because these different aspects have been described within a single framework, the consistency between these aspects can be easily maintained. In this paper, we focus on the formal model for the syntax and static semantics of WSMO. The dynamic semantics of WSMO will be discussed in a future paper.

Object-Z (OZ) [10] is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. We chose Object-Z over other formalisms to specify WSMO because:

- The object-oriented modelling style adopted by Object-Z has good support for modularity and reusability.
- The semantics of Object-Z itself is well studied. The denotational semantics [12] and axiomatic semantics [18] of Object-Z are closely related to Z standard work [21]. Object-Z also has a fully abstract semantics [19].
- Object-Z provides some handy constructs, such as *Class-union* [5] etc., to define the polymorphic and recursive nature of language constructs effectively. Z has previously been used to specify the Web Service Definition Language (WSDL) [4]; however, as Z lacks the object-oriented constructs found in OZ, a significant portion of the resulting model focused on solving several low level modeling issues, such as the usage of free types, rather than the WSDL language itself. Thus, using OZ can greatly simplify the model, and hence avoid users from being distracted by the formalisms itself rather than focusing on the resulting model.

The paper is organized as follows. Section 2 briefly introduces the notion of WSMO and Object-Z. Section 3 is devoted to a formal Object-Z model of WSMO syntax and static semantics. Section 4 discusses some of the benefits of this formal model. Section 5 concludes the paper and discusses possible future work.

2 Overview

2.1 WSMO

The Web Service Modelling Ontology (WSMO) [17] is one of the major approaches for modeling services semantically, based on the earlier work on Unified Problem Solving Method, which was part of a "...framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components..."[11]. WSMO provides a framework for semantic descriptions of Web Services and acts as a meta-model for such Services based on the Meta Object Facility (MOF) [15]. Semantic service descriptions, according to the WSMO meta model, can be defined using one of several formal languages defined by WSML (Web Service Modelling Language) [3], and consists of four core elements deemed necessary to support Semantic Web services: *Ontologies*, *Goals*, *Web Services* and *Mediators*. *Ontologies* are described in WSMO at a meta-level. A meta-ontology supports the description of all the aspects of the ontologies that provide the terminology for the other WSMO elements. *Goals* are defined in WSMO as the objectives that a client may have when consulting a Web service. *Web Services* provide a semantic description of services on the web, including their functional and non-functional properties, as well as other aspects relevant to their interoperation. *Mediators* in WSMO are special elements used to link heterogeneous components involved in the modelling of a Web service. They define the necessary mappings, transformations and reductions between linked elements.

2.2 Object-Z (OZ)

Object-Z [10] is an extension of the Z formal specification language to accommodate object orientation. The essential extension to Z in Object-Z is the *class* construct, which groups the definition of a state schema with the definitions of its associated operations. A class is a template for *objects* of that class: the states of each object are instances of the state schema of the class, and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means that no attribute changes value. The standard behavioral interpretation of Object-Z objects is as a transition system [19]. A behavior of a transition system consists of a series of state transitions each effected by one of the class operations.

3 Formal Object Model of WSMO

3.1 OZ Approach to WSMO Semantics

The existing specification of WSMO informally or semi-formally describes the language from three different aspects – syntax (a WSMO model is well-formed), static semantics (a WSMO model is meaningful) and dynamic semantics (how is a WSMO model interpreted and executed). We propose the use of Object-Z to provide a formal specification of all aspects of WSMO in one single unified frame-

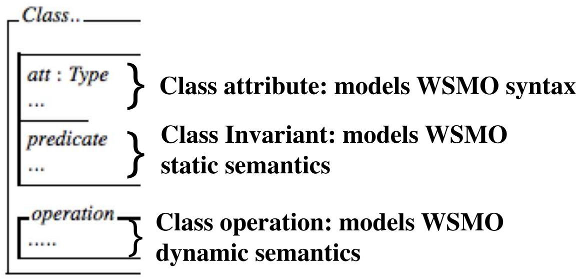
Class..
att : Type
...
Class attribute: models WSMO syntax
predicate
...
Class Invariant: models WSMO static semantics
operation
.....
Class operation: models WSMO dynamic semantics

Figure 1. The framework

work, so that the semantics of the language can be more consistently defined and revised as the language evolves. Figure 1 shows the general approach of the framework. The WSMO elements are modeled as different Object-Z classes. The syntax of the language is captured by the attributes of an Object-Z class. The predicates are defined as *class invariant* used to capture the static semantics of the language. The class operations are used to define WSMO's dynamic semantics, which describe how the state of a Web service changes. This paper focuses on the first two aspects of WSMO, i.e. the formal model of syntax and static semantics[2]. Because of the limited space, we only present a partial model here[3]. Our model is based on the latest version of WSMO (D2v1.3).

3.2 Modeling identifiers, WSMO elements and annotations

```
ID
```

$$AID, URIID, VID : \mathbb{P}\, ID$$

$$\begin{array}{l} VID \cap AID = \varnothing \\ VID \cap URIID = \varnothing \\ URIID \cap AID = \varnothing \\ VID \cup AID \cup URIID = ID \end{array}$$

Every WSMO element is identified by an identifier that can either be classified as URI reference or anonymous ID. Furthermore, WSMO can also identify variables. We use the Object-Z class *ID* to denote all possible identifiers. Note that rather than modeling the identifiers as a Z given type ([*ID*]) (similar to the Z specification of WSDL[4]), modeling them as a class allows us to further extend it and apply various Object-Z class modifiers. *URIID*, *AID* and *VID* are disjoint subsets of *ID* representing the URI reference, anonymous ID and variable ID. Due to the limited space, we only provide an abstract view of the class *ID* without any attributes. These concepts can be modeled in more details, e.g., a *URIID* reference can be expressed by a qualified name, etc.

[2]The dynamic semantics will be addressed in a separate paper.

[3]A more complete model can be found at `http://www.ecs.soton.ac.uk/~hw/WSMO-OZ.pdf`.

WSMO refers to the concepts it defines as "elements", which are modeled as *WSMOElement*.

WSMOElement

$$\begin{array}{l} id : ID \\ hasAnnotation : Annotation \end{array}$$

Annotation

$$\begin{array}{l} hasContributor : \mathbb{P}\, DC\_Contributor \\ hasDate : \mathbb{P}\, DC\_Date;\ \ldots \end{array}$$

Each *WSMOElement* has one ID and optionally a set of *annotations*. *Annotation*, being modeled as an Object-Z class, is used in the definition of WSMO elements. It contains different annotation values which can be applied to any WSMO element, such as *DC_Contributor*[4], *DC_Date*, etc. These values are also defined as Object-Z classes, but they are not shown in this paper. The WSMO specification does not define any cardinality constraints on the number of annotation values an element can have. For example, a WSMO element can have more than one creation date. We model this by specifying that the value of attribute *hasDate* is a set of *DC_Date* values. The tool developers have the freedom to extend the model and add extra constraints, e.g., by adding the predicate '$\#hasDate \leq 1$' to ensure that a WSMO element can only have at most one creation date.

The elements defined within WSMO models can be divided into two groups – top level elements (*TopLevelComponent*) and nested element (*NestedComponent*). WSMO has four kinds of top level elements as the main concepts to describe Semantic Web services. They are *Ontology*, *Serivce*, *Goal* and *Mediator*. Each of which is modeled as a subclass of *WSMOElement*. They will be described in detail in following subsections.

$$TopLevelComponent \mathrel{\widehat{=}} Ontology \cup Goal \cup Service \cup Mediator$$

Essentially, nested elements are attached to some other WSMO elements. In our model, *NestedComponent* denotes all possible nested elements. A nested component has attribute *parentID* that refers to the WSMO element it is attached to. A WSMO element can not be attached to variables or to itself.

NestedComponent

WSMOElement

$$parentID : ID$$

$$parentID \notin VID \wedge id \neq parentID$$

[4]DC_ stands for the Dublin Core.

3.3 Top level element – Ontologies

An ontology is a formal specification of a conceptualization. In WSMO, Ontologies are one of the key elements and they provide the terminology used by other WSMO elements to describe the relevant aspects of the domains of discourse. Ontologies conceptualize a problem domain by defining a set of *concepts*, *relations*, *instances* and some *axioms*. *Ontology* will be formally defined later in this subsection.

3.3.1 Ontology elements

OntologyElement

$NestedComponent[inOntologyID/parentID]$

$$\begin{array}{l}
\Delta \\
inOntology : Ontology \\
\hline
\exists_1\, o : Ontology \bullet inOntologyID = o.id \\
\quad \wedge\, inOntology = o
\end{array}$$

The class *OntologyElement* denotes all the possible WSMO elements defined within Ontologies. It is defined as a subclass of *NestedComponent*. We rename the attribute *parentID* to *inOntologyID* for clarity reasons and also define a secondary attribute [8] *inOntology* to denote the Ontology which *inOntologyID* refers to. The invariant shows that there exists one and only one Ontology given an *inOntologyID*.

3.3.2 Concepts

Concepts constitute the basic elements of the agreed terminology for some problem domains. *Concept* is derived from *OntologyElement*. In it, *hasSuperConcept* attribute denotes the super-concepts of a concept. *hasAttribute* and *hasInstance* denote the *attributes* and *instances* explicitly defined for a concept, while the secondary attribute *totalSuperConcept* denotes all the ancestor concepts, *totalAttr* denotes the total attributes – explicitly declared in a concept and implicitly inherited, and *totalIns* denotes the instances of a concept and its super-concepts. *hasDefinition* denotes the logical expression used to define the semantics of a concept (the logical expression used by WSMO is formally defined in Section 3.7).

The class invariant of *Concept* also specifies that:

- all the super concepts must be defined (directly or indirectly) within *Ontology*;
- the concept belongs to the *Ontology* it is attached to;
- all instances and attributes contained by a concept must belong to the concept;
- the definition of a concept must be one of the forms of $\Rightarrow$, $\Leftarrow$ or $\Leftrightarrow$ implication. The left hand side of the implication must be an expression with the form of '*memberOf C*' where *C* must be the defined concept. The terms used in the definition must be well defined. The left-hand side and right-hand side of the expression has only one common free variable. The terms used in the expression must be defined in *Ontology*;
- a concept inherits the attributes of this superconcepts.

The paper only shows part of the model.

Concept

OntologyElement

$$\begin{array}{l}
hasSuperConcept : \mathbb{P}\, Concept;\ hasAttribute : \mathbb{P}\, Attribute \\
hasInstance : \mathbb{P}\, Instance;\ hasDefinition : Expression \\
\Delta \\
totalAttr : \mathbb{P}\, Attribute;\ totalIns : \mathbb{P}\, Instance \\
totalSuperConcept : \mathbb{P}\, Concept \\
\hline
hasSuperConcept \subseteq inOntology.totalConcept \\
self \in inOntology.hasConcept \\
\forall\, i : hasInstance \bullet self \in i.type \\
\forall\, a : hasAttribute \bullet self = a.inConcept \\
hasDefinition \in LeftImpExp \cup RightImpExp \\
\qquad\qquad \cup DualmpExp \\
hasDefinition.left \in MemberOfExp \\
hasDefinition.left.con = self \\
hasDefinition.left.hasVariable \\
\quad = hasDefinition.right.hasVariable \\
\#hasDefinition.right.hasVariable = 1 \\
hasDefinition.usedTerm \subseteq inOntology.totalterm \\
totalAttr = hasAttribute \cup \\
\quad \bigcup\{s : hasSuperConcept \bullet s.totalAttr\} \\
totalSuperConcept = \ldots
\end{array}$$

Attribute

ConceptElement

$typeModel ::= ofType \mid impliesType$

$$\begin{array}{l}
hasType : Concept;\ hasTypeModel : typeModel \\
\hline
hasType \in inConcept.inOntology.totalConcept
\end{array}$$

The elements defined within *Concept* are defined as *ConceptElement*. It is derived from *NestedComponent* and the attribute *parentID* is renamed to *InConceptID* and a secondary attribute *inConcept*(The definition is omitted here). *Attribute*, defined for each concept, represents a named slot for data values for instances, whereas *hasType* denotes the possible values of that slot. The class invariant specifies that all the value types must be defined within *Ontology*.

3.3.3 Instances

WSMO instances are modeled as *Instance*. *hasType* denotes the explicitly asserted concepts of which the instance is an instance, while *totalType* denotes all asserted and inferred type concepts. The complete definition of *Instance* is omitted due to the space limitation.

$$
\begin{array}{l}
\textit{Instance} \\
\textit{OntologyElement} \\
\hline
\textit{hasType} : \mathbb{P}\,\textit{Concept} \\
\textit{hasAttributeValues} : \mathbb{P}\,\textit{AtttribueValue} \\
\Delta \\
\textit{totalType} : \mathbb{P}\,\textit{Concept} \\
\hline
\ldots
\end{array}
$$

The *Relation* and *RelationInstance* of *Ontology* can be similarly defined.

3.3.4 Ontologies

$$
\begin{array}{l}
\textit{Ontology} \\
\textit{WSMOElement} \\
\hline
\textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\textit{usesMediator} : \mathbb{P}\,\textit{OOMediator} \\
\textit{hasConcept} : \mathbb{P}\,\textit{Concept};\ \ldots\ldots \\
\Delta \\
\textit{totalOntologies} : \mathbb{P}\,\textit{Ontology};\ \textit{totalConcept} : \mathbb{P}\,\textit{Concept} \\
\ldots \\
\textit{totalTerm} : \mathbb{P}(\textit{Concept} \cup \textit{Relation} \cup \textit{Function} \\
\quad \cup \textit{Instance} \cup \textit{RelationInstance}) \\
\hline
\textit{totalOntologies} = (\textit{importsOntology} - \{o : \textit{Ontology} \mid \\
\qquad \exists\, m : \textit{usesMediator} \bullet m.\textit{sourceOntology} = o\}) \\
\quad \cup \{o : \textit{Ontology} \\
\qquad \mid \exists\, m : \textit{usesMediator} \bullet m.\textit{targetOntology} = o\} \\
\textit{totalConcept} = \textit{hasConcep} \cup \\
\quad \bigcup \{o : \textit{totalOntologies} \bullet o.\textit{totalConcept}\} \\
\forall\, c : \textit{hasConcept} \bullet c.\textit{inOntology} = \textit{self} \\
\ldots \\
\textit{totalTerm} = \textit{totalConcept} \cup \textit{totalRelation} \cup \ldots
\end{array}
$$

It is not a trivial task to develop and use an ontology for some particular problem domains. One standard way of dealing with the complexity is modularization. The advantage of modularizing ontologies has been well studied [16]. WSMO uses two different mechanisms – *import* and *mediator*, to design ontologies in a modular way. Importing can be used as long as no conflicts need to be resolved, otherwise an *OOMediator* is necessary. Mediators are described in more detail in Section 3.6.

The basic blocks of an ontology are *concepts*, *relations*, *functions*, *concept instances*, *relation instances* and *axioms*. They are modeled as the attributes of *Ontology*. The secondary attribute *totalOntologies* denotes the set of ontologies whose terms can be used by within the defined ontology. *totalConcept*, *totalRelation*, *totalFunction*, *totalInstance* and *totalRelationInstance* denote all the elements defined within an ontology and imported from other ontologies or OOMediators.

The invariant denotes that:

- an ontology can use the terms defined by the target ontologies of used OOMediators and those imported ontologies which are not sources of any used OOMediators;
- the total *Concept*, *Relation*, *Function* and other ontology elements in an ontology include those elements defined directly in this ontology and all the elements defined in those ontologies in *totalOntologies*.

Only partial model has been presented here.

3.4 Top level element – Web Services

A Web service description in WSMO consists of five sub-components: not-functional properties, imported ontologies, used mediators, a capability and interfaces. The details of these elements will be discussed later in this section. The secondary attribute *totalOntologies* denotes the ontologies whose terms may be used by a Web service, which include the target ontologies of used OOMediators and those imported ontologies which are not sources of any used OOMediators. *ServiceElement* denotes all the Web Service components. It is defined as a subclass of *NestedComponent* with a rename and a secondary attribute *inService*. *NonFunctionalProperty* is a set of properties which strictly belong to a service other than functional and behavioral, e.g. the security level a service must comply. We omit the definition of *ServiceElement* and*NonFunctionalProperty* classes here and present part of *WebSerivce* model.

$$
\begin{array}{l}
\textit{WebService} \\
\textit{WSMOElement} \\
\hline
\textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\textit{usesMediator} : \mathbb{P}(\textit{OOMediator} \cup \textit{WWMediator}) \\
\textit{hasNonFunctionProperty} : \mathbb{P}\,\textit{NonFunctionalProperty} \\
\textit{hasCapability} : \textit{Capability} \\
\textit{hasInterface} : \mathbb{P}\,\textit{Interface} \\
\Delta \\
\textit{totalOntologies} : \mathbb{P}\,\textit{Ontology} \\
\hline
\textit{totalOntologies} = \ldots
\end{array}
$$

3.4.1 Capability

A Web service has *exactly one* capability, which defines the functionality of the service. A Web service capability is defined by specifying the *precondition*, *postcondition*, *assumption*, and *effect*, each of which is a set of *expressions*. A Web service capability also declares a set of variables shared between expressions. The terms used in these expressions must be formally defined in some ontologies which must be imported either directly or via OOMediators. A capability, and therefore a Web service, may be linked to certain goals that are resolved by the Web service via special types of mediators, named *WGMediators* (*WGMediators* will be explained in a later section). The last two predicates in *Capability* invariant ensure that the shared variables have appeared in some used expressions, and the expressions must have used well-defined terms.

$$
\begin{array}{l}
\textit{Capability} \\
\hline
\textit{ServiceElement} \\
\\
\quad \textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\quad \textit{usesMediator} : \mathbb{P}(\textit{OOMediator} \cup \textit{WGMediator}) \\
\quad \textit{hasNonFunctionProperty} : \mathbb{P}\,\textit{NonFunctionalProperty} \\
\quad \textit{hasSharedVariable} : \mathbb{P}\,\textit{Variable} \\
\quad \textit{hasPrecondition}, \textit{hasPostcondition} : \mathbb{P}\,\textit{Expression} \\
\quad \textit{hasAssumption}, \textit{hasEffect} : \mathbb{P}\,\textit{Expression} \\
\quad \Delta \\
\quad \textit{totalOntologies} : \mathbb{P}\,\textit{Ontology} \\
\hline
\quad \ldots\ldots \\
\quad \forall\, oo : \textit{useMediator} \bullet oo \in \textit{OOMediator} \Rightarrow \\
\qquad oo.\textit{sourceOntology} \in \textit{importsOntology} \\
\quad \textit{totalOntologies} = \ldots \\
\quad \textit{hasSharedVariable} \subseteq \\
\qquad \bigcup\{e : \textit{hasPrecondition} \bullet e.\textit{hasVariables}\} \cup \ldots \cup \\
\qquad \bigcup\{e : \textit{hasAssumption} \bullet e.\textit{hasVariables}\} \\
\quad \forall\, e : \textit{hasPrecondition} \cup \textit{hasPostcondition} \cup \\
\qquad\quad \textit{hasEffect} \cup \textit{hasAssumption} \\
\qquad \bullet \forall\, t : e.\textit{usedTerm} \bullet \\
\qquad\quad \exists\, o : \textit{totalOntologies} \cup \textit{inService}.\textit{totalOntologies} \\
\qquad\qquad \bullet t \in o.\textit{totalTerm}
\end{array}
$$

3.4.2 Interfaces

An interface describes how the functionality of the Web service can be achieved from two different views:

- *Choreography* describes the communication pattern that allows one to consume the functionality of the Web service.
- *Orchestration* describes how different Web service providers can operate to achieve the overall functionality of the Web service.

Besides *Choreography* and *Orchestration*, an interface also declares a set of imported ontologies and OOMediators. *InterfaceElement* denotes all the components defined within an *Interface*.

$$
\begin{array}{l}
\textit{Interface} \\
\hline
\textit{ServiceElement} \\
\\
\quad \textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\quad \textit{usesMediator} : \mathbb{P}\,\textit{OOMediator} \\
\quad \textit{hasNonFunctionProperty} : \mathbb{P}\,\textit{NonFunctionalProperty} \\
\quad \textit{hasChoreography} : \mathbb{P}\,\textit{Choreography} \\
\quad \textit{hasOrchestration} : \mathbb{P}\,\textit{Orchestration} \\
\quad \Delta \\
\quad \textit{totalOntologies} : \mathbb{P}\,\textit{Ontology} \\
\hline
\quad \ldots\ldots
\end{array}
$$

In this paper we only show the specification of *Choreography*. WSMO *Choreography* shows how a client deals with the Web service. *Choreography* has three main components. *StateSignature* defines the static part of the state descriptions. *State* (or ground facts) models the dynamic part of the state descriptions, and *transitionRule* models the state changes by changing the values of the ground facts as defined in the set of the imported ontologies.

$$
\begin{array}{l}
\textit{Choreography} \\
\hline
\textit{InterfaceElement} \\
\\
\quad \textit{hasNonFunctionProperty} : \mathbb{P}\,\textit{NonFunctionalProperty} \\
\quad \textit{hasStateSignature} : \textit{StateSignature} \\
\quad \textit{hasState} : \textit{State} \\
\quad \textit{hasTransitionRules} : \mathbb{P}\,\textit{TransitionRule}
\end{array}
$$

StateSignature defines the state ontology used by the service. *importsOntology* denotes a non-empty set of ontologies which defines the state signature over which the transition rules are executed and *usesMediator* denotes a set of OOMediators to solve possible heterogeneity issues among imported state ontologies.

$$\textit{operation} ::= \textit{READ} \mid \textit{WRITE}$$

$$
\begin{array}{l}
\textit{mode} \\
\hline
\textit{ConceptAndRelation} \\
\\
\quad \textit{hasGrounding} : \mathbb{P}\,\textit{operation}
\end{array}
$$

StateSignature also defines the *modes* (or rules) for each concept and relation in the state ontology. We model *mode* as a subclass of the union class of *Concept* and *Relation* (named as *ConceptAndRelation*) with some grounding mechanisms defined. Focusing on the WSMO model itself, we ignore the details of grounding mechanisms and abstract them as either *read* or *write* operations. There are five different types of roles for the *concepts* and *relations*. *Static*, is the default type for all concepts and relations imported by the signature of the choreography, unless defined otherwise in the state signature header. It denotes that the extension of the concept cannot be changed. *in* means that the 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* means that the 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* means that 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* means that the extension of the concept is changed and read only by the choreography execution. The partial invariant of *StateSignature* is presented to capture some of these constraints.

$$
\begin{array}{l}
\textit{StateSignature} \\
\hline
\textit{ChoreographyElement} \\
\\
\quad \textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\quad \textit{usesMediator} : \mathbb{P}\,\textit{OOMediator} \\
\quad \textit{hasNonFunctionProperty} : \mathbb{P}\,\textit{NonFunctionalProperty} \\
\quad \textit{hasStatic}, \textit{hasIn}, \textit{hasOut}, \textit{hasShared}, \textit{hasControlled} \\
\qquad : \mathbb{P}\,\textit{mode} \\
\quad \Delta \\
\quad \textit{totalOntologies} : \mathbb{P}\,\textit{Ontology} \\
\quad \textit{totalConcept} : \mathbb{P}\,\textit{Concept} \\
\quad \textit{totalRelation} : \mathbb{P}\,\textit{Relation} \\
\hline
\quad \ldots \\
\quad \forall\, c : \textit{totalConcept} \cup \textit{totalRelation} \bullet \\
\qquad c \notin \textit{hasIn} \cup \textit{hasOut} \cup \textit{hasShared} \cup \textit{hasControlled} \\
\qquad\quad \Rightarrow c \in \textit{hasStatic} \\
\quad \forall\, m : \textit{hasIn} \bullet \textit{WRITE} \in m.\textit{hasGrounding} \\
\quad \forall\, m : \textit{hasOut} \bullet \textit{READ} \in m.\textit{hasGrounding} \\
\quad \forall\, m : \textit{hasShared} \bullet \{\textit{WRITE}, \textit{READ}\} = m.\textit{hasGrounding} \\
\quad \ldots
\end{array}
$$

State shows the status of a service at a certain point of time and it is defined as a set of ground facts.

$$
\begin{array}{l}
\textit{State} \\
\hline
\textit{ChoreographyElement} \\
\\
\quad \textit{groundInstance} : \mathbb{P}\,\textit{Instance} \\
\quad \textit{groundRelationInstance} : \mathbb{P}\,\textit{RelationInstance} \\
\hline
\quad \forall\, i : \textit{groundInstance} \bullet \exists\, t : i.\textit{hasType} \bullet \\
\qquad t \in \textit{inChoreography}.\textit{hasStateSignature}.\textit{totalConcept} \\
\quad \forall\, i : \textit{groundRelationInstance} \bullet \exists\, t : i.\textit{hasType} \bullet \\
\qquad t \in \textit{inChoreography}.\textit{hasStateSignature}.\textit{totalRelation}
\end{array}
$$

The *transition rules* are used to represent how the service states change. They are triggered when the current state fulfils certain conditions (we omit the formal definition here).

Note that in this paper we only focus on the syntax and static semantic of WSMO. The dynamic semantics of WSMO, which can be formally modeled as a set of Object-Z operations, will be addressed in another paper.

3.5 Top level element – Goals

Goals in WSMO are representations of an objective for which fulfillment is sought through the execution of a Web service. The WSMO *GOAL* can be similarly modeled as WSMO *Service*, but due to the limited space, we will not show the details of its formal specification.

3.6 Top level element – Mediators

Mediator is also a kind of top-level element of WSMO and it is concerned with handling heterogeneity by resolving possibly occurring mismatches between resources. WSMO has four mediators types which connect different WSMO elements (*hasSource* and *hasTarget*) and resolve the mismatches between them using different mediating serives (*hasMediationService*).

$$
\begin{array}{l}
\textit{Mediator} \\
\hline
\textit{WSMOElement} \\
\\
\quad \textit{importsOntology} : \mathbb{P}\,\textit{Ontology} \\
\quad \textit{hasSource}, \textit{hasTarget} : \textit{WSMOElement} \\
\quad \textit{hasMediationService} : \textit{WSMOElement}
\end{array}
$$

3.6.1 OO Mediators

OO Mediators (*OOMediator*) are mainly used to resolve terminological mismatch; and they represent bridging entities between different ontologies. *OOMediator* is modeled as a sub class of *Mediator* with two extra secondary attributes *sourceOntology* and *targetOntology* which denotes the ontologies used as the input and result of a mediation process.

$$
\begin{array}{l}
\textbf{OOMediator} \\
\hline
\mathit{Mediator} \\
\quad \mathit{MediatingOO} : \mathit{Ontology} \times \\
\qquad (\mathit{Service} \cup \mathit{Goal} \cup \mathit{WWMediator}) \nrightarrow \mathit{Ontology} \\
\\
\quad \Delta \\
\quad \mathit{sourceOntology}, \mathit{targetOntology} : \mathit{Ontology} \\
\hline
\quad \mathit{hasSource} \in \mathit{Ontology} \cup \mathit{OOMediator} \\
\quad \mathit{hasSource} \in \mathit{Ontology} \Rightarrow \mathit{sourceOntolgy} = \mathit{hasSource} \\
\quad \mathit{hasSource} \in \mathit{OOMediator} \Rightarrow \\
\qquad \mathit{sourceOntolgy} = \mathit{hasSource.targetOntology} \\
\quad \mathit{hasTarget} \in \\
\qquad \mathit{Ontology} \cup \mathit{Goal} \cup \mathit{Service} \cup \mathit{OOMediator} \\
\quad \mathit{hasTarget} \in \mathit{Ontology} \Rightarrow \mathit{targetOntology} = \mathit{hasTarget} \\
\quad \mathit{hasTarget} \in \mathit{OOMediator} \Rightarrow \\
\qquad \mathit{targetOntology.id} = \mathit{hasTarget.id} \\
\quad \mathit{hasTarget} \in \mathit{Service} \cup \mathit{Goad} \Rightarrow \\
\qquad \mathit{sourceOntolgy} \in \mathit{hasTarget.importsOntolgy} \\
\quad \mathit{hasMediationService} \in \mathit{Service} \cup \mathit{Goal} \cup \mathit{WWMediator} \\
\quad (\mathit{sourceOntology}, \mathit{hasMediationService}, \mathit{targetOntology}) \\
\qquad \in \mathit{MediatingOO}
\end{array}
$$

The class invariant denotes that:

- the source of an OOMediator can be either an *Ontology* or an *OOMediator*.
- if the source is an ontology, the mediation process will be to this ontology.
- if the source is an OOMediator, the role of the source ontology for the defined OOMediator will be played by the target of its source OOMediator.
- the target of an OOMediator can be either an *Ontology*, a *Goal*, a *Service* or an *OOMediator*.
- if the target is an ontology, the result of mediation process will be this ontology as well.
- if the target is an OOMediator, the result of mediation contains terms made available in the name space of the target OOMediator itself.
- an OOMediator with a Goal or a Service as a target component, resolves the heterogeneity problems between its source ontology and the ontologies imported by the Goal. Therefore, the sourceOntology must be in the imported ontologies of the OOMediator's target Goal or Service.
- a *Service*, *WWMediator* or *Goal* can be declared as the *hasMediationService* representing the link which realize the meditating process.
- as we are not interested in any concrete mediation techniques, the relation *MediatingOO* is used to abstract the relations between a source ontology, mediationService and targetOntology.

Due to limited space, we omit the formal representation of other mediator types from this section.

3.7 Logic Expression

In this section we show the formal model for the logical language used for defining formal statements in WSMO. Due to the limited space, we only shows the logical language defined for the WSMO core language.

In WSMO, the value space includes WSMO element value and literal values. We also define a special kind of value called the *nil* value. This is used when variables have not been bound to any concrete values. The WSMO values in general are modeled as a class union:

$$\mathit{Value} \mathrel{\widehat{=}} \mathit{WSMOElement} \cup \mathit{LiteralValue} \cup \mathit{NilValue}$$

$$
\begin{array}{l}
\textbf{NilValue} \\
\hline
\quad v : \mathit{Value} \\
\hline
\quad v = \mathit{self}
\end{array}
\qquad
\begin{array}{l}
\textbf{LiteralValue} \\
\hline
\end{array}
\qquad
\begin{array}{l}
\textbf{Variable} \\
\hline
\quad \mathit{id} : \mathit{VID} \\
\quad v : \mathit{Value}
\end{array}
$$

Variable has two attributes, *id* which denotes the name of a variable; and *v* which denotes the value a variable bounded. *Expression* denotes the general WSMO expression. The *hasVariables* and *usedTerms* denotes the set of variables and WSMO elements used in an expression.

$$
\begin{array}{l}
\textbf{Expression} \\
\hline
\quad \mathit{hasVariables} : \mathbb{P}\,\mathit{Variable};\ \mathit{usedTerms} : \mathbb{P}\,\mathit{WSMOElement}
\end{array}
$$

$$
\begin{array}{l}
\textbf{MemberOfExp} \\
\hline
\mathit{Expression} \\
\\
\quad \mathit{con} : \mathit{Concept} \cup \mathit{Variable};\ \mathit{ins} : \mathit{Instance} \cup \mathit{Variable} \\
\hline
\quad \mathit{hasVariables} = (\{\mathit{con}\} \cup \{\mathit{ins}\}) \cap \mathit{Variable} \\
\quad \mathit{usedTerms} = (\{\mathit{con}\} \cup \{\mathit{ins}\}) \cap \mathit{WSMOElement}
\end{array}
$$

Logic expressions may be simple or complex. There are two basic types of simple logical expressions – molecules expression and relation expressions (*RelExp*). WSMO molecules expressions can have serval forms. *MemberOfExp* denotes the instance molecule with the form of *I memeberOf C*, where *I* is an instance and *C* is a concept. Other forms of molecules expressions, such as

AttListExp, *SubConceptExp*, *ConAttributeDefExp* and relation expressions (*RelExp*), can be defined as well.

WSMO has two kinds of complex logical expressions. The *compound logical expression*, modeled by the class *CompoundExp*, consists of a number of simple logical expressions connected with the keyword *and*. The *Formula* consists of two (simple or compound) logical expression, separated by an implication symbols ($\Leftarrow$, $\Rightarrow$ and $\Leftrightarrow$). They are modeled as *LeftImpExp*, *RightImpExp* and *DualmpExp*. Due to the space limitation, we only shows the model for *LeftImpExp*.

LeftImpExp

Expression

$left : Expression; \; right : Expression$

$usedTerms = left.usedTerms \cup right.usedTerms$
$hasVariables = left.hasVariables \cup right.hasVariables$

4 Discussion

The formal specification of WSMO can be beneficial to the Semantic Web service communities in many different ways, as discussed in the following Subsections.

4.1 Checking the consistency of WSMO language

WSMO is currently a relatively new technology, and thus may still contain errors. As our formal model provides a rigorous foundation of the language, by using existing formal verification tools, it may be possible to find those errors and improve the quality of the WSMO standard. For example, suppose that we define a concept *helpRequest* which has an attribute *ResponseGroup*. The range of *ResponseGroup* is strings[5]. This WSMO definition can be translated into Object-Z as:

$ResponseGroup : Attribute; \; helpRequest : Concept$

$ResponseGroup.hasType = String$
$helpRequest.hasAttribute = \{ResponseGroup, ...\}......$

Note that the translation from WSMO to Object-Z can be automatically realized by a tool. However, when we load our formal WSMO model and the above Object-Z definition into an Object-Z type checker, the tool complains that there is a type error. After studying this problem, we realized that the problem is that according to the WSMO documents (Section 3.3.2), the *hasType* attribute defined for an concept attribute can only have a WSMO concept as its values. On the other hand, *String* is a subclass of literal datatype Value. LiteralValue and Concept are considered disjointed in many Semantic Web languages. Thus, the WSMO standard should be revised as illustrated in Figure 2.

Current WSMO specification:
Class *attribute* sub-Class wsmoElement
  hasType type *concept* ... .
Revised WSMO specification:
Class *attribute* sub-Class wsmoElement
  hasType type **concept or dataType** ... .

Figure 2. WSMO specification revision.

4.2 Making the WSMO language precise and removing ambiguity

Large sections of the WSMO document are in normative text, which could result in several divergent interpretations of the language by different users and tool developers. Furthermore, the documentation makes many assumptions and implications, which are implicitly defined. This could lead to inconsistent conclusion being drawn. Our formal model of WSMO can be used to improve the quality of the normative text that defines the WSMO language, and to help ensure that: the users understand and used the language correctly; the test suite covers all important rules implied by the language; and the tools developed work correctly and consistently.

4.3 Reasoning the WSMO by using exiting formal tools directly

Since Semantic Web Service research in general, and WSMO in particular are still evolving, current verification and reasoning tools (though rudimentary) are also improving. In contrast, there have been decades of development into mature formal reasoning tools that are used to verify the validity of software and systems. By presenting a formal semantic model of WSMO, many Object-Z and Z tools can be possibly used for checking, validating and verifying WSMO model. For example, in our previous work, we have applied Z/EVES [7, 6] and AA [9] separately to reasoning over Web ontologies. In the previous section, we also applied an Object-Z type checker to validate a WSMO model. Instead of developing new techniques and tools, reusing existing tools provides a cheap, but efficient way to provide support and validation for standards driven languages, such as WSMO.

4.4 The ease of extendibility

As WSMO is still evolving, an advantage of using an object-oriented approach in the language model is to

[5]A full version of this example accompanies the WSMO release, and can be found from `http://www.wsmo.org/TR/d3/d3.4/v0.1/`

achieve the extendibility of the language model. Suppose that we want to add a new kind of logic expressions, *NotExp*, to WSMO core. Then in our model it is necessary to add only the following class:

NotExp

Expression

$operand : Expression$

$usedTerms = operand.usedTerms$
$hasVariables = operand.hasVariables$

The introduction of this extension does not involve any changes to the classes defined in the previous section. Validation tools can then be used to confirm the validity of the extended model as can be observed in this example.

5 Conclusion

WSMO is one of the most important technologies for Semantic Web service. It complements the existing syntactic Web service standards, by providing a conceptual model and language for the semantic markup describing all relevant aspects of general services which are accessible through a Web service interface. This paper has presented an Object-Z semantics for WSMO, whereby the WSMO constructs are modeled as objects. The advantage of this approach is that the abstract syntax and static and dynamic semantics for each the WSMO construct are grouped together and captured in an Object-Z class; hence the language model is structural, concise and easily extendible. Subsequent work will address and complete the dynamic semantics of WSMO. We believe this OZ specification can provide a useful document for developing support tools for WSMO.

Acknowledgment

This work is partially supported by the EU-funded TAO project (IST-2004-026460).

References

[1] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In *First International Semantic Web Conference (ISWC) Proceedings*, pages 348–363, 2002.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, May 2001.

[3] J. Bruijn, H. Lausen, A. Polleres, and D. Fensel. The web service modelling language wsml: An overview. In *Proceedings of the 3rd European Semantic Web Conference*, pages 590–604, Budva, Montegegro, June 2006. Springer-Verlag.

[4] R. Chinnici, J. J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. `http://www.w3.org/TR/wsdl20/wsdl20-z.html`, 2006.

[5] J. S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12*, pages 181–190. Prentice-Hall, Nov. 1993.

[6] J. S. Dong, C. H. .Lee, Y. F. Li, and H. Wang. A combined approach to checking web ontologies. In *The 13th ACM International World Wide Web Conference (WWW'04)*, pages 714–722. ACM Press, May 2004.

[7] J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. Verifying DAML+OIL and Beyond in Z/EVES. In *Proc. The 26th International Conference on Software Engineering (ICSE'04)*, pages 201–210, Edinburgh, Scotland, May 2004.

[8] J. S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. Technical Report 95-20, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.

[9] J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *12th Internation Symposium on Formal Methods Europe (FM'03)*. Springer-Verlag, September 2003.

[10] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[11] D. Fensel and E. Motta. Structured development of problem solving methods. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling, and Management (KAW '98), Banff, Canada*, APR 1998.

[12] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *Object-Oriented Systems*, 2:195–215, Chapman & Hall 1995.

[13] S. K. Kim and D. Carrington. Formalizing UML Class Diagram Using Object-Z. In R. France and B. Rumpe, editors, *UML'99*, Lect. Notes in Comput. Sci. Springer-Verlag, Oct. 1999.

[14] S. McIlraith, T. Son, and H. Zeng. Semantic web services, 2001.

[15] Object Management Group. Meta object facility (MOF) specification, 2002. `http://www.omg.org`.

[16] A. Rector. Modularisation of domain ontologies implemented in description logics and related formalisms including owl. In J. Genari, editor, *Knowledge Capture 2003*, pages 121–128, Sanibel Island, FL, 2003. ACM.

[17] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web services modeling ontology. *Journal of Applied Ontology*, 39(1):77–106, 2005.

[18] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, Sept. 1995.

[19] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[20] W. K. Tan. A Semantic Model of A Small Typed Functional Language using Object-Z. In J. S. Dong, J. He, and M. Purvis, editors, *The 7th Asia-Pacific Software Engineering Conference (APSEC'00)*. IEEE Press, Dec. 2000.

[21] J. Woodcock and S. Brien. W : A logic for Z. In *Proceedings of Sixth Annual Z-User Meeting*, University of York, Dec 1991.