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

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 have been one of the most significant research areas within the Semantic Web vision, and have been recognized as a promising technology that exhibits huge commercial potential. Current Semantic Web Service research focuses on defining models and languages for the semantic markup of all relevant aspects of services, which are accessible through a Web service interface. The Web Service Modelling Ontology (WSMO) is one of the most significant Semantic Web Service framework proposed to date. 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 provides a formal unambiguous specification, which can be used to develop tools and facilitate future development.

1 Introduction

The growth of the World-Wide Web and the corresponding evolution of the *Web Service* paradigm; i.e. distributed, heterogeneous, self-contained software components that provide user access to applications supporting tasks such as e-commerce, entertainment, etc., has resulted in an increase in automation and outsourcing of business processes for both B2B and B2C applications [15]. However, it relies heavily on engineering expertise and a-priori agreed service descriptions (using XML-based web service standards) to construct all but the most trivial service-composed application. The assembly of workflows between organisations generally requires that engineers have a deeper understanding of the services (beyond the description of data models or type definitions published within the service descriptions), and often require additional adapters or components that facilitate mediation between otherwise incompatible services. The paradigmatic shift brought by the Semantic Web allows such Web Services to be described in a *machine-understandable* form, thus facilitating automated assistance through tool support, automated interoperability, or even runtime discovery, comprehension, composition and invocation of services. This notion of Semantic Web Services [14] has been one of the most significant research areas within the Semantic Web vision.

Current Semantic Web Service research has focussed on defining models and languages for the semantic markup of all relevant aspects of services, which are then pragmatically accessible through Web Service interface definitions [1, 17]. The Web Service Modeling Ontology (WSMO) [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 Web services. WSMO is based on earlier work on Unified Problem Solving Methods [10], which was part of a *"...framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components..."* [9]. It 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) [16]. 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.

In the linguistics of computer languages, the terms syntax, static semantics and dynamic semantics are used to categorize descriptions of language characteristics. To achieve a consistent usage of a language, all these three aspects must be precisely defined. The syntax and semantics of WSMO are defined in terms of its meta-model. 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 by a variety of different authors, 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. Likewise, the ontological specification of WSMO is formalised using WSML [2] rather than OWL or RDF (as used by OWL-S [1] or SAWSDL[1], for example). To support a common understanding, and facilitate standardization[2] 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) [8]. 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 modelling languages [13, 21]. Object-Z has been used to provide one single formal model 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 our previous work, we have presented the formal model for the syntax and static semantics of WSMO [22]. This paper focuses on the dynamic semantics of WSMO.

The paper is organized as follows. Section 2 briefly introduces the notion of Object-Z, and the formal Object-Z model of the WSMO execution semantics is presented in Section 3. Section 4 discusses some of the benefits of this formal model, and the paper concludes in Section 5 and discusses possible future work.

2 WSMO and Object-Z Overview

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..."[9]. 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) [16]. 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) [2], 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.

Object-Z [8] 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. The motivation for using this extension is that it improves the clarity of large specifications through enhanced structuring.

Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means that no attribute changes value. OZ also allows composite operations to be defined using different operation operators, such as conjunction operator '$\wedge$', parallel operator '$\|$', sequential operator '$\fatsemi$', choice operator '$\Box$' and etc.

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 Web Service

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).

[1]http://www.w3.org/2002/ws/sawsdl/ - Semantic Annotations for WSDL

[2]http://www.w3.org/Submission/WSMO - Web Service Modeling Ontology (WSMO)

We propose the use of Object-Z to provide a formal specification of all aspects of WSMO in one single unified framework, so that the semantics of the language can be more consistently defined and revised as the language evolves. The first two aspects of WSMO (i.e. the formal model of syntax and static semantics) have been addressed in a separate paper [22], whereas this paper focuses the dynamic semantics of WSMO[3], and is based on the latest version of WSMO (D2v1.3).

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 [11] and axiomatic semantics [18] of Object-Z are closely related to Z standard work [23]. Object-Z also has a fully abstract semantics [19].
- Object-Z provides some handy constructs, such as *Class-union* [4] 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) [3]; 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.
- In our previous work [22], OZ has also been used to specify the OWL-S language, which is another significant Semantic Web Service alternative. Modeling both OWL-S and WSMO in the same language provides an opportunity to formally compare the two approaches and identify possible integration and translation between the two languages.

As the model of the dynamic behaviours of WSMO requires the understanding of some basic WSMO entities, such as values, variables and expressions etc, we will start from these basic constructs. 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.

[3]Because of the limited space, we only present a partial model here and a more complete model can be found at `http://www.ecs.soton.ac.uk/~hw/WSMO-OZ.pdf`.

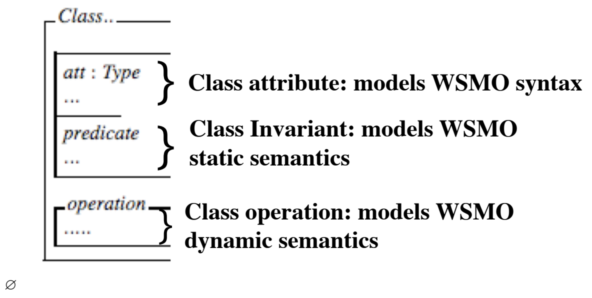
Class..
att : Type
...
Class attribute: models WSMO syntax
predicate
...
Class Invariant: models WSMO static semantics
operation
.....
Class operation: models WSMO dynamic semantics
∅

Figure 1. The framework

3.2 Values, Variables and Expressions

WSMO includes several different types of values in its universe. We first define two special value types namely *nil* and *void*, which are modeled as class *Nil* and *VoidVal*. They are used to define how variables are bound. A variable that equates to *nil* means that the variable has no value (not being bound yet), while a variable that equates to *void* means that it has a value but we do not care what it is. The meaning of a void value is simply just the value itself. When an object of class *Nil* or *VoidVal* is instantiated, the identifier *self* will denote the identity of the object self.

```
Nil              VoidVal          Bool
                                  VoidVal
  V : Value        V : Value
  V = self         V = self         val : 𝔹
```

The class *Bool* and *LiteralValue* denote the boolean and literal (defined as an OZ class) values.

```
LiteralValue          WSMOElement
VoidVal               VoidVal

  val : Literal         id : ID; hasAnnotation :
                          Annotation
```

$$Value == LiteralValue \cup VoidVal \cup Nil \cup Bool \cup \downarrow WSMOElement$$

WSMO refers to all the concepts it defines as *elements*, modeled as *WSMOElement*, and they are one kind of *values* as well. The WSMO *value* in general are modeled as a class union[4].

[4]$\downarrow$ is a special convention in OZ denoting a class and all derivatives of this class.

Variable has two attributes, *id* which denotes the name of the variable (The name of variables is modelled as *VID*); and *v* which denotes the value which the variable is bounded to. Furthermore, three operations defined in *Variable* – *OutVal*, *Assign* and *ToNil*, allows it to output its value, change its value and reset it to a nil value.

$$
\textit{Variable}\quad
\begin{array}{|l}
\hline
id : VID \\
V : Value \\
\hline
\end{array}
\quad
\begin{array}{|l}
\textit{OutVal} \\
\hline
val! : Value \\
\hline
val! = V \\
\hline
\end{array}
\quad
\begin{array}{|l}
\textit{Assign} \\
\hline
\Delta(V) \\
val? : Value \\
\hline
V' = val? \\
\hline
\end{array}
\quad
\begin{array}{|l}
\textit{ToNil} \\
\hline
\Delta(V) \\
\hline
V' \in Nil \\
\hline
\end{array}
$$

The logical expression in WSMO can be divided into simple logical expression and complex logical expression. Before we model any of them, we model some of the common attributes of a WSMO expression first. The *hasVariables* and *usedTerms* denotes the set of variables and WSMO elements used in an expression. *V* represents the truth-value of the expression.

$$
\textit{Expression}\quad
\begin{array}{|l}
\hline
hasVariables : \mathbb{P}\, Variable;\ usedTerms : \mathbb{P} \downarrow WSMOElement \\
V : Bool \cup Nil \\
\hline
\end{array}
$$

MemberOfExp is one form of simple expression which denotes the instance molecule with the form of *I memeberOf C*, where *I* is an instance and *C* is a concept. The last two predicates in the class invariant means that a *MemberOfExp* has 'true' value *if* element *ins* is an instance of concept *con*. Other forms of simple logical expressions, such as *AttListExp*, *SubConceptExp*, *ConAttributeDefExp* and relation expressions (*RelExp*) etc., can be defined as well.

$$
\textit{MemberOfExp}\quad
\begin{array}{|l}
Expression \\
\hline
con : Concept \cup Variable;\ ins : Instance \cup Variable \\
\hline
hasVariables = (\{con\} \cup \{ins\}) \cap Variable \\
usedTerms = (\{con\} \cup \{ins\}) \cap \downarrow WSMOElement \\
(con.V \in Nil \lor ins.V \in Nil) \Rightarrow V \in Nil \\
\neg\, (con.V \in Nil \lor ins.V \in Nil) \Rightarrow \\
\qquad (V \in Bool \land (V.val \Leftrightarrow (con.V \in ins.V.totalType))) \\
\hline
\end{array}
$$

The complex logical expression is extended form the simple expression. For example *LeftImpExp* models $\Leftarrow$ form implication and *ForAllExp* denotes a 'For All' qualified expression. Other types of complex expression, such as *RightImpExp*, *DualmpExp*, *AndExp* and *ExistExp* etc., can be defined as well.

$$
\textit{LeftImpExp}\quad
\begin{array}{|l}
Expression \\
\hline
left :\downarrow Expression;\ right :\downarrow Expression \\
\hline
usedTerms = left.usedTerms \cup right.usedTerms \\
hasVariables = left.hasVariables \cup right.hasVariables \\
left.V \in Nil \lor right.V \in Nil \Rightarrow V \in Nil \\
\neg\, (left.V \in Nil \lor right.V \in Nil) \Rightarrow \\
\qquad (\neg\, left.V.val \lor right.V.val) \Leftrightarrow V.val \\
\hline
\end{array}
$$

$$
\textit{ForAllExp}\quad
\begin{array}{|l}
Expression \\
\hline
var : Variable;\ operand :\downarrow Expression \\
\hline
hasVariables = operand.hasVariables \\
usedTerms = operand.usedTerms \\
var \in hasVariables \\
V.val \Leftrightarrow (\forall\, v : Value \setminus Nil \bullet var.V = v \Rightarrow \\
\qquad operand.V.val \Leftrightarrow \text{true}) \\
\hline
\end{array}
$$

3.3 Web Service model

WSMO has four top-level elements as the main concepts which have to be described in order to define Semantic Web Services: namely, *Ontologies*, *Web Services*, *Goals* and *Mediators*. Among them, *Web Service* is the key element that represents the functional (and behavioural) aspects of a service. Each Web service represents an atomic piece of functionality that can be reused to build more complex ones. Web services are described in WSMO from alternate perspectives: *functionality* and *behaviour*. A Web service can be described by multiple interfaces, but has one and only one *capability*. The capability of a Web service encapsulates its functionality and an interface of a Web Service describes the behaviour of the Web Service from two perspectives: *communication* and *collaboration*. We will formally specify their behaviours later.

3.3.1 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*.

Detailed model of the syntax and static semantics of *Capability* can be found in [22]. The operation *PreAssSat* is used to check if the *preconditions* and *assumptions* are true before service execution. *NotPreAssSat* denotes the opposite situation which is the existence of some preconditions or assumptions whose values are not true. *PostEffSat* is used to check if the *postconditions* and *effects* are true after the execution of the service.

Capability

ServiceElement

$$\mathit{hasPrecondition}, \mathit{hasPostcondition}, \mathit{hasAssumption}, \mathit{hasEffect} : \mathbb{P} \downarrow \mathit{Expression};\ \ldots\ldots$$

PreAssSat

$$\forall\, \mathit{pre} : \mathit{hasPrecondition} \bullet \mathit{pre}.V.\mathit{val} \Leftrightarrow \mathrm{true}$$
$$\forall\, \mathit{ass} : \mathit{hasAssumption} \bullet \mathit{ass}.V.\mathit{val} \Leftrightarrow \mathrm{true}$$

PostEffSat

$$\forall\, \mathit{post} : \mathit{hasPostcondition} \bullet \mathit{post}.V.\mathit{val} \Leftrightarrow \mathrm{true}$$
$$\forall\, \mathit{eff} : \mathit{hasEffect} \bullet \mathit{eff}.V.\mathit{val} \Leftrightarrow \mathrm{true}$$

NotPreAssSat

$$\exists\, e : \mathit{hasPrecondition} \cup \mathit{hasAssumption} \bullet \neg\, (e.V.\mathit{val} \Leftrightarrow \mathrm{true})$$

3.3.2 Interface:

An interface describes how the functionality of the Web service can be achieved from two different views – *Choreography* and *Orchestration*. *Orchestration* describes how the service makes use of other services in order to achieve its capability by defining the communication pattern that allows it to consume the functionality of Web Services and Goals. However these aspects of WSMO are still to be further investigated, and are not covered in this paper.

Interface

ServiceElement

$$\mathit{hasChoreography} : \mathbb{P}\, \mathit{Choreography}$$
$$\mathit{hasOrchestration} : \mathbb{P}\, \mathit{Orchestration};\ \ldots\ldots$$
$$\Delta$$
$$\mathit{enabledChoreography} : \mathbb{P}\, \mathit{Choreography}$$

$$\mathit{enabledChoreography} = \{\mathit{cho} : \mathit{hasChoreography} \mid \mathit{cho}.\mathit{enabledRules} \neq \varnothing\}$$

selectChor

$$\mathit{chor}! : \mathit{enabledChoreography}$$

$$\mathit{Execute} \mathrel{\widehat=} \mathit{selectChor} \bullet \mathit{chor}!.\mathit{Execute}$$

The attribute *enabledChoreography* denotes the set of enabled Choreography, i.e. these Choreography descriptors which enable *transition rules*. The operation *Execute* models the fact that one of enabled choreography defined in the interface will be chosen and executed. The execution of *Choreography* will be defined later.

3.3.3 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. The secondary attribute, *enabledRules*, denotes all transition rules that can be currently performed.

The behavior of *Choreography* is defined by the operation *Execute*. If the knowledge base is inconsistent, the execution will terminate and an error message is returned. ‘MSG’ is defined as a Z given type. The WSMO ontology consists of a number of variants based on the different logical formalisms as defined by the WSML modeling language [2]: *Core*, *DL*, *FLight*, *Rule* and *Full*; and support increasing levels of entailment. Specifying these different semantics in Z or OZ is valuable, as it allows us to reuse existing formal tools to provide reasoning service for those variant ontologies. However, the detailed specification of these ontology semantics is beyond the scope of this paper. Here, the function *checkConsistent* is used to abstract the relations between an ontology or service state and its consistency. If the knowledge base is consistent, an enabled rule will be selected and executed. Note that in our model, the only constraint is that the selected transition rule is a subset of the *enabledRules* (operation *selectEnables*); however, WSMO does not define how the enabled rules are fired. The service providers have the freedom to implement their own systems, e.g., by adding the predicate '$\#toBeExRules! = 1$' in *selectEnables* to indicate that the enabled rule will be fired once every time or by adding '$toBeExRules = enabledRules$' to indicate that all the enabled rules will be fired together. These steps are repeated until no more conditions of any rule are equal to true.

$$
\begin{array}{l}
\textit{Choreography} \\
\hline
InterfaceElement \\
\quad \begin{array}{|l}
\hline
\ldots\ldots \\
hasStateSignature : StateSignature;\ hasState : State; \\
hasTransitionRules : \mathbb{P} \downarrow TransitionRule \\
\Delta \\
enabledRules : \mathbb{P} \downarrow TransitionRule \\
\hline
enabledRules = \\
\qquad \{rule : hasTransitionRules \mid rule.enable \Leftrightarrow \text{true}\} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\textit{selectEnables} \\
\hline
toBeExRules! : \mathbb{P} \downarrow TransitionRule \\
\hline
toBeExRules! \subseteq enabledRules \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\textit{stateConsistent} \\
\hline
consistent! : \mathbb{B} \\
\hline
consistent! = \\
\quad checkConsistent(hasState) \\
\hline
\end{array}
\quad \begin{array}{|l}
\textit{ReportInconsistent} \\
\hline
inconMSG! : MSG \\
\hline
\end{array} \\
FireRules \mathrel{\widehat{=}} \left[toBeExRules? : \mathbb{P} \downarrow TransitionRule \right] \bullet \\
\qquad \bigwedge r : toBeExRules? \bullet r.Execute \\
Execute \mathrel{\widehat{=}} stateConsistent \parallel \\
\qquad (\left[consistent? : \mathbb{B} \mid consistent? \Leftrightarrow \text{true} \right] \bullet (\\
\qquad\quad (\left[enabledRules \neq \varnothing \right] \bullet \\
\qquad\qquad (selectEnables \fatsemi FireRules)) \fatsemi Execute \\
\qquad\quad \Box \\
\qquad\quad (\left[enabledRules = \varnothing \right])) \\
\qquad \Box \\
\qquad \left[consistent? : \mathbb{B} \mid consistent? \Leftrightarrow \text{false} \right] \bullet \\
\qquad\qquad ReportInconsistent) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
checkConsistent : (Ontology \cup State) \rightarrow \mathbb{B}
\end{array}
$$

3.3.4 Transition rules:

In a choreography specification, transaction rules express changes of states by changing the set of instances. *TransitionRule* denotes a WSMO transition rule in general. The secondary attribute *enable* denotes if a *TransitionRule* is ready to fire. *Execute* denotes how a *TransitionRule* changes the world.

$$
\begin{array}{l}
\textit{TransitionRule} \\
\hline
\quad \begin{array}{|l}
\hline
\Delta \\
enable : \mathbb{B} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\textit{Execute} \\
\hline
\hline
\end{array} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{AddMemberOf} \\
\hline
TransitionRule \\
\quad \begin{array}{|l}
\hline
fact : MemberOfExp \\
\hline
enable \Leftrightarrow \\
\quad fact.con \in Concept \\
\quad \wedge fact.ins \in Instance \\
\hline
\end{array}
\quad \begin{array}{|l}
\textit{getCon} \\
\hline
con! : Concept \\
\hline
con! = fact.con \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} \left[enable \Leftrightarrow \text{true} \right] \bullet fact.addType \fatsemi getCon \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{UpdateMemberOf} \\
\hline
TransitionRule \\
\quad \begin{array}{|l}
\hline
oldFact : \mathbb{P}\, MemberOfExp;\ newFact : MemberOfExp \\
\hline
\#oldFact \leq 1 \wedge (\forall o : oldFact \bullet o.ins = newFact.ins) \\
enable \Leftrightarrow newFact.con \in Concept \wedge newFact.ins \in Instance \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\textit{getNewCon} \\
\hline
con! : Concept \\
\hline
con! = newFact.con \\
\hline
\end{array} \\
Execute_1 \mathrel{\widehat{=}} \left[oldFact = \varnothing \right] \bullet \\
\qquad \bigwedge c : \{y : Concept \mid y \neq newFact.con \\
\qquad\qquad\qquad \wedge\ newFact.con \notin y.totalSuperConcept\} \bullet \\
\qquad\qquad \left[con! : Concept \mid con! = c \right] \fatsemi newFact.removeType \\
\qquad \bigwedge getNewCon \fatsemi newFact.addType \\
Execute_2 \mathrel{\widehat{=}} \left[oldFact \neq \varnothing \right] \bullet \\
\qquad \bigwedge of : oldFact \bullet (of.getCon \fatsemi newFact.removeType) \\
\qquad \bigwedge getNewCon \fatsemi newFact.addType \\
Execute \mathrel{\widehat{=}} \left[enable \Leftrightarrow \text{true} \right] \bullet (Execute_1 \Box Execute_2) \\
\hline
\end{array}
$$

The transition rules can be expressed in various forms. The first form is a set of update rules. For example *AddMemberOf* denotes the rules used to assert that an *instance* is a member of a *Concept* and *UpdateMemberOf* denotes the rules to change the membership of an instance. *UpdateMemberOf* has an optional attribute *oldFact*. If *oldFact* is defined (modeled by $Execute_2$), the execution of the rule will delete the old fact and add the new fact. Otherwise, execution will remove all existing memberships of the instance and add the new membership (modeled by $Execute_1$). Due to the space limitation, here we omit the definitions for some ontology operations like *addType removeType* and etc.

IfRule denotes the if-then rule which executes a rule if the condition is satisfied.

$$
\begin{array}{l}
\textit{IfRule} \\
\hline
\textit{TransitionRule} \\
\quad \begin{array}{|l}
\hline
condition :\downarrow Expression;\ rule : TransitionRule \\
\hline
enable \Leftrightarrow (condition.V.val \Leftrightarrow \text{true}) \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} \left[enable \Leftrightarrow \text{true} \right] \bullet rule.Execute \\
\hline
\end{array}
$$

ChooseRule denotes the choose rule which executes an update with an arbitrary binding of a variable chosen among those satisfying the selection condition.

$$
\begin{array}{l}
\textit{ChooseRule} \\
\hline
\textit{TransitionRule} \\
\quad \begin{array}{|l}
\hline
variable : Variable;\ condition :\downarrow Expression \\
rule : TransitionRule \\
\hline
enable \Leftrightarrow (\exists\, v : Value \bullet variable.V = v \Rightarrow \\
\qquad condition.V.val \Leftrightarrow \text{true}) \\
variable \in condition.hasVariables \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} \left[enable \Leftrightarrow \text{true} \right] \bullet \\
\qquad\qquad \left[v? : Value \mid variable.V = v? \Rightarrow \right. \\
\qquad\qquad\qquad \left. condition.V.val \Leftrightarrow \text{true} \right] \bullet \\
\qquad v?.outVal \mathbin{⨟} (variable.Assign \wedge rule.Execute) \\
\hline
\end{array}
$$

ForAllRule denotes the 'for all' rule which simultaneously executes the updates of each binding of a variable satisfying a given condition.

$$
\begin{array}{l}
\textit{ForAllRule} \\
\hline
\textit{TransitionRule} \\
\quad \begin{array}{|l}
\hline
variable : Variable;\ condition :\downarrow Expression \\
rule : TransitionRule \\
\hline
enable \Leftrightarrow (\exists\, v : Value \bullet variable.V = v \Rightarrow \\
\qquad condition.V.val \Leftrightarrow \text{true}) \\
variable \in condition.hasVariables \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} \bigwedge v? : \{ v : Value \mid \\
\qquad\qquad variable.V = v \Rightarrow condition.V.val \Leftrightarrow \text{true} \} \bullet \\
\qquad v?.OutVal \mathbin{⨟} (variable.Assign \wedge rule.Execute) \\
\hline
\end{array}
$$

PipedRule contains a set of rules which is used for non-determinism. When a *PipedRule* is executed, an enabled rule from the rule sets will be randomly picked up and executed.

$$
\begin{array}{l}
\textit{PipedRule} \\
\hline
\textit{TransitionRule} \\
\quad \begin{array}{|l}
\hline
rules : \mathbb{P}\, TransitionRule \\
\hline
rules \neq \varnothing \\
enable \Leftrightarrow (\exists\, rule : rules \bullet rule.enable \Leftrightarrow \text{true}) \\
\hline
\end{array}
\quad \begin{array}{|l}
\textit{SelectRule} \\
\hline
ru? : TransitionRule \\
\hline
ru? \in rules \wedge ru?.enable \Leftrightarrow \text{true} \\
\end{array} \\
Execute \mathrel{\widehat{=}} SelectRule \bullet ru?.Execute \\
\hline
\end{array}
$$

3.3.5 Service execution

Before we formally define the overall execution of a *WebService*, we first introduce a few new attributes to assist in the specification. WSMO itself does not include any grounding standards[5]. In our model, two attributes *inputVar*, *outputVar* and two operations *AssignVarFromEvn*, *WriteToEvn* are used to abstract the communication between a service and its environment[6]. *AssignVarFromEvn* reads the values from the environment and assign them to corresponded variables. If the input value is a WSMO ontology *Instance* or WSMO ontology *RelationInstance*, its type must have the role (mode) as 'in' or 'shared' and cannot be 'controlled'. The secondary attributes *totalIn*, *totalOut* and *totalControl* etc. denote all the *Concept* and *Relation* mode defined by the service's *state signature*[7]. *WriteToEvn* is used to output values to environment. If the output value is WSMO ontology *Instances* or WSMO ontology *RelationInstance*, its type must have the role (mode) as 'out' or 'shared' and cannot be 'controlled'. To make our model more readable, we assume that before communicating with environments, all conflicts (if there are any) has been dealt with by the *OOMediators*.

[5]A general guide has been given for translating WSMO services into WSDL. (`http://wsmo.org/TR/d24/d24.2/v0.1`)

[6]The description of concrete grounding details is beyond the scope of this paper.

[7]More details about *state signature* can be found in [22].

Service

....

$\ldots$

$inputVar, outputVar : \mathbb{P}\, Variable$

Δ

$totalStatic, totalIn, totalOut, totalShared,$
$\quad totalControlled : \mathbb{P}\, mode$

$totalStatic =$
$\quad \bigcup\{c : \bigcup\{i : hasInterface \bullet i.hasChoreography\}$
$\qquad \bullet c.hasStateSignature.hasStatic\}$

$totalIn = \ldots \land totalOut = \ldots \land totalShared = \ldots\ldots$

$\ldots\ldots$

SelectInterfaces

$inter? : hasInterface$

$\exists\, cho : inter?.asChoreography \bullet cho.enabledRules \neq \varnothing$

ReadFromEvn

$\ldots.$

WriteToEvn

$toVars! : \mathrm{seq}\, Variable$

$\mathrm{ran}\, toVars! \subseteq outputVar$

$\forall\, var : \mathrm{ran}\ toVars! \bullet var.V \in Instance \cup RelationInstance$
$\quad \Rightarrow var.V.totalType \cap totalControlled = \varnothing$
$\quad \land var.V.totalType \cap (totalOut \cup totalShared) \neq \varnothing$

$Execute \mathrel{\widehat=} ReadFromEvn \mathbin{⨾} ($
$\qquad hasCapability.PreAssSat$
$\qquad\quad ⨾ SelectInterfaces \bullet inter?.Execute$
$\qquad\quad ⨾ hasCapability.PostEffSat$
$\qquad \Box\ hasCapability.NotPreAssSat)$
$\quad ⨾ WriteToEvn$

The behavior of a *WebService* is defined as the operation *Execute*. Firstly, the service gets requests from users and initialize necessary variables. If the *preconditions* and *assumptions* are satisfied, then the service will be executed, based on the selected *interface* and *postconditions* of the service. However, if the *preconditions* and *assumptions* are not satisfied and the service is invoked, the behavior is undefined. Tool developers have the freedom to implement their own scenarios to handle this situation, such as terminate the execution and report an error message or simply ignore it. After execution, the service will return any necessary information to users (outputs or error messages).

4 Discussions

We have used several Object-Z tools and technology such as [12, 20], to validate and evaluate the syntax, consistency and validity of the formal model and we are also in the process of modeling existing WSMO reference services to further evaluate the full combined model presented here and in [22].

This formal specification of WSMO can be beneficial to the Semantic Web service communities in many different ways, as discussed below:

- **Checking the consistency of WSMO language**

  As stated in Section 1, WSMO is currently a relatively new and continually evolving technology, and thus may still contain errors. The formal model of the dynamic semantics of WSMO presented here (and the accompanying models for the syntax and static semantics [22]) provide a rigorous foundation of the language. As demonstrated previously [22], by using existing formal verification tools, it is possible to find and correct those errors in the specification, and thus 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[8]. This WSMO definition can be translated into Object-Z as:

  $ResponseGroup : Attribute;\ helpRequest : Concept$

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

  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, 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.

- **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

[8]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/`

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.

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: a) the users understand and use the language correctly; b) the test suite covers all important rules implied by the language; and c) the tools developed work correctly and consistently.

- **Reasoning the WSMO by using existing 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 [6, 5] and AA [7] 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.

- **The ease of extendibility**

  As WSMO is still evolving, an advantage of using an object-oriented approach in the language model is to achieve the extendibility of the language model. Suppose that later the Orchestration of WSMO were clearly defined. Then, in our model, all that is required is to simply add and define the class *OrchestrationRule*.

OrchestrationRule
TransitionRule

$target : wgMediator \cup WebService$; ......

$enable \Leftrightarrow$ ......

$Execute \mathrel{\widehat{=}}$ ....

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 formal operational semantics of WSMO. Together with our previous work [22], we have present a complete formal model of WSMO. The advantage of this approach is that the abstract syntax, static and dynamic semantics of each WSMO construct are grouped together and captured in one single Object-Z class; hence the language model is more structural, concise and easily extendible. 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] 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.

[3] 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.

[4] 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, November 1993.

[5] 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.

[6] 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.

[7] 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.

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

[9] Dieter 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.

[10] Dieter Fensel, Enrico Motta, V. Richard Benjamins, Monica Crubezy, Stefan Decker, Mauro Gaspari, Rix Groenboom, William Grosso, Frank van Harmelen, Mark Musen, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The unified problem-solving method development language upml. *Knowledge and Information Systems*, 5(1):83–131, 2002.

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

[12] W. Johnston. A type checker for Object-Z. Technical report 96-24, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, July 1996.

[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, October 1999.

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

[15] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The International Journal on Very Large Data Bases*, 12(1):59–85, 2003.

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

[17] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubn Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter 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, September 1995.

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

[20] G. Smith. Reasoning about Object-Z specifications. In *Proc. Asia Pacific Software Engineering Conference '95 (APSEC'95)*, pages 489–497. IEEE Press, 1995.

[21] 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, December 2000.

[22] H. H. Wang, N. Gibbins, T. Payne, A. Saleh, and J. Sun. A Formal Semantic Model of the Semantic Web Service Ontology (WSMO). In *The 12th IEEE International Conference on Engineering Complex Computer Systems*, Auckland, July 2007.

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