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

Hai H. Wang[a], Nick Gibbins[b], Terry R. Payne[c], Domenico Redavid[d]

[a]*School of Engineering and Applied Science, Aston University, Birmingham, UK h.wang10@aston.ac.uk*
[b]*School of Electronics and Computer Science, University of Southampton, Southampton, UK nmg@ecs.soton.ac.uk*
[c]*School of Computer and Mathematical Sciences, University of Liverpool, Liverpool, UK T.R.Payne@liverpool.ac.uk*
[d]*Dipartimento di Informatica, Universit degli Studi di Bari, Bari, Italy redavid@di.uniba.it*

Abstract

Semantic Web Service, 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 been proposed as an enabling framework for the total/partial automation of the tasks (e.g., discovery, selection, composition, mediation, execution, monitoring and etc.) involved in both intra- and inter-enterprise integration of Web Services. To support the standardisation and tool support of WSMO, a formal model of the language is highly desirable. As several variants of WSMO have been proposed by the WSMO community, which are still under development, the syntax and semantics of WSMO should be formally defined to facilitate easy reuse and future development. In this paper, we present a formal Object-Z formal model of WSMO, where 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 present in existing documentation.

Keywords:
Semantics Web Service, WSMO, Object-Z

1. Introduction

The emergence of *Web services* as a near-ubiquitous standards-based technology has greatly facilitated the uptake and use of this service-oriented paradigm, due to its being built upon de facto Web standards for syntax, addressing, and communication protocols. The use of syntactic frameworks such as XML has enabled the representation and publication of machine-readable, declarative specifications that can be obtained and used by developers and applications alike. This standards-based approach is generally considered as one of the best ways to lead the Web and its related technologies to their full potential. By developing open protocols and guidelines, software components can easily interact with each other across different platforms and different languages in a standardized manner. However, despite the uptake and adoption of standards for services, there are still at least two major obstacles for facilitating (and automating) the construction of large-scale workflows within open environments: *semantic* and *schematic* heterogeneity. The semantics of Web service standards are well-defined (if perhaps implicitly so within Web service tools), and the XML Schema definitions for these standards can be used to validate the service descriptions published by service providers. However, a problem emerges when one considers the definitions of the services themselves, and their messages. Although XML was designed to define the *syntax* of a document, it says nothing about the *semantics* of entities within a document, and consequently does not assist in the interpretation or comprehension of messages or exchange sequences [1].

Semantic Web Services [2, 3] address this problem by providing a declarative, ontological framework for describing services, messages, and concepts in a machine-readable format that can also facilitate logical reasoning. Thus, service descriptions can be interpreted based on their meanings, rather than simply a symbolic representation. Provided that there is support for reasoning over a Semantic Web Service description (i.e. the ontologies used to ground the service concepts are identified, or if multiple ontologies are involved, then alignments between ontologies exist that facilitate the transformation of concepts from one ontology to the other), workflows and service compositions can be constructed based the semantic similarity of the used concepts.

Semantic Web Services can be viewed as "*Semantics + Web Services*", whereby existing Web service descriptions are annotated with semantically rich descriptions, which can be used by applications and middleware to discover, compose, and validate services and workflows. This approach necessitates the development of a Semantic Web Service infrastructure, and it supports the migration of syntactically defined Web services through the inclusion of an additional semantic descriptions (by providing annotations of data, message and service elements that facilitate reasoning). Current Semantic Web Service research has therefore focused on defining models and languages for the semantic markup of all relevant aspects of services, which are accessible through a Web service interface [4, 5].

The Web Service Modelling Ontology (WSMO) is one of the most significant Semantic Web Service frameworks proposed to date [5]. It 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. The language characteristics are defined through the description of each term's *syntax*, *static semantics* and *dynamic semantics*, across a number of documents, in terms of the WSMO metamodel. To achieve a consistent usage of WSMO, all three aspects must be therefore precisely and consistently defined. However, one of the major problems with the existing WSMO definition is that on closer inspection, it is possible to find small deviations in the way the model is defined. This is mainly due to ambiguities in the way they are represented, and more specifically, in the fact that the descriptions are primarily in a 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 a difference in comprehension and hence interpretation for different users (including Web service providers and tool developers) for the same WSMO model. To support a common understanding, and facilitate both standardisation[1] and tool development for WSMO, a formal model of its language is therefore highly desirable. Also, being a relatively young field, research into Semantic Web services and WSMO is still ongoing, and consequently a formal representation of WSMO needs to be reusable and extendable in a way that can accommodate this evolutionary process.

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

We therefore define and present a complete formal denotational model of WSMO using Object-Z (OZ) [6]. A denotational approach has been proved to be one of the most effective ways to define the meaning of a language, and has been used to give formal semantics for many programming and modelling languages [7, 8]. In our work, Object-Z is used to provide a 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. Object-Z (OZ) [6] 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.

The paper is organised as follows. Section 2 briefly introduces WSMO, whereas Section 3 presents how to model the WSMO constructs using Object-Z. Section 4 is devoted to a formal Object-Z model of WSMO syntax and static semantics, whereas Section 5 presents a formalisation of the dynamic (execution) semantics of the WSMO model constructs. In Section 6, we use Amazon Web services as a case study to demonstrate how the Object-Z model can be used to specify a WSMO service model. Section 7 discusses some of the benefits of this formal model, before we conclude the paper and discuss possible future work in Section 8.

2. WSMO

The *Web Service Modelling Ontology (WSMO)* [5] is one of the major proposed approaches for modelling services semantically, and it is based on the earlier work on the 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) [10]. 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) [11], and consist of four core elements deemed necessary to support Semantic Web Services, namely: *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. A typical ontology consists of non-functional properties (from the existing set of predefined core properties), imported ontologies, and the definition of concepts, relations, axioms, functions, and instances of the ontology. In addition, it declares the *ooMediators* (see below) that are used for importing ontologies for which alignment, merging or transformation problems must be solved.

Goals are defined in WSMO as the objectives that a client may have when consulting a Web service. They consist of non-functional properties, imported ontologies, mediators used, postconditions and effects. Postconditions and effects describe the state of the information space and the world, desired by the requester, respectively. Ontologies can be directly imported as the terminology to define the goal when no conflicts need to be resolved. However, if any aligning, merging, or conflict resolution is required, they are imported through *ooMediators*.

Web service descriptions in WSMO provide a semantic description of the semantically annotated Web services themselves, including their functional and non-functional properties, as well as other aspects relevant for interoperating with them. The required terminology, as for goals, can be imported directly or via *ooMediators* when conflicts need to be resolved. In addition, the capability and interfaces of the service are described: a *capability* defines the functional aspects of the offered service, modelled in terms of preconditions, assumptions, postconditions and effects; whereas *interfaces* represent the data-centric interfaces of a service (such as those defined using WSDL), and provide details about its operation in terms of its *choreography* and its *orchestration* (described in Section 4.3.2).

The notion of *Mediators* is introduced by WSMO in order to resolve heterogeneity problems and they are special elements used to link heterogeneous components involved in the modelling of a Web service. They define the necessary mappings, transformations or reductions between the linked elements. Four different types of mediators are defined as *ggMediators*, *ooMediators*, *wgMediators* and *wwMediators*.

- **ggMediators:** These link two goals, expressing the reduction of a source goal into a target goal. They can use ooMediators to bypass the differences in the terminology employed to define these goals. In addition, WSMO allows linking not only of goals, but also of goals to ggMediators, thus allowing the reuse of multiple goals to define a new one.

- **ooMediators:** These import ontologies and resolve possible representation mismatches between them, such as differences in representation languages or in conceptualizations of the same domain.

- **wgMediators:** They link a Web service to a goal. This link represents the (total or partial) fulfilment of the goal by the Web service. The *wgMediators* can use *ooMediators* to resolve heterogeneity problems between the Web service and the goal.
- **wwMediators:** These link two Web services, containing the ooMediators necessary to overcome the heterogeneity problems that may arise in cases where the services use different vocabularies.

WSMO utilises F-Logic [12] as the underlying semantic framework for describing (and reasoning over) concepts and service descriptions.

3. OZ Approach to WSMO Semantics

The existing specification of WSMO *informally* or *semi-formally* describes the language from three different aspects – its *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), *separately*. We propose the use of Object-Z to provide a formal specification of all three aspects of WSMO in one single, unified framework, so that the meaning of the language can be more consistently defined and revised as the language evolves.

Object-Z [6] 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 the conjunction operator '$\wedge$', parallel operator '$\|$', sequential

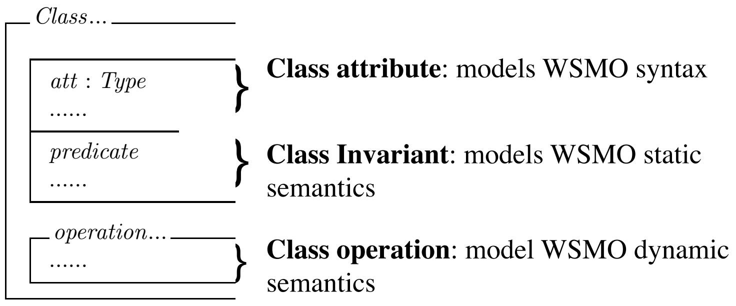
Class...
att : Type
......
predicate
......
operation...
......
Class attribute: models WSMO syntax
Class Invariant: models WSMO static semantics
Class operation: model WSMO dynamic semantics

Figure 1: The framework used to formally model WSMO in Object-Z.

operator '$\fatsemi$', choice operator ' $[]$ ' and etc. The standard behavioral interpretation of Object-Z objects is as a transition system [13], which consists of a series of state transitions each effected by one of the class operations.

Object-Z is chosen over other formalisms for specifying 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 [14] and axiomatic semantics [15] of Object-Z are closely related to Z standard work [16]. Object-Z also has a fully abstract semantics [13].
- Object-Z provides some handy constructs, such as *Class-union* [17], which can define the polymorphic and recursive nature of language constructs effectively. Z has previously been used to specify the Web Service Definition Language (WSDL) [18]; 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 [19, 20], 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.

Figure 1 shows the general approach of the modelling 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 invariants*, used to capture the static semantics of the language. The class operations are used to define WSMO's dynamic semantics, which describes how the state of a Web service changes. Our model is based on the latest version of WSMO (D2v1.3)[2].

4. Formal Object Model of the WSMO Syntax and Static Semantics

In this section, we first formalise the first two aspects of WSMO, i.e., its syntax and static semantics in Object-Z. The dynamic semantics (i.e. how a WSMO model is interpreted and executed) appears in Section 5.

4.1. Modeling identifiers, WSMO elements and annotations

ID

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

$$VID \cap AID = \varnothing$$
$$VID \cap URIID = \varnothing$$
$$URIID \cap AID = \varnothing$$
$$VID \cup AID \cup URIID = ID$$

Every WSMO element is identified by an identifier that can either be classified as a URI reference or an anonymous ID. Furthermore, WSMO can also identify variables. We therefore use the Object-Z class *ID* to denote all possible identifiers. Note that rather than modeling the identifiers as a Z given type ([*ID*]) (the approach adopted by the Z specification of WSDL[18]), 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 (in the above invariant, we only provide an abstract view of the class *ID* without any

[2]`http://www.wsmo.org/TR/d2/v1.3/`

attributes). These concepts can be modeled in more detail, e.g., a *URIID* reference can be expressed by a qualified name and etc.

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

WSMOElement

$id : ID$

$hasAnnotation : \mathbb{P}\ Annotation$

Annotation

$hasContributor : \mathbb{P}\ DC\_Contributor$

$hasDate : \mathbb{P}\ DC\_Date;\ \ldots$

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*, *DC_Date*, and etc. The ‘*DC_*’ prefix refers to the *Dublin Core* [3], which provides a set of commonly used cross-domain metadata definitions (usually used when authoring documents). 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 constraint 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; thus, 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 elements (*NestedComponent*). WSMO has four kinds of top-level elements: *Ontology*, *Service*, *Goal* and *Mediator* – as the main concepts to describe Semantic Web Services (as described above in Section 2). Each of these can be modeled as a subclass of *WSMOElement* (described in more detail in the following subsections). *TopLevelComponent* is modeled as a class union[4].

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

[3]Details on the Dublin Core can be found at `http://dublincore.org/`.

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

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

NestedComponent

WSMOElement

$parentID : ID$

$parentID \notin VID \land id \neq parentID$

OntologyElement

$NestedComponent[inOntologyID / parentID]$

Δ

$inOntology : Ontology$

$\exists_1\ o : Ontology \bullet (inOntologyID = o.id$
$\qquad \land\ inOntology = o)$

4.2. Top-level element – Ontologies

The need to share diverse knowledge and/or information with other applications already built has given rise to a growing interest in research on ontology. Ontologies are domain theories that specify a domain-specific vocabulary of entities, classes, properties, predicates, and functions as a set of relationships that exist among those vocabulary terms [21]. Through the representation of domain-specific knowledge, ontologies provide a way of sharing and reusing domain knowledge among people and heterogeneous applications. *Ontology* is one of the key elements defined in WSMO, and they provide the terminology used by other WSMO elements to describe the relevant aspects of the domains of discourse. Ontologies conceptualise a problem domain by defining a set of *concepts*, *relations*, *instances* and some *axioms*. We provide the formal invariant for *Ontology* below.

4.2.1. Ontology elements

The class *OntologyElement*, defined as a subclass of *NestedComponent*, denotes all the possible WSMO elements defined within the WSMO Ontologies. We rename the attribute *parentID* to *inOntologyID* for the reason of clarification and also define a secondary attribute [22] *inOntology* to denote the Ontology to which *inOntologyID* refers. The invariant shows that there exists one and only one Ontology given an *inOntologyID*.

4.2.2. Concepts

Concepts constitute the basic elements of the agreed terminology for some problem domains, and thus the *Concept* class is derived from *OntologyElement*. The *hasSuperConcept* attribute denotes the super-concepts of a concept. *hasAttribute* and *hasInstance* denote the *attributes* and *instances* explicitly defined for a concept, whereas the secondary attribute *totalSuperConcept* denotes all the ancestor concepts. The 'total-attributes' are denoted by *totalAttr* – explicitly declared in a concept and implicitly inherited; whereas *totalIns* denotes the instances of a concept and its super-concepts. Finally, *hasDefinition* denotes the logical expression used to define the semantics of a concept (the logical expressions used in WSMO are formally defined in Section 5.1).

Concept

OntologyElement

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

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 left-hand side and right-hand side of the expression have only one common free variable. The terms used in the expression must be defined in *Ontology*, and the terms used in the definition must be well defined;
- a concept inherits the attributes of this superconcepts.

The elements defined within *Concept* are defined as *ConceptElement*. It is derived from *NestedComponent*, the attribute *parentID* is renamed to *InConceptID* and a secondary attribute *inConcept* is defined. *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*.

```
Attribute
  ConceptElement
  typeModel ::= ofType | impliesType

    hasType : Concept; hasTypeModel : typeModel
    ──────
    hasType ∈ inConcept.inOntology.totalConcept
```

```
Instance
  OntologyElement

    hasType : ℙ Concept; hasAttributeValues :
                          ℙ AttributeValue
    Δ
    totalType : ℙ Concept
    ──────
    ...
```

4.2.3. Instances

WSMO instances are modelled as *Instance* and *hasType* denotes the explicitly asserted concepts of which the instance is an instance, while *totalType* denotes all asserted and inferred type concepts. The *Relation* and *RelationInstance* of *Ontology* can be similarly defined.

AttributeValue

InstanceElement

$hasAttribute : Attribute$

$hasValues : \mathbb{P}\, Instance$

$\forall v : hasValues \bullet hasAttribute.hasType \in v.totalType$

4.2.4. Ontologies

It is not a trivial task to develop and use an ontology for a particular problem domain, where the modelling of the domain can depend on a number of factors, such as the relevant characteristics of a domain, the type of reasoning required from the domain model itself and likewise the questions that it should be able to answer. There has been a significant focus on ontology reuse, and one common mechanism for identifying self-contained ontological fragments that can be shared and reused is through *ontology modularization* [23, 24]. Thus, ontologies can be modularised, shared, and reused (through inclusion or reference) to construct other ontologies for similar tasks. WSMO uses two different mechanisms – *import* and *mediator*, to design ontologies in a modular way. Importing can be used as long as no conflict to be resolved, otherwise an *ooMediator* will be necessary. Mediators will be described in more detail in Section 4.5.

The basic blocks of an ontology are *concepts*, *relations*, *functions*, *concept instances*, *relation instances* and *axioms*. They are all modelled as the attributes of *Ontology*. The secondary attribute, *totalOntologies*, denotes the set of ontologies whose terms can be used within the defined ontology. *totalConcept*, *totalRelation*, *totalFunction*, *totalInstance* and *totalRelationInstance* denote the elements defined within an ontology and imported from other ontologies or ooMediators. We present a partial model of the *Ontology* invariant below, which denotes that:

- an ontology can use the terms defined by the target ontologies of ooMediators, and those imported ontologies which are not sources of any ooMediators used;

- the total *Concepts*, *Relations*, *Functions* 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*.

Ontology

WSMOElement

$importsOntology : \mathbb{P}\ Ontology;\ usesMediator : \mathbb{P}\ ooMediator$

$hasConcept : \mathbb{P}\ Concept;$ ......

Δ

$totalOntologies : \mathbb{P}\ Ontology;\ totalConcept : \mathbb{P}\ Concept$

...

$totalTerm : \mathbb{P} \downarrow WSMOElement$

$$
\begin{aligned}
totalOntologies = {} & (importsOntology - \{o : Ontology \mid \\
& \qquad \exists\, m : usesMediator \bullet m.sourceOntology = o\}) \\
& \cup \{o : Ontology \\
& \qquad \mid \exists\, m : usesMediator \bullet m.targetOntology = o\}
\end{aligned}
$$

$$
\begin{aligned}
totalConcept = {} & hasConcep \cup \\
& \bigcup \{o : totalOntologies \bullet o.totalConcept\}
\end{aligned}
$$

$\forall\, c : hasConcept \bullet c.inOntology = self$

...

$totalTerm = totalConcept \cup totalRelation \cup \ldots$

4.3. Top-level element – Web services

A Web service description in WSMO consists of five sub-components: *non-functional properties*, *imported ontologies*, *used mediators*, a *capability* and *interfaces* (further 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 any ooMediators used, and those imported ontologies which are not sources of any of those ooMediators. *ServiceElement* denotes all the Web service components, and it is defined as a subclass of *NestedComponent* with a renamed attribute and a secondary attribute *inService*. *NonFunctionalProperty* is a set of properties which strictly belongs to a service other than functional and behavioral, e.g. the security level to which a service must comply. We omit the definition of *ServiceElement* and *NonFunctionalProperty* classes here and present only part of the *Service* model.

Service

WSMOElement

$importsOntology : \mathbb{P}\ Ontology$

$usesMediator : \mathbb{P}(ooMediator \cup WWMediator)$

$hasNonFunctionProperty : \mathbb{P}\ NonFunctionalProperty$

$hasCapability : Capability;\ hasInterface : \mathbb{P}\ Interface$

$totalOntologies = \ldots$

4.3.1. 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*. 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* (these are described below). 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.

Capability

ServiceElement

$importsOntology : \mathbb{P}\ Ontology$

$usesMediator : \mathbb{P}(ooMediator \cup WGMediator)$

$hasNonFunctionProperty : \mathbb{P}\ NonFunctionalProperty$

$hasSharedVariable : \mathbb{P}\ Variable$

$hasPrecondition, hasPostcondition : \mathbb{P} \downarrow Expression$

$hasAssumption, hasEffect : \mathbb{P} \downarrow Expression$

Δ

$totalOntologies : \mathbb{P}\ Ontology$

......

$\forall\, oo : usesMediator \bullet oo \in ooMediator \Rightarrow oo.sourceOntology \in importsOntology$

$totalOntologies = ...$

$hasSharedVariable \subseteq$

$\quad \bigcup\{e : hasPrecondition \bullet e.hasVariables\} \cup ... \cup$

$\quad \bigcup\{e : hasAssumption \bullet e.hasVariables\}$

$\forall\, e : hasPrecondition \cup hasPostcondition \cup hasEffect \cup hasAssumption$

$\quad \bullet \forall\, t : e.usedTerms \bullet$

$\quad\quad \exists\, o : totalOntologies \cup inService.totalOntologies \bullet t \in o.totalTerm$

4.3.2. Interfaces

An interface describes how the functionality of a Web service can be achieved from both a choreographic and an orchestral view. These views determine how the interface is utilised when being attached to several other services. We define the notions of *orchestration* and *choreography* below.

Orchestration is a declarative specification that describes a workflow to support the execution of a specific business process, operation or service. Languages such as BPEL4WS describe both the data and process flow through the workflow; i.e. the pattern of interactions that a Web service agent must follow; as well as notions such as transactions and role-back (in case of service failure). For example, a supply chain process orchestration might describe the business protocol that formalises: 1) the information elements a product-order may consist of; 2) the order (and references to interfaces defined in WSDL) that these elements should be provided; and 3) what exceptions may have to be handled. Orchestrations are typically *enacted* by a single workflow engine, which is responsible for managing the communication and data flow between services. The role of orchestration has been a significant factor in the development of both eBusiness and eScience workflows, and many semi-automated tools and workbenches have evolved to facilitate the construction of these workflows [25, 26, 27].

Choreography defines the sequence (and conditions) under which multiple, cooperating, independent agents exchange messages in order to perform a task to achieve a goal state. Languages such as WS-Choreography (which is based on Pi-Calculus [28]) describe the interactions with an invoking party (which might be other Web Services, applications or human beings), and may consist of multiple separate interactions whose composition constitutes a complete transaction. This composition, along with its message protocols, interfaces, sequencing, and associated logic, is considered to be a choreography[5].

Thus, to summarise, the *Choreography* describes the communication pattern that allows one to consume the functionality of the Web service, whereas the *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*.

[5]Adapted from the W3C Glossary - http://www.w3.org/2003/glossary/

Interface

ServiceElement

$$\begin{array}{l}
\mathit{importsOntology} : \mathbb{P}\ \mathit{Ontology};\\
\qquad\qquad \mathit{usesMediator} : \mathbb{P}\ \mathit{ooMediator}\\
\mathit{hasNonFunctionProperty} :\\
\qquad\qquad \mathbb{P}\ \mathit{NonFunctionalProperty}\\
\mathit{hasChoreography} : \mathbb{P}\ \mathit{Choreography};\\
\qquad\qquad \mathit{hasOrchestration} : \mathbb{P}\ \mathit{Orchestration}\\
\Delta\\
\mathit{totalOntologies} : \mathbb{P}\ \mathit{Ontology}\\
\hline
\ldots\ldots
\end{array}$$

Choreography

InterfaceElement

$$\begin{array}{l}
\mathit{hasNonFunctionProperty} : \mathbb{P}\ \mathit{NonFunctionalProperty}\\
\mathit{hasStateSignature} : \mathit{StateSignature};\ \mathit{hasState} : \mathit{State}\\
\mathit{hasTransitionRules} : \mathbb{P} \downarrow \mathit{TransitionRule}
\end{array}$$

In this paper we only present the specification of *Choreography*, as the WSMO community is still working on defining the *Orchestration*, and thus is not yet stable enough to warrant modelling. WSMO *Choreography* models how a client deals with the Web service and it has three main components: *StateSignature*, *State* and *TransitionRule*. The *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.

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

$$\mathit{mode} : (\mathit{Relation} \cup \mathit{Concept}) \nrightarrow \mathbb{P}\ \mathit{operation}$$

StateSignature

ChoreographyElement

$importsOntology : \mathbb{P}\, Ontology;\ usesMediator : \mathbb{P}\, ooMediator$

$hasNonFunctionProperty : \mathbb{P}\, NonFunctionalProperty$

$hasStatic, hasIn, hasOut, hasShared, hasControlled : \mathbb{P}\, Concept \cup Relation$

Δ

$totalOntologies : \mathbb{P}\, Ontology;\ totalConcept : \mathbb{P}\, Concept$

$totalRelation : \mathbb{P}\, Relation$

...

$\forall\, c : totalConcept \bullet c \notin (hasIn \cup hasOut \cup hasShared \cup hasControlled) \Rightarrow c \in hasStatic$

$\forall\, c : totalRelation \bullet c \notin (hasIn \cup hasOut \cup hasShared \cup hasControlled) \Rightarrow c \in hasStatic$

$\forall\, m : hasIn \bullet WRITE \in mode(m)$

$\forall\, m : hasOut \bullet READ \in mode(m)$

$\forall\, m : hasShared \bullet \{WRITE, READ\} = mode(m)$...

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 which solves possible heterogeneity issues among the imported state ontologies.

StateSignature also defines the *mode* (or rule) for each concept and relation in the state ontology. We model *mode* as a function which maps a *Concept* or *Relation* to some defined grounding mechanisms. 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 *concepts* and *relations*: *static*, *in*, *out*, *shared*, and *controlled*. These are defined below:

- *static* is the default type for all concepts and relations imported by the signature of a choreography, unless defined otherwise in the state signature header. It denotes that the extension of the concept cannot be changed.

- *in* corresponds to the extension of the concept or relation which 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. As with *in*, a grounding mechanism 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 that implements read/write access for the environment and the service, may be provided, and thus is optional.

- *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.

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

State

ChoreographyElement

$groundInstance : \mathbb{P}\ Instance$

$groundRelationInstance : \mathbb{P}\ RelationInstance$

$\forall i : groundInstance \bullet \exists t : i.hasType \bullet$
$\quad t \in inChoreography.hasStateSignature.totalConcept$

$\forall i : groundRelationInstance \bullet \exists t : i.hasType \bullet$
$\quad t \in inChoreography.hasStateSignature.totalRelation$

The *transition rules* are used to represent how the service states change. They are triggered when the current state fulfils certain conditions. We will formally define them in the next section. Likewise, the dynamic semantics of WSMO, which can be formally modeled as a set of Object-Z operations, will be addressed in next section.

4.4. Top-level element – Goals

Goals in WSMO are representations of an objective for which fulfilment is sought through the execution of a Web service. WSMO *Goal* can be similarly modelled as WSMO *Service*, but due to the limited space, we will not show the details of its formal specification. The only difference is that *gwMediator* and *ggMediator* can be used to resolve possibly occurring mismatches between *Goals* and *Web Services*.

Goal

$WSMOElement$

$importsOntology : \mathbb{P}\ Ontology$
$usesMediator : \mathbb{P}(OOMediator \cup GWMediator \cup GGMediator)$
$hasNonFunctionProperty : \mathbb{P}\ NonFunctionalProperty$
$hasCapability : Capability;\ hasInterface : \mathbb{P}\ Interface$

$totalOntologies = ...$

4.5. Top-level element – Mediators

Mediator is concerned with the handling and management of heterogeneity, and achieves this by resolving possibly occurring mismatches between resources. WSMO uses four specific mediator types to connect different WSMO elements (*hasSource* and *hasTarget*) and resolve the mismatches between them using different mediating services (*hasMediationService*).

Mediator

$WSMOElement$

$importsOntology : \mathbb{P}\ Ontology$
$hasSource, hasTarget, hasMediationService : \downarrow WSMOElement$

4.5.1. ooMediators

$$\mathit{ooMediator}$$

$$\mathit{Mediator}$$

$$\begin{array}{l}
\mathit{MediatingOO} : \mathit{Ontology} \times \\
\qquad (\mathit{Service} \cup \mathit{Goal} \cup \mathit{WWMediator}) \nrightarrow \mathit{Ontology}
\end{array}$$

$$\begin{array}{l}
\Delta \\
\mathit{sourceOntology}, \mathit{targetOntology} : \mathit{Ontology} \\
\hline
\mathit{hasSource} \in \mathit{Ontology} \cup \mathit{ooMediator} \\
\mathit{hasSource} \in \mathit{Ontology} \Rightarrow \mathit{sourceOntology} = \mathit{hasSource} \\
\mathit{hasSource} \in \mathit{ooMediator} \Rightarrow \\
\qquad \mathit{sourceOntology} = \mathit{hasSource}.\mathit{targetOntology} \\
\mathit{hasTarget} \in \mathit{Ontology} \cup \mathit{Goal} \cup \mathit{Service} \cup \mathit{ooMediator} \\
\mathit{hasTarget} \in \mathit{Ontology} \Rightarrow \mathit{targetOntology} = \mathit{hasTarget} \\
\mathit{hasTarget} \in \mathit{ooMediator} \Rightarrow \mathit{targetOntology}.\mathit{id} = \mathit{hasTarget}.\mathit{id} \\
\mathit{hasTarget} \in \mathit{Service} \cup \mathit{Goal} \Rightarrow \\
\qquad \mathit{sourceOntology} \in \mathit{hasTarget}.\mathit{importsOntolgy} \\
\mathit{hasMediationService} \in \mathit{Service} \cup \mathit{Goal} \cup \mathit{WWMediator} \\
\mathit{MediatingOO}(\mathit{sourceOntology}, \mathit{hasMediationService}) = \mathit{targetOntology}
\end{array}$$

Ontology to Ontology Mediators (*ooMediator*) are mainly used to resolve terminological mismatch; and they represent bridging entities between different ontologies. *ooMediator* is modelled as a subclass of *Mediator* with two extra secondary attributes: *sourceOntology* and *targetOntology*, which denote the ontologies used as the input and result of a mediation process.

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 the mediation process will also be that ontology.

- 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 included 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 realises the meditation process. As we are not interested in any concrete mediation techniques, the relation *MediatingOO* is used to abstract the links between a source ontology, mediationService and targetOntology.

The invariants for other mediator types are similar to that of *ooMediator*, as illustrated in the *ggMediator* (below), and thus the invariants for *wgMediator* and *wwMediator* are not included here.

The class invariant for *ggMediator* denotes that:

- the source of a ggMediator can be either a *Goal* or a *ggMediator*.

- if the source is a goal, the mediation process will be to this goal.

- if the source is a ggMediator, the role of the source goal for the defined ggMediator will be played by the target of its source ggMediator.

- the target of a ggMediator can be either a *Goal*, or a *ggMediator*.

- if the target is a goal, the result of the mediation process will also be this goal.

- a *Service*, *wwMediator* or *Goal* can be declared as the *hasMediationService* representing the link, which realise the mediation process.

ggMediator

Mediator

$MediatingGG : (Goal \cup ggMediator) \times$
$\quad (Service \cup Goal \cup WWMediator) \nrightarrow (Goal \cup ggMediator)$

Δ

$sourceGoal, targetGoal : Goal$

$hasSource \in Goal \cup ggMediator$

$hasSource \in Goal \Rightarrow sourceGoal = hasSource$

$hasSource \in ggMediator \Rightarrow$
$\quad sourceGoal = hasSource.targetGoal$

$hasTarget \in Goal \cup ggMediator$

$hasTarget \in Goal \Rightarrow targetGoal = hasTarget$

$hasTarget \in ggMediator \Rightarrow targetGoal.id = hasTarget.id$

$hasMediationService \in Service \cup Goal \cup WWMediator$

$MediatingGG(sourceGoal, hasMediationService) = targetGoal$

5. Formal Object Model of the WSMO Dynamic (Execution) Semantics

In this section, we extend the syntax and static semantics model of WSMO defined in the previous section and formalise the third aspect of WSMO, i.e., its execution semantics, in Object-Z. As the model of the dynamic behaviours of WSMO requires the understanding of some basic WSMO entities, such as values, variables, expressions and etc., we start from these basic constructs.

5.1. Values, Variables and Logic Expressions

In WSMO, the value space includes both WSMO element values and literal values. We also define two special value types which are used to define how variables are bound: *nil* and *void*, which are modelled as class *Nil* and *VoidVal* respectively. A variable that equates to *nil* means that the variable has no value (not being bound yet), whereas 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* denotes the identity of the object self.

Nil

$V : Value$

$V = self$

VoidVal

$V : Value$

$V = self$

Bool

VoidVal

$val : \mathbb{B}$

Bool and *LiteralValue* denote both boolean and literal values (defined as OZ classes).

literal

LiteralValue

VoidVal

$val : Literal$

WSMOElement

VoidVal

......

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

As all the elements defined by WSMO are also of type *values*, we extend the class *WSMOElement* (defined in the previous section) as a subclass of *VoidVal*. The WSMO *Value* is modeled as a class union. *Variable* has two attributes: *id* which denotes the name of a variable (variable names are modelled as *VID*); and *v* which denotes the value to which a variable is bounded. Furthermore, three operations defined in *Variable*; i.e. *OutVal*, *Assign* and *ToNil*, which allow it to output its value, change its value and reset it to a *nil* value respectively.

Variable

$id : VID$
$V : Value$

OutVal
$val! : Value$
$val! = V$

Assign
$\Delta(V)$
$val? : Value$
$V' = val?$

ToNil
$\Delta(V)$
$V' \in Nil$

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

Expression

$hasVariables : \mathbb{P}\ Variable;\ usedTerms : \mathbb{P} \downarrow WSMOElement$
$V : Bool \cup Nil$

There are two basic types of simple logical expressions: molecule expressions, and relation expressions (*RelExp*). WSMO molecule expressions can have several forms. For example, *MemberOfExp* 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 represent the fact that a *MemberOfExp* has the value 'true' *if* element *ins* is an instance of concept *con*. *hasValue* models the relation expression which denotes if an instance has a particular attribute value. Other forms of simple logical expressions, such as *AttListExp*, *SubConceptExp*, *ConAttributeDefExp* and relation expressions (*RelExp*) and etc., can likewise be defined in a similar way.

MemberOfExp

Expression

$con : Concept \cup Variable;\ ins : Instance \cup Variable$

$hasVariables = (\{con\} \cup \{ins\}) \cap Variable$

$usedTerms = (\{con\} \cup \{ins\}) \cap \downarrow WSMOElement$

$(con.V \in Nil \lor ins.V \in Nil) \Rightarrow V \in Nil$

$\neg\ (con.V \in Nil \lor ins.V \in Nil) \Rightarrow (V \in Bool \land (V.val \Leftrightarrow (con.V \in ins.V.totalType)))$

hasValue

Expression

$ins : Instance \cup Variable;\ att : Attribute \cup Variable;\ val : Value \cup Variable$

$hasVariables = (\{ins\} \cup \{att\} \cup \{val\}) \cap Variable$

$usedTerms = (\{ins\} \cup \{att\} \cup \{val\}) \cap \downarrow WSMOElement$

$(ins.V \in Nil \lor att.V \in Nil \lor val.V \in Nil) \Rightarrow V \in Nil$

$\neg\ (ins.V \in Nil \lor att.V \in Nil \lor val.V \in Nil) \Rightarrow$
$\quad (V \in Bool \land (V.val \Leftrightarrow$
$\qquad (\forall\, av \in ins.V.hasAttributeValues \bullet av.hasAttribute = att \Rightarrow av.hasValues = \{val.V\}))$

Complex logical expressions are extended form the simple expressions. For example, *LeftImpExp* models a '⇐' implication and *ForAllExp* denotes a 'For All' qualified expression. Other types of complex expression, such as *RightImpExp*, *DualmpExp*, *AndExp*, *OrExp*, *ExistExp* and etc., can also be defined similarly.

LeftImpExp

Expression

$left :\downarrow Expression;\ right :\downarrow Expression$

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

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

$left.V \in Nil \vee right.V \in Nil \Rightarrow V \in Nil$

$\neg\,(left.V \in Nil \vee right.V \in Nil) \Rightarrow (\neg\, left.V.val \vee right.V.val) \Leftrightarrow V.val$

ForAllExp

Expression

$var : Variable;\ operand :\downarrow Expression$

$hasVariables = operand.hasVariables$

$usedTerms = operand.usedTerms$

$var \in hasVariables$

$V.val \Leftrightarrow (\forall\, v : Value \setminus Nil \bullet var.V = v \Rightarrow operand.V.val \Leftrightarrow \text{true})$

5.2. Web Service execution model

As shown in Section 4, WSMO has four top-level elements as the main concepts which have to be described in order to define Semantic Web Services. They are *Ontologies*, *Web Services*, *Goals* and *Mediators*. Out of these, the *Web Service* element is the key element for representing the functional (and behavioural) aspects of a service, and it connects computers and devices with each other using the standard Web-based protocols to exchange data and combine data in new ways. Thus, we focus on the execution semantics of *Services* within this section. In WSMO, 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 two different 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 – choreography and orchestration (as described in Section 4.3.2).

5.2.1. Capability

A Web service capability is defined by specifying the *precondition*, *postcondition*, *assumption*, and *effect*, each of which is a set of *expressions*. Extending the definition of *Capability* defined in Section 4, 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

......

PreAssSat

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

PostEffSat

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

NotPreAssSat

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

Interface

ServiceElement

......

$$enabledChoreography : \mathbb{P}\ Choreography$$

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

selectChor

$$chor! : enabledChoreography$$

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

5.2.2. Interface

To model the execution of an *Interface*, the attribute *enabledChoreography* is introduced to denote the set of enabled choreographies, i.e. these *Choreography* elements that have enabled *transition rules*. The operation *Execute* models the fact that one of the enabled choreographies defined in an interface will be chosen and executed. The execution of *Choreography* is defined below.

WSMO *Choreography* defines how a client deals with the Web service. As introduced earlier (Section 4.3.2), *Choreography* has three main components: *StateSignature*, which defines the static part of the state descriptions; *State* (or ground facts), which models the dynamic part of the state descriptions; and *transitionRule* which models the state changes by changing the values of the ground facts as defined in the set of imported ontologies. The secondary attribute, *enabledRules*, denotes all the valid transition rules that can be currently performed.

Choreography

$$\begin{array}{l}
\ldots\ldots \\
enabledRules : \mathbb{P} \downarrow TransitionRule \\
\hline
enabledRules = \{rule : hasTransitionRules \mid rule.enable \Leftrightarrow \text{true}\}
\end{array}$$

selectEnables

$$\begin{array}{l}
toBeExRules! : \mathbb{P} \downarrow TransitionRule \\
\hline
toBeExRules! \subseteq enabledRules
\end{array}$$

stateConsistent

$$\begin{array}{l}
consistent! : \mathbb{B} \\
\hline
consistent! = checkConsistent(hasState)
\end{array}$$

ReportInconsistent

$$\begin{array}{l}
inconMSG! : MSG
\end{array}$$

$$FireRules \mathrel{\widehat{=}} \left[\, toBeExRules? : \mathbb{P} \downarrow TransitionRule \,\right] \bullet \bigwedge r : toBeExRules? \bullet r.Execute$$

$$\begin{array}{l}
Execute \mathrel{\widehat{=}} stateConsistent \parallel \\
\quad (\left[\, consistent? : \mathbb{B} \mid consistent? \Leftrightarrow \text{true} \,\right] \bullet (\\
\qquad (\left[\, enabledRules \neq \varnothing \,\right] \bullet (selectEnables \mathbin{⨾} FireRules)) \mathbin{⨾} Execute \\
\qquad [\!] \\
\qquad (\left[\, enabledRules = \varnothing \,\right])) \\
\quad [\!] \\
\quad \left[\, consistent? : \mathbb{B} \mid consistent? \Leftrightarrow \text{false} \,\right] \bullet ReportInconsistent)
\end{array}$$

$$checkConsistent : (Ontology \cup State) \rightarrow \mathbb{B}$$

$[MSG]$

The behavior of *Choreography* is defined by the operation *Execute*. If the knowledge base is inconsistent, the execution will terminate and an error message of type 'MSG' will be returned. The WSMO ontology consists of a number of variants based on several different logical formalisms, namely *"Core"*, *"DL"*, *"Flight"*, *"Rule"* and *"Full"* [11]. The semantics of these variants is different, which may also result in different entailments. Specifying these different semantics in Z or Object-Z 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. Dong *et. al.* [29, 30] have previously presented a Z semantics for DL based ontologies.

If the knowledge base is consistent, some enabled rules will be selected and executed. Note that in our model, we impose an additional restriction as that a subset of the *enabledRules* is selected as transition rules (operation *selectEnables*), whereas WSMO does not define how the enabled rules are fired. The service providers will 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.

TransitionRule

Δ

$enable : \mathbb{B}$

Execute

AddMemberOf

TransitionRule

$fact : MemberOfExp$

$enable \Leftrightarrow fact.con \in Concept$
$\quad \land fact.ins \in Instance$

getCon

$con! : Concept$

$con! = fact.con$

$Execute \mathrel{\widehat{=}} \left[enable \Leftrightarrow \mathrm{true} \right] \bullet fact.addType \mathbin{⨾} getCon$

5.2.3. Transition rules

In a choreography specification, the 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 whether or not a *TransitionRule* is ready to perform. *Execute* denotes how a *TransitionRule* changes the world.

The transition rules can be represented in several alternate formats. The first corresponds to 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. The *UpdateMemberOf* transition rule has an optional attribute *oldFact*, which if defined (modelled by $Execute_2$), then 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 (modelled by $Execute_1$). Due to the space limitation, here we omit the definitions for some ontology operations like *addType removeType* and etc.

UpdateMemberOf

$TransitionRule$

$oldFact : \mathbb{P}\, MemberOfExp;\ newFact : MemberOfExp$

$\#oldFact \leq 1 \wedge (\forall\, o : oldFact \bullet o.ins = newFact.ins)$

$enable \Leftrightarrow newFact.con \in Concept \wedge newFact.ins \in Instance$

getNewCon

$con! : Concept$

$con! = newFact.con$

$$\begin{aligned}
&Execute_1 \mathrel{\widehat=} \big[\, oldFact = \varnothing \,\big] \bullet \\
&\qquad \textstyle\bigwedge c : \{y : Concept \mid y \neq newFact.con \\
&\qquad\qquad\qquad \wedge\ newFact.con \notin y.totalSuperConcept\} \bullet \\
&\qquad\qquad \big[\, con! : Concept \mid con! = c \,\big] \fatsemi newFact.removeType \\
&\qquad \textstyle\bigwedge getNewCon \fatsemi newFact.addType \\
&Execute_2 \mathrel{\widehat=} \big[\, oldFact \neq \varnothing \,\big] \bullet \\
&\qquad \textstyle\bigwedge of : oldFact \bullet (of.getCon \fatsemi newFact.removeType) \\
&\qquad \textstyle\bigwedge getNewCon \fatsemi newFact.addType \\
&Execute \mathrel{\widehat=} \big[\, enable \Leftrightarrow \mathrm{true} \,\big] \bullet (Execute_1 \mathbin{[\!]} Execute_2)
\end{aligned}$$

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

IfRule

$TransitionRule$

$condition : \downarrow Expression;\ rule : TransitionRule$

$enable \Leftrightarrow (condition.V.val \Leftrightarrow \mathrm{true})$

$$Execute \mathrel{\widehat=} \big[\, enable \Leftrightarrow \mathrm{true} \,\big] \bullet rule.Execute$$

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

ChooseRule

TransitionRule

$$variable : Variable;\ condition :\downarrow Expression;\ rule : TransitionRule$$

$$enable \Leftrightarrow (\exists\, v : Value \bullet variable.V = v \Rightarrow condition.V.val \Leftrightarrow \mathrm{true})$$
$$variable \in condition.hasVariables$$

$$\begin{aligned} Execute \mathrel{\widehat=} & \left[\, enable \Leftrightarrow \mathrm{true} \,\right] \bullet \\ & \left[\, v? : Value \;\middle|\; variable.V = v? \Rightarrow condition.V.val \Leftrightarrow \mathrm{true} \,\right] \bullet \\ & v?.outVal \mathbin{⨟} (variable.Assign \bigwedge rule.Execute) \end{aligned}$$

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

ForAllRule

TransitionRule

$$variable : Variable;\ condition :\downarrow Expression;\ rule : TransitionRule$$

$$enable \Leftrightarrow (\exists\, v : Value \bullet variable.V = v \Rightarrow condition.V.val \Leftrightarrow \mathrm{true})$$
$$variable \in condition.hasVariables$$

$$\begin{aligned} Execute \mathrel{\widehat=} & \bigwedge v? : \{v : Value \mid \\ & \qquad variable.V = v \Rightarrow condition.V.val \Leftrightarrow \mathrm{true}\} \bullet \\ & v?.OutVal \mathbin{⨟} (variable.Assign \bigwedge rule.Execute) \end{aligned}$$

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 selected and executed.

PipedRule

TransitionRule

$rules : \mathbb{P}\, TransitionRule$

$rules \neq \varnothing$

$enable \Leftrightarrow (\exists\, rule : rules \bullet rule.enable \Leftrightarrow \mathrm{true})$

SelectRule

$ru? : TransitionRule$

$ru? \in rules \wedge ru?.enable \Leftrightarrow \mathrm{true}$

$Execute \mathrel{\widehat{=}} SelectRule \bullet ru?.Execute$

5.2.4. Service execution

Service

...

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

Δ

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

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

$totalIn = ... \wedge totalOut = ... \wedge totalShared = ...$

......

SelectInterfaces

$inter? : hasInterface$

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

ReadFromEvn

....

WriteToEvn

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

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

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

$$Execute \mathrel{\widehat{=}} ReadFromEvn \mathbin{\S} ($$
$$hasCapability.PreAssSat$$
$$\S\, SelectInterfaces \bullet inter?.Execute$$
$$\S\, hasCapability.PostEffSat$$
$$[]\ hasCapability.NotPreAssSat)$$
$$\S\, WriteToEvn$$

Before we formally define the overall execution of a *Service*, we first introduce a few new attributes to assist the specification. WSMO itself does not include any grounding standards[6]. In our model, the two attributes, *inputVar* and *outputVar*, and the two operations, *AssignVarFromEvn* and *WriteToEvn*, are used to abstract the communication between a service and its environment. However, further details regarding the concrete grounding details fall beyond the scope of this paper. *ReadFromEvn* reads the values from the environment and assigns them to the corresponding variables. If the input value is a WSMO ontology *Instance* or *RelationInstance*, its type must have the role (mode) as *'in'* or *'shared'* and cannot be *'controlled'*. The secondary attributes *totalIn*, *totalOut*, *totalControl* and etc. denote all the *Concept* and *Relation* modes defined by the service's *state signature*. *WriteToEvn* is used to output values to environment. If the output values are either WSMO ontology *Instances* or *RelationInstances*, its type must have the roles (modes) as *'out'* or *'shared'* and cannot be *'controlled'*. To make our model more readable, we assume that before communicating with an environment, all conflicts (if there are any) have been dealt with by *ooMediators*.

The behaviour of a *Service* is defined using the operation *Execute*. Initially, the service gathers requests from users and initialises any necessary variables. If the *preconditions* and *assumptions* are then satisfied, the service will be executed based on the selected *interface* and *postconditions* of the service. After execution, the service will then return any necessary information to the users (i.e. outputs or error messages). If, however, the service is invoked without having satisfied the *preconditions* and *assumptions*, then the behaviour of the service will be undefined. In such cases, tool developers have the freedom to implement their own scenarios to handle this situation, such as terminating the execution and reporting an error message, or simply ignoring it.

6. Case study – Amazon Web services

In this section, we use the WSMO model for Amazon Associates Web service (A2S) as a case study to illustrate how a WSMO model can be represented in OZ using the semantic library we just defined. To the best of our knowledge, the Amazon A2S WSMO description is one of the largest WSMO models being developed so far.

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

Amazon Web services (AWS) provide developers with direct access to Amazon's robust technology platform. By using them, external developers and businesses can build their own applications on AWS in a reliable, flexible, and cost-effective manner. Amazon Web services offer a variety of Web services, such as Amazon Associates Web service (A2S), which exposes Amazon's product data and e-commerce functionality, Amazon Elastic Compute Cloud (Amazon EC2), which provides resizable compute capacity in the cloud, and etc. This paper focused on Amazon A2S.

Amazon Associates Web service (formerly named the Amazon E-Commerce service "ECS") exposes Amazon's product data through an easy-to-use Web services interface that, when combined with the Amazon Associates Program, is a powerful combination for website owners, Web developers, and Amazon sellers to make money. Developers may use the Amazon Associates Web service as long as it's used primarily to drive traffic back to Amazon's websites or sales of Amazon products and services.

The functionality of A2S is defined in a WSDL file, which contains more than 20 operations. These operations support different tasks on Amazon's retail website, including:

- search Amazon products, such as *ItemSearch*, *ItemLookup*, *SimilarityLookup*, *SellerListingSearch* and *SellerListingLookup*.
- create a shopping cart of items and purchase them, such as *CartCreate*, *CartAdd*, *CartModify*, *CartClear* and *CartGet*.
- look up publicly available customer content (reviews, wish lists and Listmania lists), such as *CustomerContentSearch*, *CustomerContentLookup*, *ListLookup* and *ListSearch*.
- look up feedback on specific sellers and get seller product listings, such as *SellerLookup*.
- other support operations, such as *Help* which returns information about how to use an A2S operation or about what to expect from A2S response groups.

The full Amazon A2S WSMO model can be found in [31].

6.1. WSMO ontology for A2S

This section shows how the WSMO ontologies used for the Amazon service datatypes can be modelled in Object-Z using the formal model defined in the previous sections.

Any concrete WSMO elements are modelled as Object-Z instances. For example, the following shows the WSMO definition of *HelpRequest* concept and instances of this concept are used as input for the *help* defined in Amazon A2S framework. *HelpRequest* concept has three '*ofType*' kind attributes, i.e., *about*, *hasType* and *responseGroup* and their types are all String.

```
concept helpRequest
    about ofType _string
    helpType ofType _string
   responseGroup ofType _string
```

They are modelled in OZ as follows, where *String* is a Z given type.

$$\begin{array}{|l}
about, helpType, responseGroup : Attribute \\
helpRequest : Concept \\
\hline
about.hasType = String \land about.hasTypeModel = ofType \\
helpType.hasType = String \land helpType.hasTypeModel = ofType \\
responseGroup.hasType = String \land responseGroup.hasTypeModel = ofType \\
helpRequest.hasAttribute = \{about, helpType, responseGroup\} \\
\ldots.
\end{array}$$

The following WSMO axiom limits the type of *help* to be 'Operation' and 'ResponseGroup' only, which is modelled in OZ as well.

```
        axiom helpTypeConstraint //The type of help can only be 'Operation' or
'ResponseGroup'
          definedBy !-
            ?x memberof helpRequest and
            ?x[helpType hasValue ?type] and
            (?type hasValue "Operation" or
             ?type hasValue "ResponseGroup")
```

$$
\begin{array}{|l}
\mathit{Operation}, \mathit{ResponseGroup} : \mathit{String} \\
\mathit{helpTypeConstraintX} : \mathit{Variable} \\
\mathit{helpTypeConstraintExp}1 : \mathit{MemberOfExp} \\
\mathit{helpTypeConstraintExp}2, \mathit{helpTypeConstraintExp}3 : \mathit{hasValueExp} \\
\mathit{helpTypeConstraintExp}4 : \mathit{OrExp} \\
\mathit{helpTypeConstraint} : \mathit{RightImpExp} \\
\hline
\mathit{helpTypeConstraintExp}1.\mathit{con} = \mathit{helpRequest} \\
\mathit{helpTypeConstraintExp}1.\mathit{ins} = \mathit{helpTypeConstraintX} \\
\mathit{helpTypeConstraintExp}2.\mathit{ins} = \mathit{helpTypeConstraintX} \\
\mathit{helpTypeConstraintExp}2.\mathit{att} = \mathit{helpType} \\
\mathit{helpTypeConstraintExp}2.\mathit{value} = \mathit{Operation} \\
\mathit{helpTypeConstraintExp}3.\mathit{ins} = \mathit{helpTypeConstraintX} \\
\mathit{helpTypeConstraintExp}3.\mathit{att} = \mathit{helpType} \\
\mathit{helpTypeConstraintExp}3.\mathit{value} = \mathit{ResponseGroup} \\
\mathit{helpTypeConstraintExp}4.\mathit{operand} = \{\mathit{helpTypeConstraintExp}2, \mathit{helpTypeConstraintExp}3\} \\
\mathit{helpTypeConstraint}.\mathit{left} = \mathit{helpTypeConstraintExp}1 \\
\mathit{helpTypeConstraint}.\mathit{right} = \mathit{helpTypeConstraintExp}4
\end{array}
$$

All other A2S WSMO ontology concepts, such as *HelpResponse*, *itemSearchRequest*, *ItemSearchResponse* and etc. can be similarly defined.

6.2. WSMO capability of the Amazon services

The functionality of the Amazon A2S allows clients to search or browse Amazon's product catalog; to retrieve detailed product information, reviews, and images; and to interface with customer shopping carts, and so on. The WSMO *capability* of A2S includes that the client can get several types of information and/or that the client can create and manipulate the content of shopping charts. In particular, the *capability* is defined as follows. The precondition and input is that a client wants to get information about Amazon products or purchase some products (and has some data that distinguishes the products). The postcondition and output of the A2S is that the client has information about the Amazon products and/or a shopping cart ready for manual completion of the purchase [31].

The WSMO model of A2S follows Amazon's decision to put all five Web services in a single big Web service description. We model each of the components in details.

The precondition of *amazonWS* captures all the inputs of the Amazon Web service. Amazon can accept any of the inputs in a given moment and hence the precondition is defined in terms of a disjunction of all the possible inputs. The only shared variable is ?*request* which is required in order to capture what post-condition will take effect given a particular input.

```
webService amazonWS /*WSMO definition*/
  capability amazonWSCapability
    sharedVariables {?request}
    precondition
     definedBy
       //Request for a help topic
       ?request memberOf helpRequest or
       ... ....
       //Search for a listing by an identifier
       ?request memberOf sellerListingLookupRequest.
```

This can be modelled in OZ as following:

$$amazonWS : Service$$

$$\begin{array}{|l}
amazonWSCapability : Capability;\ request : Variable \\
capabilityPreExp1, ..., capabilityPreExp15 : MemberOfExp; \\
capabilityPrecondition : OrExp \\
\hline
amazonWS.hasCapablity = amazonWSCapability \\
capabilityPreExp1.con = helpRequest \\
capabilityPreExp1.ins = request \\
.... \\
capabilityPreExp15.con = sellerListingLookupRequest \\
capabilityPreExp15.ins = request \\
capabilityPrecondition.operand = \{capabilityPreExp1, ..., capabilityPreExp15\} \\
.... \\
amazonWSCapability.hasSharedVariable = \{request\} \\
amazonWSCapability.hasPrecondition = \{capabilityPrecondition\}
\end{array}$$

The post-condition, which takes the form of implication rules, means that given a particular pre-condition, one particular post-condition will take effect. There is one postcondition for each A2S operation. For example, for the *Help Request* operation, the following post-condition shows that for given a request for a particular type of help, a response with the same type is returned. In Amazon, the type of help can be either for an operation or a response group.

```
postcondition
   definedBy
      (?request[helpType hasValue ?type] memberOf
           helpRequest implies
      exists ?response (?response[responseType hasValue ?type]
           memberOf helpResponse)
      ) and
        ... ...
```

This post-condition can be modelled in OZ as:

$$
\begin{array}{|l}
\mathit{capabilityHelpReqPostExp}1 : \mathit{MemberOfExp} \\
\mathit{capabilityHelpReqPostExp}2 : \mathit{hasValueExp} \\
\mathit{typeCapabilityHelpReqPost} : \mathit{Variable};\ \mathit{response} : \mathit{Variable} \\
\mathit{capabilityHelpReqPostExp}3 : \mathit{ExistExp} \\
\mathit{capabilityHelpReqPostExp}4 : \mathit{MemberOfExp} \\
\mathit{capabilityHelpReqPostExp}5 : \mathit{hasValueExp} \\
\mathit{capabilityHelpReqPostExp}6 : \mathit{AndExp} \\
\mathit{capabilityHelpReqPostExp}7 : \mathit{AndExp} \\
\mathit{capabilityHelpReqPostcondition}1 : \mathit{LeftImpExp} \\
\hline
\mathit{capabilityHelpReqPostExp}1.\mathit{con} = \mathit{helpRequest} \\
\mathit{capabilityHelpReqPostExp}1.\mathit{ins} = \mathit{request} \\
\mathit{capabilityHelpReqPostExp}2.\mathit{ins} = \mathit{request} \\
\mathit{capabilityHelpReqPostExp}2.\mathit{att} = \mathit{helpType} \\
\mathit{capabilityHelpReqPostExp}2.\mathit{value} = \mathit{typeCapabilityHelpReqPost} \\
\mathit{capabilityHelpReqPostExp}4.\mathit{con} = \mathit{helpResponse} \\
\mathit{capabilityHelpReqPostExp}4.\mathit{ins} = \mathit{response} \\
\mathit{capabilityHelpReqPostExp}5.\mathit{ins} = \mathit{response} \\
\mathit{capabilityHelpReqPostExp}5.\mathit{att} = \mathit{reponseType} \\
\mathit{capabilityHelpReqPostExp}5.\mathit{value} = \mathit{typeCapabilityHelpReqPost} \\
\mathit{capabilityHelpReqPostExp}6.\mathit{operands} = \\
\quad \{\mathit{capabilityHelpReqPostExp}4, \mathit{capabilityHelpReqPostExp}5\} \\
\mathit{capabilityHelpReqPostExp}3.\mathit{var} = \mathit{response} \\
\mathit{capabilityHelpReqPostExp}3.\mathit{operand} = \mathit{capabilityHelpReqPostExp}6 \\
\mathit{capabilityHelpReqPostExp}7.\mathit{operands} = \\
\quad \{\mathit{capabilityHelpReqPostExp}1, \mathit{capabilityHelpReqPostExp}2\} \\
\mathit{capabilityHelpReqPostcondition}1.\mathit{left} = \mathit{capabilityHelpReqPostExp}7 \\
\mathit{capabilityHelpReqPostcondition}1.\mathit{right} = \mathit{capabilityHelpReqPostExp}3
\end{array}
$$

Similarly, the post-conditions for other operations can be modelled as well.

$$amazonWSCapability.hasPostcondition = \{capabilityHelpReqPostcondition1, ...\}$$

6.3. WSMO Choreography for Amazon A2S

This section describes the interface part of the WSMO Amazon service, which starts with the state signature (defined as follows).

```
interface amazonWSInterface
  choreography
   stateSignature
    in
      concept helpRequest
      ... ....
      concept sellerListingLookupRequest
    out
      concept helpResponse
      ... ....
```

This can be represented in OZ as following.

$$\begin{array}{|l}
amazonWSInterface : Interface \\
amazonWSInterfaceChoreography : Choreography \\
amazonWSInterfaceStateSignature : StateSignature \\
\hline
amazonWS.hasInterface = \{amazonWSInterface\} \\
amazonWSInterface.hasChoreography = \{amazonWSInterfaceChoreography\} \\
amazonWSInterfaceChoreography.hasStateSignature = \\
\quad amazonWSInterfaceStateSignature \\
amazonWSInterfaceStateSignature.hasIn = \{helpRequest, ...\} \\
amazonWSInterfaceStateSignature.hasOut = \{helpResponse, ...\}
\end{array}$$

For the WSMO choreography of the A2S, a single transition rule is modelled for each of the service's operations. For example, the following transition rule is for the "Help" operation.

```
if (?HelpRequest memberOf helpRequest) then
    add(_# memberOf helpResponse)
endIf
```

It is can be modelled in OZ as the following.

$$
\begin{array}{|l}
\mathit{amazonWSHelpTransitionRule} : \mathit{IfRule} \\
\mathit{amazonWSHelpTransitionRuleCondition} : \mathit{MemberOfExp} \\
\mathit{amazonWSHelpTransitionRuleRule} : \mathit{AddMemberOf} \\
\mathit{amazonWSHelpTransitionRuleHelpV1} : \mathit{Variable} \\
\mathit{amazonWSHelpTransitionRuleRuleFact} : \mathit{MemberOfExp} \\
\mathit{amazonWSHelpTransitionRuleHelpV2} : \mathit{Variable} \\
\hline
\mathit{amazonWSHelpTransitionRule.condition} = \\
\quad \mathit{amazonWSHelpTransitionRuleCondition} \\
\mathit{amazonWSHelpTransitionRule.rule} = \mathit{amazonWSHelpTransitionRuleRule} \\
\mathit{amazonWSHelpTransitionRuleCondition.con} = \mathit{helpRequest} \\
\mathit{amazonWSHelpTransitionRuleCondition.ins} = \\
\quad \mathit{amazonWSHelpTransitionRuleHelpV1} \\
\mathit{amazonWSHelpTransitionRuleRule.fact} = \mathit{amazonWSHelpTransitionRuleRuleFact} \\
\mathit{amazonWSHelpTransitionRuleRuleFact.ins} = \mathit{amazonWSHelpTransitionRuleHelpV2} \\
\mathit{amazonWSHelpTransitionRuleRuleFact.con} = \mathit{helpResponse} \\
\ldots\ldots \\
\mathit{amazonWSInterfaceChoreography.hasTransitionRules} = \\
\quad \{\mathit{amazonWSHelpTransitionRule}, ...\}
\end{array}
$$

Other transition rules can be defined as well. After all the WSMO A2S model is defined in OZ, OZ tools can be used to check the consistence of the model as discussed in the next section.

7. Discussion

The formal specification of WSMO provides a single, canonical and unambiguous specification of the WSMO framework, which can be beneficial to the Semantic Web Service community in many different ways. In this section, we discuss several of these advantages and present relevant examples.

7.1. Checking the consistency of the 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, the current WSMO A2S model defines a concept *helpRequest* which has an attribute

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.

ResponseGroup, where the range of this attribute is of data type *string* [7]. This WSMO definition can be translated into Object-Z as follows:

$$ResponseGroup : Attribute;\ helpRequest : Concept$$

$$ResponseGroup.hasType = String$$

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

Note that the translation from WSMO to Object-Z can be automatically realised by a tool. However, when we load our formal WSMO model and the above Object-Z definition into an Object-Z type checker, the type checker complains that there is a type error. After studying this problem, we realize that the problem is due to the fact that according to the WSMO documents (Section 4.2.2), the *hasType* attribute defined for a concept attribute can only have a WSMO *Concept* as its value. As *String* is a subclass of literal datatype *Value*, and many Semantic Web languages consider this to be disjointed with *Concept*, this results in a type violation. Thus, to resolve this violation, we would propose that the WSMO standard should be revised as illustrated in Figure 2.

7.2. Making the WSMO language precise and removing ambiguity

Large sections of the existing 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

[7] 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/`

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 use the language correctly; the test suite covers all important rules implied by the language; and the tools developed work correctly and consistently.

7.3. 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 [29, 32] and AA [30] separately to reasoning over Web ontologies. In Section 7.1, 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.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotExp}} \\
\hline
\textit{Expression} \\
\\
\begin{array}{|l}
\hline
\textit{operand} : \downarrow \textit{Expression} \\
\hline
\textit{hasVariables} = \textit{operand}.\textit{hasVariables} \\
\textit{usedTerms} = \textit{operand}.\textit{usedTerms} \\
\textit{operand}.V \in \textit{Nil} \Rightarrow V \in \textit{Nil} \\
\textit{operand}.V \notin \textit{Nil} \Rightarrow V.\textit{val} \Leftrightarrow \neg\ \textit{operand}.V.\textit{val} \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrchestrationRule}} \\
\hline
\textit{TransitionRule} \\
\\
\begin{array}{|l}
\hline
\textit{target} : \textit{wgMediator} \cup \textit{Service} \\
\ldots\ldots \\
\hline
\textit{enable} = \ldots\ldots \\
\hline
\end{array} \\
\textit{Execute} \mathrel{\widehat=} \ldots\ldots. \\
\hline
\end{array}$$

7.4. 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 we want to add a new kind of logic expression, *NotExp*, to WSMO core. Then in our model it is necessary to add only the *NotExp* class (defined above). Alternatively, consider the case when the development of WSMO *Orchestration* completes and *Orchestration* is clearly defined. Then in our model it is necessary to augment the model with a class such as the *OrchestrationRule* class (defined above) to include this mechanism. The introduction of these extensions does not involve any changes to the classes defined in the previous sections. Validation tools can then be used to confirm the validity of the extended model.

8. Conclusion

WSMO is one of the most important technologies within the field of Semantic Web Service research. This paper has presented an Object-Z semantics model for WSMO, whereby the WSMO constructs are modelled 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. This Object-Z specification provides an invaluable adjunct to the current WSMO documentation and specifications, and will support further development, validation and verification of WSMO (and the corresponding tools) as it evolves. This Object-Z specification of WSMO is not a new Semantic Web service formalism. Rather than replacing WSMO model, our work complements the existing WSMO specification by adding rigorous, precise and mathematical foundations to WSMO. It helps users to understand and use WSMO more easily.

In our previous work [19, 20], OZ has also been used to specify another significant SWS alternative – the OWL-S language. One of the future works is to formally compare the two approaches and identify possible integration and translation between the two languages.

References

[1] C. Bussler, B2B Protocol Standards and their Role in Semantic B2B Integration Engines, Bulletin of the Technical Committee on Data Engineering 24 (1).

[2] S. A. McIlraith, T. C. Son, H. Zeng, Semantic web services, IEEE Intelligent Systems 16 (2) (2001) 46–53. doi:http://dx.doi.org/10.1109/5254.920599.

[3] T. Payne, O. Lassila, Guest editors' introduction: Semantic web services, IEEE Intelligent Systems 19 (4) (2004) 14–15. doi:http://dx.doi.org/10.1109/MIS.2004.29.

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

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

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

[7] S. K. Kim, D. Carrington, Formalizing UML Class Diagram Using Object-Z, in: R. France, B. Rumpe (Eds.), UML'99, Lect. Notes in Comput. Sci., Springer-Verlag, 1999.

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

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

[10] Object Management Group, Meta object facility (MOF) specification, `http://www.omg.org` (2002).
URL `\url{http://www.omg.org/technology/documents/formal/mof.htm}`

[11] J. Bruijn, H. Lausen, A. Polleres, D. Fensel, The web service modelling language wsml: An overview, in: Proceedings of the 3rd European Semantic Web Conference, Springer-Verlag, Budva, Montenegro, 2006, pp. 590–604.

[12] M. Kifer, G. Lausen, J. Wu, Logical foundations of object oriented and frame based languages, Journal of ACM 42 (1995) 741–843.

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

[14] A. Griffiths, G. Rose, A Semantic Foundation for Object Identity in Formal Specification, Object-Oriented Systems 2 (Chapman & Hall 1995) 195–215.

[15] G. Smith, Extending W for Object-Z, in: J. P. Bowen, M. G. Hinchey (Eds.), Proceedings of the 9th Annual Z-User Meeting, Springer-Verlag, 1995, pp. 276–295.

[16] J. Woodcock, S. Brien, W : A logic for Z, in: Proceedings of Sixth Annual Z-User Meeting, University of York, 1991.

[17] J. S. Dong, R. Duke, Class Union and Polymorphism, in: C. Mingins, W. Haebich, J. Potter, B. Meyer (Eds.), Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12, Prentice-Hall, 1993, pp. 181–190.

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

[19] H. H. Wang, A. Saleh, T. Payne, N. Gibbins, Formal specification of owl-s with object-z: the static aspect, in: The 2007 IEEE/WIC/ACM International Conference on Web Intelligence, Silicon Valley, USA, 2007.

[20] H. H. Wang, T. Payne, N. Gibbins, A. Saleh, Formal specification of owl-s with object-z: the dynamic aspect, in: The 8th International Conference on Web Information Systems Engineering, Springer, 2007.
URL `http://eprints.ecs.soton.ac.uk/14395/`

[21] T. R. Gruber, A translation approach to portable ontology specifications, Knowl. Acquis. 5 (2) (1993) 199–220. doi:http://dx.doi.org/10.1006/knac.1993.1008.

[22] J. S. Dong, G. Rose, R. Duke, The Role of Secondary Attributes in Formal Object Modelling, Tech. Rep. 95-20, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia (1995).

[23] A. Rector, Modularisation of domain ontologies implemented in description logics and related formalisms including owl, in: J. Genari (Ed.), Knowledge Capture 2003, ACM, Sanibel Island, FL, 2003, pp. 121–128.

[24] P. Doran, V. A. M. Tamma, L. Iannone, Ontology module extraction for ontology reuse: an ontology engineering perspective, in: M. J. Silva, A. H. F. Laender, R. A. Baeza-Yates, D. L. McGuinness, B. Olstad, O. H. Olsen, A. O. Falcão (Eds.), CIKM, ACM, 2007, pp. 61–70.

[25] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.

[26] M. G. Nanda, S. Chandra, V. Sarkar, Decentralizing execution of composite web services, in: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, New York, NY, USA, 2004, pp. 170–187. doi:http://doi.acm.org/10.1145/1028976.1028991.

[27] C. Wroe, C. Goble, M. Greenwood, P. Lord, S. Miles, J. Papay, T. Payne, L. Moreau, Automating experiments using semantic data on a bioinformatics grid, IEEE Intelligent Systems 19 (1) (2004) 48–55. doi:http://doi.ieeecomputersociety.org/10.1109/MIS.2004.1265885.

[28] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes I and II, Information and Computation 100 (1992) 1 – 41, 42 – 78.

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

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

[31] K. Jacek, R. Dumitru, S. James, Wsmo use case: Amazon e-commerce service, unpublished manuscript (2006).

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