

Formal Specification of OWL-S with Object-Z

Hai H. Wang Ahmed Saleh Terry Payne Nick Gibbins

School of Electronics and Computer Science ,
University of Southampton, Southampton SO17 1BJ, UK
{hw, amms, trp, nmg}@ecs.soton.ac.uk

Abstract. Semantic Web Services, one of the most significant research areas within the Semantic Web vision, has been recognized as a promising technology that exhibits huge commercial potential, and attracts a great deal of attention from both the research community and Industry. OWL-S, one of the most significant Semantic Web Service ontologies proposed to date, provides Web Service providers with a core ontological framework and guidelines for describing the properties and capabilities of their Web Services in unambiguous, computer-interpretable form. To support standardization and tool support of OWL-S, a formal semantics of the language is highly desirable. In this paper, we present a formal Object-Z semantics of OWL-S. 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 the paper, can be used to identify and eliminate errors in the current documentation.

1 Introduction

The Semantic Web represents a paradigmatic shift from the notion of distributed *documents*, interconnected through embedded hyperlinks and presented seamlessly through human-oriented browsers, to a distributed web of interlinked resources, whose meanings are defined by a plethora of ontologies. This evolution of the web allows services, hitherto intended for human consumption through form-based web pages or through declarative specifications for software developers, to be described in a *machine-understandable* form, thus facilitating automated 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, and has been recognized as a promising technology that exhibits huge commercial potential.

The standards-based approach is 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. 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 [1, 16]. OWL-S, as an OWL-based Web Service Ontology, is one of the most significant

Semantic Web Service framework proposed to date [1]. It provides Web Service developers with a core ontological model which can be used for describing the domain-independent properties and capabilities of their Web services in an unambiguous and computer-intepretable form. Following the layered approach to markup language development, the current version of OWL-S builds on top of OWL.

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 OWL-S are defined in terms of its metamodel – as a set of OWL ontologies complemented with supporting documentation in English¹. Although OWL (more precisely, the OWL-DL portion of OWL) has a well-defined formal meaning (despite being only a small fragment of FOL), OWL lacks the expressivity to define all the desired properties of OWL-S. Despite its aim of providing an unambiguous model for defining services, OWL-S users and tool developers have had to rely heavily on English-language documentation and comments to understand the language and interpret its models. However, this use of natural language is ambiguous and can be interpreted in different ways. This lack of precision in defining the semantics of OWL-S can result in different usage, as Web service providers and tool developers may have a different interpretation and understanding of the same OWL-S model. Another major problem with the current OWL-S definition is that the three components of OWL-S (i.e. the *profile*, *process model* and *service grounding*) have been separately described in various formats (such as natural language, Ontologies and Petri Nets etc.). These different descriptions contain redundancy and sometimes contradiction in the information provided. Furthermore, with the continuous evolution of OWL-S, it has been very difficult to consistently extend and revise these separate descriptions. To support a common understanding, and facilitate standardization for OWL-S, a formal semantics of its language is highly desirable. Also, if we are to reason about OWL-S and ultimately formally verify, or even adequately test the correctness of a Semantic Web Service system, we need to have a precise specification of the OWL-S model. In addition, being a relatively young field, research into Semantic Web services and OWL-S is still ongoing, and therefore a semantic representation of OWL-S needs to be reusable and extendable in a way that can accommodate this evolutionary process.

In this paper, we present a formal denotational model of OWL-S [1] using the Object-Z (OZ) specification language [10]. A denotational approach has been proved to be one of the most effective ways of defining the semantics of a language, and has been used to define the formal semantics for many programming and modeling languages [12, 19]. Object-Z has been used to provide one single formal model for the syntax, the static semantics and the dynamic semantics of OWL-S. Also, because these different aspects have been described within a single framework, the consistency between these aspects can be easily maintained. In this paper, we focus on the formal model for the syntax and static semantics of OWL-S. The dynamic semantics of OWL-S will be discussed in a future paper.

¹ There has been some published work on the definition of the dynamic semantics of OWL-S [15], as well as its precursor [2].

Object-Z [10] is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. We chose Object-Z over other formalisms to specify OWL-S 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 [17] of Object-Z are closely related to Z standard work [21]. Object-Z also has a fully abstract semantics [18].
- 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 rather than focusing on the resulting model.
- In our previous work [20], OZ has been used to specify the Semantic Web Service Ontology (WSMO) 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.

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

2 Background

2.1 OWL-S

OWL-S [1], formally known as DAML-S, originated from a need to define Web Services or Agent capabilities in such a way that was semantically meaningful (within an open environment), and also to facilitate meaningful message exchange between peers. Essentially, OWL-S provides a service model based on which an abstract description of a service can be provided. It is an upper ontology whose root class is the Service class, that directly corresponds to the actual service that is described semantically (every service that is described maps onto an instance of this concept). The upper level Service class is associated with three other classes: *ServiceProfile*, *ServiceModel* and *ServiceGrounding*. In detail, the OWL-S *ServiceProfile* describes *what the service does*. Thus, the class SERVICE *presents* a *ServiceProfile*. The *service profile* is the primary construct by which a service is advertised, discovered and selected. The OWL-S *ServiceModel* tells *how the service works*. Thus, the class SERVICE is *describedBy* a *ServiceModel*. It includes information about the service inputs, outputs, preconditions and effects (IOPE). It also shows the component processes for a complex process and

how the control flows between the components. The OWL-S *grounding* tells *how the service is used*. It specifies how an agent can pragmatically access a service.

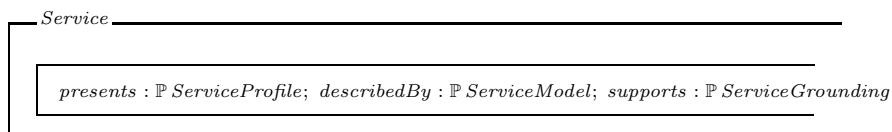
2.2 Object-Z (OZ)

Object-Z [10] is an extension of the Z formal specification language to accommodate object orientation. The essential extension to Z in Object-Z is the *class* construct, which groups the definition of a state schema with the definitions of its associated operations. A class is a template for *objects* of that class: the states of each object are instances of the state schema of the class, and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class. Operation schemas have a Δ -list of those attributes whose values may change. By convention, no Δ -list means that no attribute changes value. The standard behavioral interpretation of Object-Z objects is as a transition system [18]. 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 OWL-S

In this section, we present an Object-Z semantics of OWL-S. In this semantic model, all the different aspects of the language; syntax (an OWL-S model is well-formed), static semantics (an OWL-S model is meaningful) and dynamic semantics (how is an OWL-S model interpreted and executed), are defined in one single unified framework, so that the semantics of the language can be more consistently defined and revised as the language evolves. The OWL-S 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 defined as *class invariant* are used to capture the static semantics of the language. The class operations are used to define OWL-S's dynamic semantics, which describe how the state of a Web service changes. This paper focuses on the first two aspects of OWL-S, i.e. the formal model of syntax and static semantics². Because of the limited space, we only present a partial model here, based on the latest version of OWL-S (1.2 beta).

3.1 Upper level model



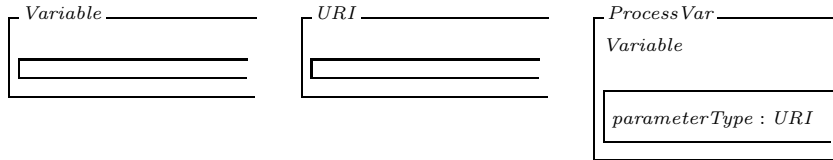
Service denotes the OWL-S service and it has three attributes *presents*, *supports* and *describedBy*, which denotes three essential types of knowledge of an OWL-S service – the profile, the process model and the grounding. The OWL-S profile describes **what the service does**. Thus, a *Service* **presents** *ServiceProfiles*. *ServiceProfile*, *ServiceModel* and *ServiceGrounding* will be formally defined later.

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

3.2 ServiceModel – Modeling Services as Processes

An OWL-S service can be viewed as a process from an interaction point of view. The OWL-S process model is intended to provide a basis for specifying the behaviors of a wide array of services. Before we present the formal definition of *Process*, we first show some necessary elements for defining *Process*.

Parameters and Expressions



The class *Variable* denotes the generic variable type, while *ProcessVar*, defined as a subclass of *Variable* denotes the variables used in an OWL-S service process. It has an attribute *parameterType* denoting the type of the variable which is specified using a URI. We use the Object-Z class *URI* to denote all possible URI address. Due to the limited space, we only provide an abstract view of the class *URI* without any attributes. These concepts can be modeled in more details, e.g., a *URI* reference can be expressed by a qualified name, etc.

$$\left| \begin{array}{l} \textit{Participant}, \textit{Parameter}, \textit{Existential}, \textit{ResultVar}, \textit{Local} : \mathbb{P} \textit{ProcessVar} \\ \hline \textit{Parameter} \cup \textit{Existential} \cup \textit{Participant} \cup \textit{ResultVar} \cup \textit{Local} = \textit{ProcessVar} \end{array} \right.$$

In an OWL-S process model, there are five distinct kinds of *ProcessVar*, as *Parameter*, *Existential*, *Participant*, *ResultVar* and *Local*.

$$\left| \begin{array}{l} (\textit{Parameter} \cup \textit{Existential} \cup \textit{ResultVar}) \\ \cap \textit{Participant} = \emptyset \end{array} \right. \quad \left| \begin{array}{l} \textit{TheClient}, \textit{TheServer} : \textit{Participant} \\ \hline \textit{TheClient} \neq \textit{TheServer} \end{array} \right.$$

A *Participant* is a variable used to denote an agent involved in a process. It is disjoint with *Parameter*, *Existential* and *ResultVar*. There are two predefined participant agents. One is *TheClient*, representing the agent from whose point of view the process is described, and another is *TheServer*, representing the principal element of the service that the client deals with. *TheClient* and *TheServer* are modeled as instances of *Participant*.

$$\left| \begin{array}{l} \textit{Input}, \textit{Output} : \mathbb{P} \textit{Parameter} \\ \hline \textit{Input} \cup \textit{Output} = \textit{Parameter} \wedge \textit{Input} \cap \textit{Output} = \emptyset \end{array} \right.$$

Parameter is the disjoint union of *Input* and *Output* and is used to denote the data transformation produced by the process.

$$\left| \begin{array}{l} \textit{Existential} \cap \textit{Parameter} = \emptyset \wedge \textit{Existential} \cap \textit{ResultVar} = \emptyset \\ \forall a : \textit{Existential} \bullet \exists e : \textit{Expression} \bullet a \in e.\textit{usedVars} \wedge \exists p : \textit{Process} \bullet e \in p.\textit{hasPrecondition} \end{array} \right.$$

An *Existential* is a variable whose value will be bound in a process prediction to make it true and this value of the existential so obtained can appear in the effects of Results, and, if the Process is composite, can be referred to in its body. The above axiom also ensures that an *Existential* will be bound in some *preconditions*. *Expression* and *Process* will be defined later.

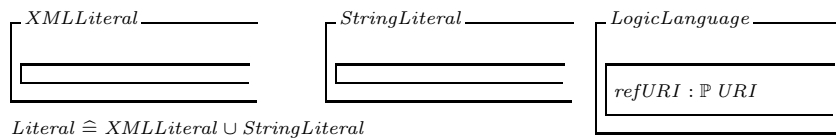
$$\left| \begin{array}{l} \text{ResultVar} \cap \text{Parameter} = \emptyset \\ \forall rv : \text{ResultVar} \bullet \exists e : \text{Expression} \bullet rv \in e.\text{usedVars} \wedge \exists r : \text{Result} \bullet e \in r.\text{inCondition} \end{array} \right.$$

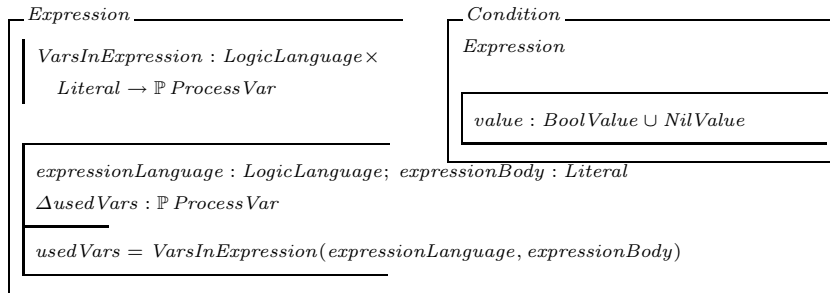
ResultVars are analogous to *Existentials*. Whereas *Existentials* are variables to be bound in preconditions and then used in the specifying result conditions, outputs and effects, *ResultVars* are scoped to a particular result, are bound in the result's condition (*inCondition*), and are used to describe the outputs and effects associated with that condition. The above axiom ensures that an *ResultVars* will be bound. *Result* will be defined later.

Local is the disjoint union of *Loc* and *Link* and it used for intermediate results in a composite process. The detail of the usage will be discussed later when we define the data flow in OWL-S.

$$\left| \begin{array}{l} (\text{Participant} \cup \text{Existential} \cup \text{ResultVar}) \\ \cap \text{Local} = \emptyset \end{array} \right. \left| \begin{array}{l} \text{Loc}, \text{Link} : \mathbb{P} \text{Local} \\ \hline \text{Loc} \cup \text{Link} = \text{Local} \wedge \text{Loc} \cap \text{Link} = \emptyset \end{array} \right.$$

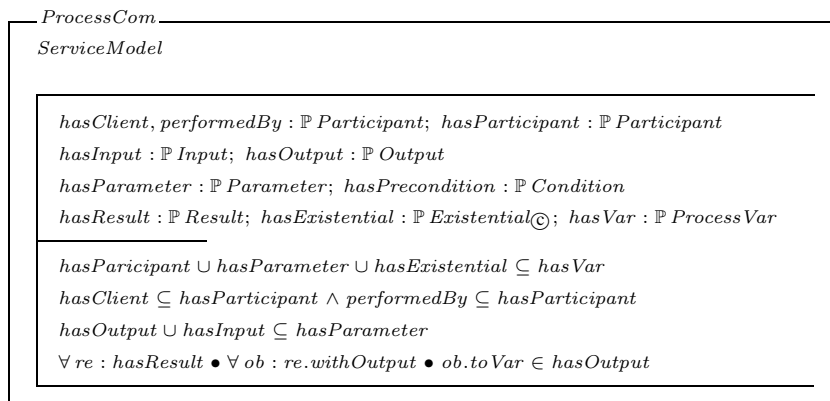
OWL-S itself does not provide any logics to define formulas and expressions. It defines a flexible framework which allows users to choose various logic languages to describe services. The key idea is that in OWL-S, expressions are treated as literals (either string or XML). Due to the space limitation, we abstract the XML literals and string literals as Object-Z classes. A complete OZ model of XML documents was presented in [13]. The class *Expression* has the attribute *expressionLanguage* to denote which *LogicLanguage* is used to express the expression. *LogicLanguage* is referenced by a URI and the attribute *expressionBody* to denote the actual literal expression. We also define a secondary attribute [8] *usedVars* to denote the variables used in *expressionBody*. Depending on different languages used to describe an expression, there are different ways to retrieve the variables from the expression. We abstract this as the function *VarsInExpression*, that is given an expression and a logic language name, *VarsInExpression* returns the set of variables used in the expression. The secondary attribute means that the value of *usedVars* depends on the values of some other attributes such as *expressionLanguage* and *expressionBody* or around environment.





Condition is a subclass of *Expression* and it has a determined truth value. We also define a special kind of value called the *nil* value. This is used when variables are not bound to any concrete values.

Processes

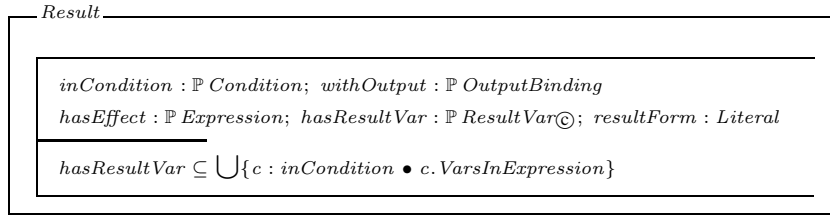


OWL-S defines three different kinds of processes – atomic process, composite process and simple process. Before formally defining each one in detail, we first specify some of the common attributes of an OWL-S process.

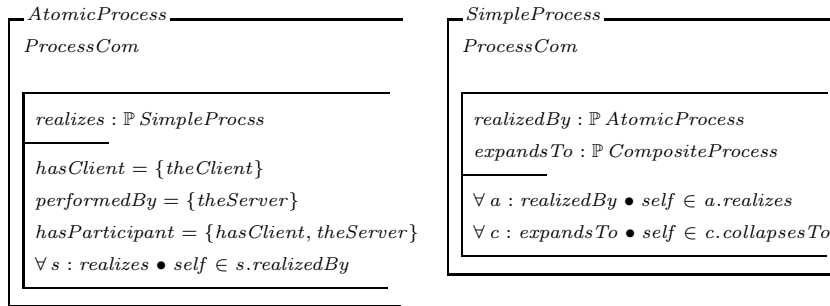
The attribute *hasClient* denotes those agents from whose point of views the process is described and *performedBy* denotes those elements of the service that the client deals with. *HasParticipant* represents all the parts involved in the process. *HasClient* and *performedBy* are subsets of *hasParticipant*. *HasInput* denotes the set of data required by the process for execution. *HasOutput* defines the information provided back by the process to the requester. *HasPrecondition* denotes the set of *conditions* that has to be satisfied for the process to perform successfully. *HasVar* denotes all the variables used in a process. *HasResult* specifies the *results* of the service. The last predicate in the invariant ensures that all the output variables used in *hasResult* are declared in the process.

Result in OWL-S model specifies under what conditions the outputs are generated as well as what domain changes are produced during the execution of the service.

InCondition denotes the conditions under which the result occurs. *withOutput* denotes the output bindings of output parameter of the process to a value form. *hasEffect* is a set of expressions that captures possible effects to the context. *HasResultVar* declares variables that bound in the *inCondition*. Furthermore, these variables are scoped to this particular result. The containment notation ‘ \textcircled{C} ’ in Object-Z [5] can ensure this. *OutputBinding* is a subset of *Binding* (which will be defined later) with *toVar* as an *Output*.



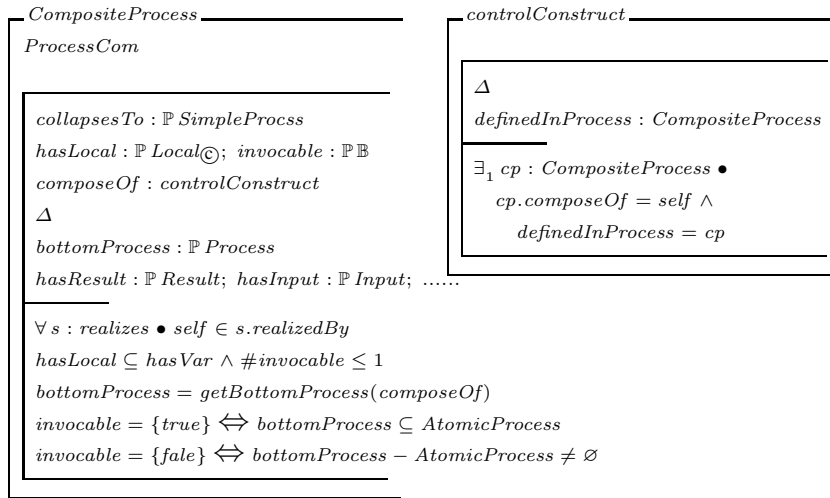
AtomicProcess is a kind of *Process* denoting the actions a service can perform by engaging in a single interaction. The invariant shows that an atomic process has the client as *theClient* and is performed by *theServer*; and that only those two participants exists.



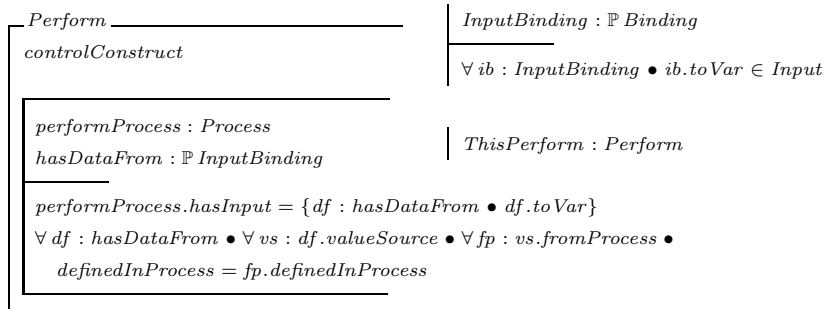
SimpleProcess is another kind of *Process*. It is mainly used either to provide a view of (a specialized way of using) some atomic processes, or a simplified representation of some composite process (for purposes of planning and reasoning).

Composite processes are processes which are decomposable into other (non-composite or composite) processes and are modeled as *CompositeProcess*. A *CompositeProcess* can *collapsesTo* some *SimpleProcesses*. *HasLocal* defines a set of *local* variables used in the ‘Producer-Push’ data flow pattern used in a composite processes. We will define this pattern later. The attribute *invocable* is an optional boolean value which is used to tell whether the *CompositeProcess* bottoms out in atomic processes. A *CompositeProcess* must have a *composedOf* attribute by which it indicates the control structure of the composite, using a *ControlConstruct*. The secondary attribute *bottomProcess* denotes the set of bottom processes in a *CompositeProcess*. Given a *controlConstruct*, the function *getBottomProcess* will return the set of bottom processes. The detail defini-

tion of *getBottomProcess* is omitted here.



Each control construct, in turn, could associate with an additional property called *components* to indicate the nested control constructs from which it is composed, and, in some cases, their ordering. The secondary attribute *definedInProcess* defined in *controlConstruct* denotes the composite process the *controlConstruct* belonging to.

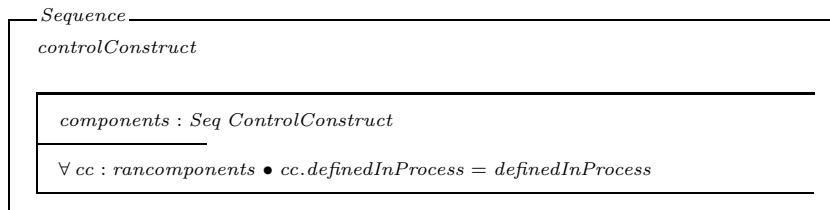


Perform is a special kind of *ControlConstruct* used to refer a process to a composite process. It has two attributes – *performProcess* which denotes the process to be invoked and *hasDataFrom* which denotes the necessary input bindings for the invoked process, where *InputBinding* is a subset of *Binding* with *toVar* to *Input*. The invariant of *Perform* ensures that all the inputs of *performProcess* are provided and if the input is derived from onther process then it can only be derived from the parameters of another *Performs* in the *same* composite process. OWL-S also introduces a standard variable, *ThisPerform*, used to refer, at runtime, to the execution instance of the enclosing process definition.

OWL-S predefines several control constructs: *Sequence*, *Split*, *Split + Join*, *Choice*, *Any-Order*, *Condition*, *If-Then-Else*, *Iterate*, *Repeat-While*, *Repeat-Until*, and *AsPro-*

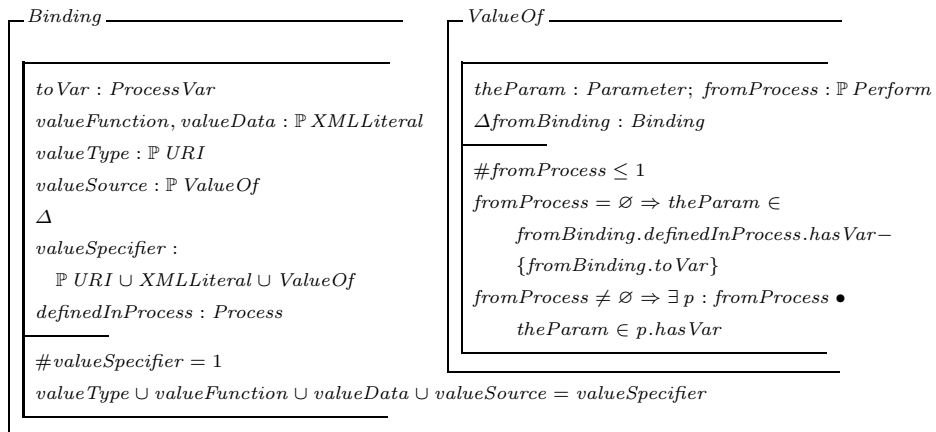
cess. Due to the limited space, we only present the formal model for some of these constructs here.

Sequence, as a subclass of *controlConstruct* itself, denotes a list of control constructs to be done in order. The class invariant means that all *ControlConstruct*s in the list must be in the same composite process. The OWL-S specification does not define how to infer the IOPE of composite process from the *Sequence*'s component process. The tool developers have the freedom to extend the model and add extra constraints, e.g., by adding the predicate '*definedInProcess.hasResult* = $\bigcup\{c : \text{rancomponents} \bullet c.\text{hasResult}\}$ ' to define the result of the sequence to be the union of the result of the individual members or '*definedInProcess.hasResult* = *components*(*#components*).*hasResult*' to use the last event's result as the result of the sequence, etc.



The formal models for other types of *controlConstruct* are omitted in this paper. Note that in this paper we only focus on the syntax and static semantic of OWL-S. The dynamic semantics of OWL-S, which can be formally modeled as a set of Object-Z operations, will be addressed in another paper. OWL-S *process* is defined as a class union as *AtomicProcess* \cup *SimpleProcess* \cup *CompositeProcess* .

Data flow and variable bindings

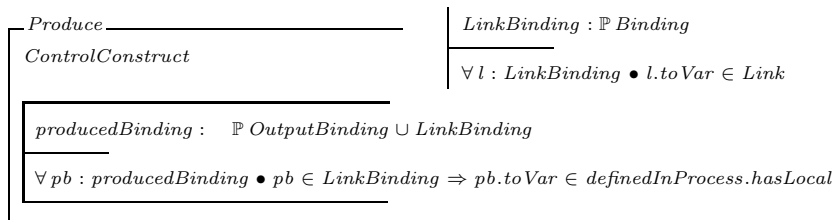


In OWL-S, there are two complementary ways to specify data flow between steps: *consumer-pull* and *producer-push*. The *consumer-pull* approach specifies the source of a datum at the point where it is used. OWL-S implements this convention by providing

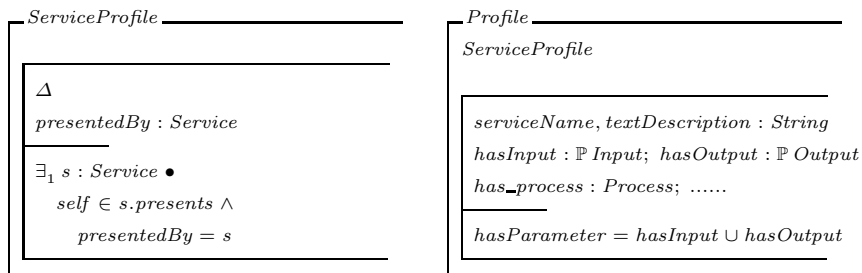
a notation for arbitrary terms as the values of input or output parameters of a process step, plus a notation for subterms denoting the output or input parameters of prior process steps. We will specify this approach first. *Bindings* are used to specify the data flow in an OWL-S model, e.g., how output parameters are computed in different result conditions for atomic processes. The attribute *toVar* of *Binding* denotes the variable to be bound. *ValueFunction*, *valueType*, *valueData* and *valueSource* are different kinds of possible value sources it receives from. They are also called *valueSpecifier*. There can only have one value source for a binding.

For example, *Binding*'s attribute *ValueSource* and its type *ValueOf* are one sort of data flow. It is used to describe the source of a value that is a parameter (*theParam*) of another process within a composite process (*fromProcess*). We model that type of *fromProcess* as a set of composite processes with a predicate of cardinality less than one to show that there is at most one *fromProcess* is defined. If the *fromProcess* corresponds to the empty set, then *theParam* must be a different parameter from the same process.

Producer-Push is another ways in OWL-S to specify how data flows. It is used when the output of a process depends on the branch of conditions the agent takes. So at the point in a branch where the data required to compute the output are known, we insert a *Produce* pseudo-step to say what the output will be. *Producer*, modeled as a subclass of *ControlConstruct*, 'pushes' a value available at run time to an Output. The attribute *producedBinding* specify the target of data flow, which specifies either a *LinkBinding* or an *OutputBinding*. *LinkBinding* is one kind of *Binding* such that the variable to be bound must be a *Link* variable. The invariant of *Produce* ensure that link variable must be defined within the composite process.



3.3 Service Profile and grounding



In OWL-S, the Service Profile provides a way of describing the services offered by providers, and the services needed by requesters. Three basic types of information about a service is provided, as what organization provides the service, what function the service computes, and a host of features that specify the service characteristics. *ServiceProfile* denotes the generic OWL-S profile, while *Profile* defines a subclass of *ServiceProfile* that denotes the predefined profile. It is used to acknowledge the different ways of profiling services from the default one.

The grounding of a service specifies the details of how to access a service - details such as protocols and message formats, serialization, transport, and addressing. OWL-S does not include an abstract construct for explicitly describing messages. Therefore, we just abstract *ServiceGrounding* as a simple OZ class.

4 Discussions

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

4.1 Providing a unified, consistent precise description of OWL-S

OWL-S is currently under further development, and thus may still contain errors. The syntax and static semantics and dynamic semantics of OWL-S are described separately using different formats, such as English, the Ontology itself, and some simple axioms. It has been very difficult to consistently extend and revise these descriptions with the continuous evolution of OWL-S. For example, Figure 1 shows a part of the text and OWL ontological definition of *AtomicProcess*. The text definition requires that for an atomic process, there are always **only** two participants, TheClient and TheServer. On the other hand, according the OWL semantics, the ontological definition requires that for an atomic process, there are **always** two participants, TheClient and TheServer. However, there could exist some other participants. Those inconsistent definitions can lead to many difficulties when OWL-S is used in practice.

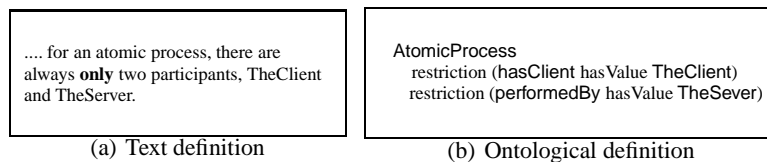


Fig. 1. Inconsistent description among different formats

Furthermore, as a small fragment of FOL, OWL is also not expressive enough to define all the desire properties of OWL-S. Our formal model of OWL-S provides a unified framework, so all different aspects of OWL-S can be formally defined using a single language consistently. Furthermore, as our formal model provides a rigorous foundation of the language, by using existing formal verification tools such as an Object-Z type checker, it may be possible to find possible errors and improve the quality of the OWL-S standard.

Large sections of the OWL-S 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 conclusions being drawn. Our formal OWL-S model can be used to improve the quality of the normative text that defines the OWL-S 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.

4.2 Reasoning the OWL-S by using exiting formal tools directly

Since research into Semantic Web Services in general, and OWL-S in particular is still evolving, current verification and reasoning tools (though rudimentary) for validating OWL-S models are also evolving. 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 OWL-S, many Object-Z and Z tools can be used for checking, validating and verifying the OWL-S model. For example, in our previous work, we have applied Z/EVES [7, 6] and AA [9] separately to reason over Web ontologies. We also applied an Object-Z type checker to validate an OWL-S 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 OWL-S.

5 Conclusion

In this paper, we have presented a formal model of the OWL-S language using the Object-Z formal specification, whereby the OWL-S constructs are modeled as objects. The advantage of this approach is that the abstract syntax, static and dynamic semantics of each OWL-S construct are all grouped together and captured in a single Object-Z class; hence the language model is structural, concise and easily extendible. Subsequent work will address and complete the dynamic semantics of OWL-S. We believe this OZ specification can provide a useful document for developing support tools for OWL-S.

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. Anupriya Ankolekar, Frank Huch, and Katia P. Sycara. Concurrent execution semantics of daml-s with subtypes. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 318–332, London, UK, 2002. 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 and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.
6. J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. A combined approach to checking web ontologies. In *The 13th ACM International World Wide Web Conference (WWW'04)*, pages 714–722. ACM Press, May 2004.
7. J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. Verifying DAML+OIL and Beyond in Z/EVES. In *Proc. The 26th International Conference on Software Engineering (ICSE'04)*, pages 201–210, Edinburgh, Scotland, May 2004.
8. J. S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. Technical Report 95-20, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.
9. J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *12th International Symposium on Formal Methods Europe (FM'03)*. Springer-Verlag, September 2003.
10. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing, Macmillan, March 2000.
11. A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *Object-Oriented Systems*, 2:195–215, Chapman & Hall 1995.
12. 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.
13. Yang Liu and Jun Sun. Algorithmic design using object-z for twig xml queries evaluation. *Electr. Notes Theor. Comput. Sci.*, 151(2):107–124, 2006.
14. S. McIlraith, T. Son, and H. Zeng. Semantic web services, 2001.
15. Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM Press.
16. 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.
17. 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.
18. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
19. 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.
20. Hai H. Wang, Nick Gibbins, Terry Payne, Ahmed Saleh, and Jun Sun. A Formal Semantic Model of the Semantic Web Service Ontology (WSMO). In *The Twelfth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'07)*, Auckland, NZ, July 2007. IEEE Press.
21. 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.