J. Software Engineering & Applications, **2009, 2:25-33**
Published Online April 2009 in SciRes (www.SciRP.org/journal/jsea)

Formal Semantics of OWL-S with Rewrite Logic[#]

Ning Huang[*], Xiao Juan Wang[*], Camilo Rocha[+]

[*]Beihang University, Beijing, China, [+]University of Illinois at Champaign Urbana, USA
Email: hn@buaa.edu.cn; wxjbuaa@hotmail.com

Received December 29th, 2008; revised January 27th, 2009; accepted February 27th, 2009.

ABSTRACT

SOA is built upon and evolving from older concepts of distributed computing and modular programming, OWL-S plays a key role in describing behaviors of web services, which are the essential of the SOA software. Although OWL-S has given semantics to concepts by ontology technology, it gives no semantics to control-flow and data-flow. This paper presents a formal semantics framework for OWL-S sub-set, including its abstraction, syntax, static and dynamic semantics by rewrite logic. Details of a consistent transformation from OWL-S SOS of control-flow to corresponding rules and equations, and dataflow semantics including "Precondition", "Result" and "Binding" etc. are explained. This paper provides a possibility for formal verification and reliability evaluation of software based on SOA.

Keywords: *SOA, Web Services, OWL-S, Formal Semantics, Rewrite Logic, Consistent Transformation, Reliability Evaluation*

1. Introduction

Service Oriented Architecture (SOA) adopts the Web services standards and technologies and is rapidly becoming a standard approach for enterprise information systems. We believe that there will be a heavy demand of reliability evaluation for the SOA software in the early development phase.

Web services are the essential of the SOA software, because SOA offers one such architecture, it unifies business processes by structuring large applications as an ad hoc collection of smaller modules called "services", Services-orientation aims at a loose coupling of services with operating systems, programming languages and other technologies which underlie applications. There are many higher level standards such as BPEL [1], WSCI, BPML, DAML-S (the predecessor of OWL-S), etc. OWL-S (Web Ontology Language for services) is a well-established language for the description of Web services based on ontology, it has been recommended by Web-ontology Working Group at the World Wide Web Consortium [2].

In order to undergo more accurate reliability evaluation and prediction of software based on SOA in the early phase, we should give software architecture the description semantics for gaining more components information, according to this motivation, we plan to use OWL-S to describe the software architecture, and then use formal method to construct a well-defined mathematical model of the system described by OWL-S, once we have this formal model, we will compute the reliability of the software based on the formal model. Here, as a meta-language of rewrite logic, Maude has been chosen as a language to give OWL-S formal semantics in a framework in static and dynamic aspects. With this framework in hand, an OWL-S specification can be transformed into an easy-understand formal one only concerning syntax, and this transformation makes a critical contribution for the formal verification and reliability evaluation of OWL-S model using the Maude language.

The rest of this paper is organized as follows. We will give an overview of related works in Section 2, the background of our method will be presented in Section 3 by an overview of OWL-S, rewrite logic and Maude, section 4 presents the abstraction of the model to introduce what we should abstract from the model. How to give OWL-S model the formal semantics using Maude in static and dynamic aspects will be presented respectively in Section 5 and Section 6. Finally, we will give a conclusion in Section 7 and an acknowledgement in Section 8.

2. Related Works

Most of previous related works focus on model checking instead of providing a precise formal semantics for the specification. For example, [3] converts OWL-S Process Model using a C-Like code specification language and then uses BLAST to validate it by the test cases automatically generated in the model checking process. [4] proposes a Petri Net-based operational semantics, which models the control flow of DAML-S (the former of OWL-S) Process Model, but lack of dataflow. Based on the work of [4,5] extends it to translate OWL-S Process Model using Promela, a SPIN specification language, and uses SPIN to do model checking. [6] presents a formal denotational model of OWL-S using the Object-Z(OZ) specification language, but it focuses on the formal model

[#]Supported by the 11th Defense Five-Year plan Foundation, and Aviation Fund of China

for the syntax and static semantics of OWL-S, and does not discuss the dynamic semantics.

These researches give good examples of formal specification of OWL-S and applications, but there are some problems:

(1) Some semantics is undefined: Because OWL-S is not a traditional language, so some properties can't be expressed directly. Some papers didn't claim this problem explicitly, but some do so, for example, in [5], "Precondition" and "Effect" can't be expressed in Promela and so ignored. In this paper, these properties are declared as important ones for processes and defined clearly.

(2) Transforming consistency: The structural operational semantics (SOS) of OWL-S hasn't been mapped to the specification language directly in related researches. This gives less consistency assurance for the transformation. But for rewrite logic, the mapping is rather clear. On the other hand, the later transformation for OWL-S specification only concerns syntax.

(3) Lack of analysis of dataflow: Among the above researches, only [5] explicitly stated the dataflow, it simplifies "Input" and "Output" as "Integer" and connects them to "channel". On the other hand, rewrite logic used in this paper is much more appropriate and efficient for properties deal not only with causality of events but also with data types or recursive constructs.

[6] and [7] use an algebra specification language Z. But the former focuses on the analysis of ontology, including some properties verification and reasoning without analysis of control flow and dataflow. The latter gives a good explanation for static semantics of OWL-S but without dynamic semantics.

3. Background

3.1 Introduction of OWL-S

OWL-S is an OWL-based (Recommendation produced by the Web-Ontology Working Group at the World Wide Web Consortium) Web service ontology, which supplies Web service providers with a core set of constructs for describing the properties and capabilities of their Web services in unambiguous, computer interpretable form. OWL-S markup of Web services will facilitate the automation of Web service tasks, including automated Web service discovery, execution, composition and interoperation.

It is the first well-researched Web Services Ontology, which has numerous users from industry and academe, and is still undergoing. Details of the latest version of OWL-S submission document can be referred to [7].

3.2 Rewrite Logic and Maude

Rewriting logic is a computational logic that can be efficiently implemented and that has good properties as a general and flexible logical and semantic framework, in which a wide range of logics and models of computation can be faithfully represented [8].

Definition 1: A rewrite theory R is a triple R = (Σ, E, R), with:

- (Σ,E) a membership equational theory, and
- R a set of labeled rewrite rules of the form: "$l : t \to t' \Leftarrow$ cond", with "l" as a label, t, t′ $\in T_{\Sigma}(X)_k$ for some kind k, and "cond" is a condition (involving the same variables X).

In general, a rule in rewrite logic is like:

$$l : t \to t' \Leftarrow \left(\bigwedge_i u_i = u'_i\right) \wedge \left(\bigwedge_j v_j : s_j\right) \wedge \left(\bigwedge_k w_k \to w'_k\right)$$

Maude is a formal programming language based on the mathematical theory of rewriting logic. With Maude system's support, this kind of language specifications can be executed and model can be checked, what is more, its mathematic semantics can be obtained. A program in Maude is just a rewrite logic theory, and Maude offers a comprehensive toolkit for the analysis of specifications, such as LTL model checker, Inductive Theorem Prover (ITP), Maude Termination Tool, Church Rosser Checker, Coherence Checker, etc.

In Maude, object-oriented systems are specified by object-oriented modules, defined by the keyword "*omod ... endom*", in which classes and subclasses are declared. A class declaration has the form *class* C|a_1: S_1, ..., a_n: S_n ,where *C* is the name of the class, the a_i are attribute identifiers, and the S_i are the sorts of the corresponding attributes. An object of a class *C* is defined as:< *O : C | a1 : v1, ... ,* $a_n : v_n$ >, where *O* is the object's name, and the v_i are the corresponding values of object's attributes, for i=1 . . . n. Objects can interact in a number of different ways such as messages passing, messages between objects can be defined by the keyword "msg". Details of Maude can be referred to [9].

The main reasons why we choose rewrite logic to give the OWL-S specification the formal semantics are listed as follows, and more detail advantages can be referred to [10]:

- ♦ Consistency in transforming: Rewrite logic is a flexible and expressive one that unifies algebraic denotational semantics and structural operational semantics (SOS) in a novel way, which can be seamlessly transformed from OWL-S SOS into rewrite rules/equations [8], and suitable for describing data types and their relationships. On the other hand, the latter transforming from an OWL-S specification model into an algebraic one only concerns syntax.
- ♦ Suitable for many logic formula expressions in OWL-S: In [11], it is argued that rewriting logic is suitable both as a logical framework in which many other logics can be represented, and as a semantic framework.
- ♦ Efficiency implementation: Maude is a high performance rewriting logic implementations [12]. It is demonstrated that the performance of the Maude model checker "is comparable to that of current explicit-state model checkers" such as SPIN [11].

♦ Analyzing tools: It has been well established now that a rewrite logic specification can be benefited for comprehensive tool-supported formal verification [13].

4. Abstraction of the Model

There are three methods to transform an OWL-S specification into a rewrite logic one:

(1) Translate every lines in OWL-S specification into corresponding rewrite logic ones directly. The translating is the same as giving semantics to OWL-S specification.

(2) Give OWL-S (the language) rewrite logic semantics for every parts of its syntax. Then the OWL-S specification itself is a rewrite logic one with semantics. None transformation is needed.

(3) Abstract the main parts of OWL-S (the language), define syntax in rewrite logic for the sub-set and give semantics for the syntax. And then translate the OWL-S specification into a rewrite logic one by abstracting it into the syntax in rewrite logic.

Method (1) is direct, but difficult to ensure the consistency. Method (2) is a complete one. For example, a service is modeled by a process in OWL-S, some tags such as < *process: CompositeProcess rdf:ID="CP"*>, <*process: hasInput rdf:resource="#inputownright1"/*> are used to give a detailed perspective within a composite web service. All tags should have semantics (here the tags are "*process: CompositeProcess rdf: ID*" and "*process: hasInput rdf: resource*").

In this paper, method (3) is accepted. One reason to do so is that abstraction can reduce the complexity of analyzing a model; another reason is that we can go to the most difficult and challenge problems quickly.

In the following section, we will explain what has been abstracted from OWL-S, including control flow and data flow. This abstraction becomes a sub-set of OWL-S.

1) Parameters and Expressions: Parameters are the basis of representing expressions, conditions, formulas and the state of an execution. In OWL-S, parameters are distinguished as *"ProcessVar", "Variables" and "ResultVar"*, etc. They can even be identified as variables in SWRL. Our abstraction in this paper doesn't distinguish these, but refer them all as parameters.

Expressions can be treated as literals in OWL-S, either string literals or XML literals. The later case is used for languages whose standard encoding is in XML, such as SWRL or RDF. In this paper, expressions are separated into Arithmetic and Boolean expressions.

2) Precondition: If a process's precondition is false, the consequences of performing or initiating the process are undefined. Otherwise, the result described in OWL-S for the process will affect its "world".

3) Input: Inputs specify the information that the process requires for its execution. It is not contradictive with the definition of messages between web services, because a message can bundle as many inputs as required, and the bundling is specified by the grounding of the process model.

4) Result and Output: The performance of a process may result in changes of the state of the world (effects), and the acquisition of information by the client agent performing it (returned to it as outputs). In OWL-S, the term "Result" is used to refer to a coupled output and effect. Having declared a result, a process model can then describe it in terms of four properties, in which, the "*inCondition*" property specifies the condition under which this result occurs, the "*withOutput*" and "*hasEffect*" properties then state what ensures when the condition is true. The "*hasResultVar*" property declares variables that are bound in the "*inCondition*".

Precondition and Result are represented as logical formulas in OWL-S, but when they are abstracted, Boolean expression and assignment are used separately in this paper.

5) Process: A Web service is regarded as a process. There are three different processes: Atomic process corresponds to the actions that a service can perform by engaging it in a single interaction; composite process corresponds to actions that require multi-step protocols and/or multiple services actions; finally, simple process provides an abstraction mechanism to provide multiple views of the same process. We focus on atomic process and composite process here.

6) Control structure: Composite processes are decomposable into other (non-composite or composite) processes; their decomposition can be specified by using eight control structures provided for web services, including *Sequence, Split, Split-Join, Choice, Any-Order, If-Then-Else, Repeat-Until, and Repeat-While.*

7) Dataflow and Variables binding: When defining processes using OWL-S, there are many conditions where the input to one process component is obtained as one of the outputs of a preceding step. This is one kind of data flow from one step of a process to another.

A Binding represents a flow of data to a variable, and it has two properties: "*toVar*", the name of the variable, and "*valueSpecifier*", a description of the value to receive. There are four different kinds of valueSpecifier for Bindings: *valueSource, valueType, valueData,* and *valueFunction*. The widely used one "*valueSource*" is addressed in this paper.

The information listed above gives an overview of how web services are bound together with control structures and dataflows.

5. Syntax and Static Semantics in Maude

According to the method (3) described above, we now need to define how to express the information abstracted in Section 3 in rewrite logic, namely, syntax of the sub-set in Maude. Because of space limited, we only explain parts of it:

1) Parameters and Expressions: To express them, several rewrite logic modules have been defined. They are *NAME*, *EXP* and *BEXP*.

To specify process variables we define a module named "*NAME*", in which "*op_._: Oid Varname-> Name* "is defined to be the form "process.var" as a variable name, while *"Oid"* is the name of a process, which has been regarded as an object identification. And a *"NameList"* is used to be a list of variables.

The value of a variable is stored in a "Location" which is indicated by an integer. When we bind a location with a variable name, the variable get the value stored in that location.

Arithmetic expressions (sort name is *"Exp"*) and Boolean expressions (sort name is *"BExp"*) are defined separately in module *EXP* and *BEXP*, which gives a description of how to use variable names to describe expressions.

2) IOPR (Input/Output/Precondition/Result), Data flow and variable bindings:

"Input" and *"Output"* of a process are defined as *"NameLists"* which are attributions of a process.

In OWL-S, *"Precondition"* and *"Effects"* are represented as logical formulas by other languages such as SWRL. Here we first simplify Precondition as "BExp" to be an attribution of a process class.

"Result" is more complicated. After separate *"Output"* as an attribution of a process, *"Result"* combines a list of *"Effect"*, while every Effect is simplified as a conditional assignment here. The definition in Maude is " *op_<-_if_: Name Exp BExp -> Effect.*"

As discussed above, there are four types of binding *"valueSpecifier"*. Here we defined binding as "*op fromto* : *Name Name -> Binding* " to specify *"valueSource"* in module *WSTYPE*. With this definition, dataflow in a composite web service is created.

3) Processes and Control Structures: Atomic and composite web services are defined as two classes with different attributions. In order to distinguish definitions of *"ControlConstructList"* and *"ControlConstructBag"* for control structure, *"OList"* is defined to represent the object list which should be executed in order, and *"OBag"* to represent there is no order for the objects.

It seems very hard to express that a web service set can be executed in any order. But benefited with Maude operator attribution "comm", we can get this with definition "*op_#_: OBag OBag -> OBag [ctor assoc comm id: noo]*". *"comm"* attribution means that this "op" is with commutative property, which makes the objects in this "bag" ignore the order unlike it is in *"OList"*.

After defining two sorts as follows:

subsort Qid < Oid < Block < BlockList.
subsort Qid < Oid < Block < BlockBag.

We define a nested control structure. For example, *"sequence"* as "*op sequence: BlockList->Block [ctor]*" and "*split" as "op split : BlockBag -> Block [ctor]* ". This separates *"Block"* into three cases:

(1) Only a process.

(2) A group of processes within one control structure (we refer it as a control block).

(3) A group of processes and control blocks within one control structure.

Obviously, the (3) is a nested control structure. If the group is order sensitive, it is a *"BlockList"*, otherwise, it is a *"BlockBag"*.

Syntax of atomic web service: A class *"Atomws"* is defined in Definition 2. When an instance of atomic web service is created, it should be declared as an object of class *"Atomws"*.

Definition 2: *class Atomws | initialized : Bool,*
father : Oid, input : NameList,
output : NameList, precondition : BExp,
result : Effect .

Syntax of composite web service: And a class *"Compositews"* is defined in Definition 3. We have explained "IOPR", "Result", "Precondition" and "Binding" above. Other attributions are: "initialized" to represent whether this instance object (composite web service) of the class has been initialized with actual values of its "IOPR", "Result", "Precondition", "Binding" and control structures. *"father"* denotes which composite web service (instance) it belongs to. *"struc"* is the control structure with *"BlockList"* and *"BlockBag"* as its subsort. Other attributions are defined to be used when the composite one is executed, especially for the nested control structures.

Definition 3: *class Compositews | initialized : Bool,*
input : NameList, output : NameList,
precondition : BExp, result : Effect,
binding : Binding, father : Oid,
struc : CStruc, nest : BlockList, wait : OList,
blockwait : NList, waitbag : OBag .

When an instance of composite web service is created, it should be declared as an object of class *"Compositews"*. And prepare an initial equation for itself (how to define an initial equation is ignored here).

6. Dynamic Semantics in Maude

6.1 Auxiliary Modules

When "Precondition" of a process is true, it can be initialized and executed. It affects the "world" by various "Effect". So we need to define what the "world" will be for a web service. Here a module of *"SUPERSTATE"* is extended with *"CONFIGURATION"* which already defines as a "soup" of floating objects and messages in Maude.

A *"Superstate"* is the "world" of a process which defined as "*op_|_: State Configuration -> Superstate*". "*State*" is a group of variables with corresponding locations, and locations with corresponding values. A message is defined as "*msg call: Oid Oid -> Msg*" for a composite web service to trigger its sub-process to execute. Another is defined as "*msg tellfinish: Oid Oid -> Msg*" to tell its father that it has finished execution.

In module *"SUPERSTATE"*, assignment, evaluation of an arithmetic expression and a Boolean expression are defined, which gives semantics to how these syntax can be executed to affect the "world" of a process.

An operator *"k"* is defined as "*op k: Configuration -> Configuration*" to indicate that one web service is ready to be executed. Two operators *"val"* and *"bval"* are defined to evaluate expression and Boolean expression values in a state.

A sort *"NList"* (natural number list) is also defined in *"NLIST"* module to give semantics of executions of a nested control structure, with the help of the four attributions: *nest, wait, blockwait, and waitbag*.

6.2 Dynamic Semantics

In this section, we first analyze how executions of web services can affect their "world", and then by giving out the SOS for control structure, explain the corresponding rewrite logic rules or equations.

1) Execution of a Service:

◎Execution of an Atomic One

As defined in OWL-S, atomic processes have no sub-processes and execute in a single step only if the service is concerned and its precondition is true. The execution gives result to its "world" by "Effect". The main parts for its execution semantics (Figure 1) have been chosen to be explained as below.

Equation (1) asks atomic web service "ws" do initialization if its precondition *"Cd"* is true and hasn't been initialized before. Initialization is designed as an equation in a module of an instance of "Atomws". It prepares an initial state for this web service.

Equation (2) explains that when an atomic web service *"ws"* gets a message from its father *"F"*, it is the same meaning that it will be executed after initialized.

```
(1) ceq   < ws : Atomws | initialized : init-state, father : F,
      precondition : Cd >   call(F, ws)
    = ( initial( < ws : Atomws | initialized : true, father : F
    > )   call(F, ws) ) if ( not init-state) and bval(Cd, st1) .
(2) eq st1 | (conf < ws : Atomws | initialized : true , father :
    F> call(F, ws)) = st1 |(conf k( < ws : Atomws | father :
    F > )) .
(3) eq   st1 | (conf k(< ws : Atomws | father : F, result : (x1
                          <- exp1 if cond) Eff >))
    = st1 | (conf k(< ws : Atomws | father : F, result : (x1 <-
    exp1 if bval(cond, st1)) Eff >)) .
(4) rl [ execute ] :
    (mem(x1, location1) loc(location1, y1) st1) | (conf k(<
    ws : Atomws | father : F , result : (x1 <- y) Eff >))
    => (mem(x1, location1) loc(location1, y) st1) | (conf k(<
    ws : Atomws | father : F , result : Eff >)) .
(5) rl [finish] :   st1 |(conf k(< ws : Atomws | father : F,
    initialized : init-state , result : noEff >))
    => st1 | (conf < ws : Atomws | father: F , initialized:
    false. result: noEff > tellfinish(F. ws)).
```

Figure 1. Semantics of execution atomic web service

Equation (3) explains that how to execute a condition inside an *"Effect"*. Of course there are rules that explain how to evaluate expression inside an *"Effect"* (ignored here).

The forth rule (4) simulates state changes by one *"Effect"*. And rule (5) ensure that only after all the *"Effect"* of this web service has been executed it tells its father it has finished, and prepares a same instance waiting for its *"Precondition"* to be true to be initialized to execute again.

◎Execution of a composite process

Composite web service changes its "world" by executing its sub-processes according to its control structures.

Different from atomic web service, before a composite one is going to be executed, it should prepare "binding" information. A rule below is used to explain how to do that. After that, "sourcedata" should be defined to affect the "world" by other rules.

```
rl   st1 | (conf call(F, ws) < ws : Compositews | initialized :
true, father : F, binding : fromto(v1 , v2) BD > )
=> (sourcedata(v1, v2, st1) st1) | (conf < ws : Compositews |
father : F, binding : BD >   call(F, ws) ) .
```

After that, composite web service will be executed when its precondition is true like the atomic one. The difference is that the sub-processes grouped in control structures should be executed according to semantics of control. How to execute these structures will be explained below.

2) Sequence: The SOS of "sequence" and the corresponding rewrite logic rule are showed in Figure 2. "BLK" is a "Block" and "BL" is a *"BlockList"*. Obviously, attribution *"nest"* here is used to separate the *"BlockList"*, leaves the first one in *"struc"* to be executed first.

The question is how to ensure the first control block be executed firstly? Especially when it is a nested control structure-because when the most inner to be executed, the decomposing difference (order sensitive of *BlockList* and opposite *BlockBag*) should be recorded.

As discussed above in Section 4, a *"Block"* has three cases. But all of them should ensure that this *"Block"* be executed before *"BL"* is going to be executed for *"sequence"*. To ensure this, two attributions *"wait"* and *"blockwait"* are defined. *"wait"* is a *"OList (object list)"* to ensure that only after the list of objects are all finished that this service "ws" can be executed. *"blockwait"* is defined as *"NList (nature number list)"*. When an order

$$\frac{<BLK, \sigma> \;\rightarrow \sigma'' <BL, \sigma''> \;\rightarrow \sigma'}{<sequence\ (BLK;\ BL)\ \sigma> \;\rightarrow \sigma'}$$

```
rl k( < ws : Compositews | father : F, struc : sequence(BLK ;
BL), nest : BL1, wait : nilo, blockwait : BW > )
=><ws:Compositews|father:F, struc:BLK, nest : sequence(BL) ;
BL1, wait: nilo, blockwait : (1 BW) >   call(F, ws) .
```

Figure 2. Semantics of execution sequence

sensitive control block (here is sequence) is separated, it definitely asks the *"Block"* left in *"struc"* (here is *BLK*) should be finished before other "Block" left in "nest" are going to be executed. So we add natural number "1" into the *"NList"* (here is *BW*). Otherwise, "0" is added for no order sensitive one, such as for control structure "Split". And "2" will be added when the outer control structure asks order but this one doesn't.

After separating a control structure, there are three cases waited to be explained of how to execute *BLK*.

Case 1: *BLK* is a process.

In this case, if it is a composite one, it can be separated recursively. If it is an atomic one (here is *"A"*), different rules should be matched according to "blockwait" (here is *BW*). The value of head of *"BW"* is "1" means *"A"* should be completely finished before *"ws"* continues, showed as Figure 3 rule (1). So *"A"* is put into "wait", and "1" is added to *"BW"* of "ws" to indicate that this an order sensitive block. And then two messages are released to trigger *"A"* and *"ws"*. Of course, there should be a corresponding rule when *"A"* completes its execution (Figure 3 rule (2)).

If *head(BW)==0 and sum(BW) > 0* (rule (3)), then "2" is added to *"BW"* and *"A"* is added to "waitbag" (here is OO), this indicates that *"A"* need not to be executed firstly in this level of the control structure, but it need to be so in outer control structure. The corresponding rule to release the blocking is showed as rule (4).

Similarly if *head (BW) == 0 and sum(BW) == 0*, "0" is added to *"BW"* to indicate there is no need for *"A"* to be executed firstly.

```
(1)  crl : k( < ws : Compositews | father : F, struc : A ,
       wait :nilo, blockwait : BW > )
          => < ws : Compositews | father : F, struc : empty,
      wait : A ,
       blockwait: ( 1 BW) >   call(ws, A) call(F, ws)
       if head(BW) == 1 .
(2)  crl k( < ws : Compositews | father : F, struc : CS1,
      nest :BL, wait : A ~ L, blockwait : BW >)    tellfinish(ws, A)
     => (call(F, ws)    < ws : Compositews | father : F,
   struc : CS1,
      nest : BL , wait : L, blockwait : BW > )    if head(BW)
   == 1 .
(3)  crl :k( < ws : Compositews | father : F, struc : A ,
      wait : nilo, blockwait : BW, waitbag : OO > )
       => call(ws, A) call(F, ws) < ws : Compositews | father :
      F, struc : empty, wait : nilo, blockwait : ( 2 BW), waitbag : A # OO >
     if head(BW) == 0 and sum(BW) > 0 .
(4)  crl k( < ws : Compositews | father : F, struc : CS1,
       nest :BL, waitbag : A # OO, blockwait : BW >)    tellfinish(ws, A)
       => < ws : Compositews | father : F, struc : CS1, nest :
      BL , waitbag : OO, blockwait : BW > call(F, ws)
          if head(BW) == 2 or sum(BW) > 0 .
```

Figure 3. Semantics of execution nested control Block

$$\frac{\langle bexp1,\ \sigma\rangle \rightarrow false}{\langle repeat\text{-}while\ bexp1\ CS1,\ \sigma\rangle \rightarrow \sigma}$$

$$\frac{\begin{array}{l}\langle bexp1,\ \sigma\rangle \rightarrow true\\ \langle CS1,\ \sigma\rangle \rightarrow \sigma''\ \langle repeat\text{-}while\ bexp1\ CS1,\ \sigma''\rangle \rightarrow \sigma'\end{array}}{\langle repeat\text{-}while\ bexp1\ CS1,\ \sigma\rangle \rightarrow \sigma'}$$

```
crl  k( < ws : Compositews | father : F, struc : repeat CS1
while bexp1, nest : BL, wait : nilo > )     =>
if bexp1
          then < ws : Compositews | father : F, struc : CS1, nest :
(repeat CS1 while bexp1) ; BL, wait : nilo >      call(F, ws)
          else < ws : Compositews | father : F, struc : empty,
nest : BL, wait : nilo > call(F, ws)
       fi.
```

Figure 4. Semantics of execution Repeat-while

Case 2 and Case 3: are similar with case1, but with recursive definition. Because of space limited, we will not discuss them here

3) Repeat-while: "Repeat-While" tests the condition, exits if it is false and does the operation if the condition is true, then loops. Its SOS and corresponding rewrite rule are showed in Figure 4.

Actually, for the control structure itself, it is not complex. The rule in Figure 4 just gives semantics of how this structure can be executed. The difficult is how "Block" can be executed inside it.

For example, when there is simple composite web service which contents only one atomic web service "A" within its repeat-while, say *(k(< ws : Compositews | father: F, struc: repeat 'A while bexp1, nest: BL, wait: nilo >))*, how about the execution of *"A"*?

As discussed for Definition 2 and 3, before this composite *"ws"* enters its execution *"k"* state, it should prepare an atomic instance *'A*. If the precondition of *'A* is true, it can be initialized and go into *"k"* execution state. This may affect the "world" of *"ws"* according to the group rules and equations in Figure 1. After 'A has finished its execution, rule (5) in Figure 1 prepares another instance *'A* waiting for its *"precondition"* again. If *"bexp1"* decides to execute *'A* again, execution can be continue, and the *"Result"* may turn *"bexp1"* to be true by affecting the "world" of *"ws"*.

4) Repeat-until: "Repeat-until" does the operation, tests for the condition, exits if the condition is true, and otherwise loops. Its SOS and corresponding rewrite rule are showed in Figure 5.

$$\frac{\langle CS1,\ \sigma\rangle \rightarrow \sigma' \quad \langle bexp1,\ \sigma'\rangle \rightarrow true}{\langle repeat\text{-}until\ bexp1\ CS1,\ \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\begin{array}{l}\langle bexp1,\ \sigma''\rangle \rightarrow false\\ \langle CS1,\ \sigma\rangle \rightarrow \sigma''\ \langle repeat\text{-}until\ bexp1\ CS1,\ \sigma''\rangle \rightarrow \sigma'\end{array}}{\langle repeat\text{-}until\ bexp1\ CS1,\ \sigma\rangle \rightarrow \sigma'}$$

```
eq   k( < ws : Compositews | father : F, struc : repeat CS1 until
bexp1, nest : BL, wait : nilo > )   =
k(< ws : Compositews | father : F, struc : CS1, nest : (repeat
CS1 while not bexp1) ; BL, wait : nilo >          call(F, ws)).
```

Figure 5. Semantics of execution Repeat-until

Actually, Repeat-While may never act, whereas Repeat-Until always acts at least once. Other executions are the same.

5) Split: The components of a “Split” process are a bag of process components to be executed concurrently. Split completes as soon as all of its component processes have been scheduled for execution.

The rule below creates *“Block”* without order (because of split definition). At last these *“Block”*s produce atomic web services and messages into the “world” of “ws”. Benefitted with objects concurrent execution in Maude, all web services that meet its precondition can be executed concurrently.

rl k(< ws : Compositews | father : F, struc : split(BLK @ BB), nest : BL, wait : nilo, blockwait : BW >)
=> < ws : Compositews | father : F, struc : BLK, nest : split(BB) ; BL , wait : nilo, blockwait : (0 BW) > call(F, ws).

6) Split-join: Here the process consists of concurrent execution of a bunch of process components with barrier synchronization. That is, “Split-Join” completes when all of its components processes have completed. To do this, a special object named “split-join” is defined, and then control structure “split-join(BB)” is equal to “sequence (split(BB) ; 'split-join)”.

7) Choice: “Choice” calls for the execution of a single control construct from a given bag of control constructs. Any of the given control constructs may be chosen for execution. As discussed above, any *“Block”* inside the control bag may be chosen to match *BLK@BB*, because of commutative property. This gives a choice to the control bag. And then “0” is added to *“BW”* means there is no need waiting for this *“BLK”*.

rl k(< ws : Compositews | father : F, struc : choice(BLK @ BB), nest : BL, wait : nilo, blockwait : BW >)
=> < ws : Compositews | father : F, struc : BLK, nest : BL, wait: nilo, blockwait : (0 BW) > call(F, ws).

8) Anyorder: “Anyorder” allows the process components (specified as a bag) to be executed in some unspecified order but not concurrently. Execution and completion of all components is required. The execution of processes in an Any-Order construct cannot overlap, i.e. atomic processes cannot be executed concurrently and composite processes cannot be interleaved. All components must be executed. As with Split+Join, completion of all components is required.

rl k(< ws : Compositews | father : F, struc : anyorder(BLK @ BB), nest : BL, wait : nilo, waitbag: OO, blockwait : BW >)
=><ws:Compositews | father:F, struc: BLK, nest : anyorder(BB) ; BL, wait: nilo, waitbag: OO, blockwait : (1 BW) > call(F, ws).

9) If-then-else: The “If-Then-Else” class is a control construct that has a property *ifCondition*, then and else

$$\frac{<bexp1, \sigma> \rightarrow true \quad <CS1, \sigma> \rightarrow \sigma'}{< if\ bexp1\ then\ CS1\ else\ CS2, \sigma> \rightarrow \sigma'}$$

```
rl k(< ws : Compositews | father : F, struc : if bexp1 then CS1
else CS2, nest : BL, wait : nilo > )     =>
if bexp1
        then < ws : Compositews | father : F, struc : CS1, nest :
BL, wait : nilo > call(F, ws)
        else < ws : Compositews | father : F, struc : CS2, nest :
BL, wait : nilo > call(F, ws)
    fi.
```

Figure 6. Execution of if-then-else

holding different aspects of the If-Then-Else. Its semantics is intended to be “Test If-condition; if True do Then, if False do Else”. Its SOS and corresponding rewrite rule are showed in Figure 6.

As discussed above, the rewrite logic rules are obviously consistent with the definition of SOS benefited from the great expressing capability of rewrite logic.

7. Case Study

Through the modules discussed above, we get a “semantics-OWL-S.maude” rewrite logic theory for semantics of the sub-set OWL-S. With this theory in hand, a software requirement or design in OWL-S can be abstracted into a rewrite logic theory with the syntax described above by extending this frame. Different from other translating methods directly mapping an OWL-S model into another specification language, this way avoids explaining the semantics in an actual model, translating work only concerns syntax mapping while semantics have been given in “semantics-OWL-S. maude”.

For verifying the rewrite logic theory, we give an example to translate the process model to a Maude program, and undergo simple verifications [14] on it.

This example presented by OWL-S in Figure 7 is a web service based on Amazon E-Commerce Web Services. The process is to search books on Amazon by inputted keyword and create a cart with selected items, it is composed of four atomic processes through a sequence control construct.

Through the rewrite theory discussed above, we get a complete Maude program. Here we only display the main parts of the Maude program. Figure 8 is the initializing equation for the third atomic process “cartCreateRequest Process”. In this module, we should build attributes to express process’s IOPRs first. Two inputs and one output are translated name by name. Precondition and Effect in Result are translated to SWRL expression.

The main process of this example is a composite process, and the translated Maude code of initializing equation of it has been shown in Figure 9.

Load the Maude program, and then execute the process by the command “*rew execute-aws*”, we can also search executing path using command “*search execute-aws =>! S: State|C: Configuration tell finish (‘, ‘mainProcess*).” If input a right number less than or equal to the length of

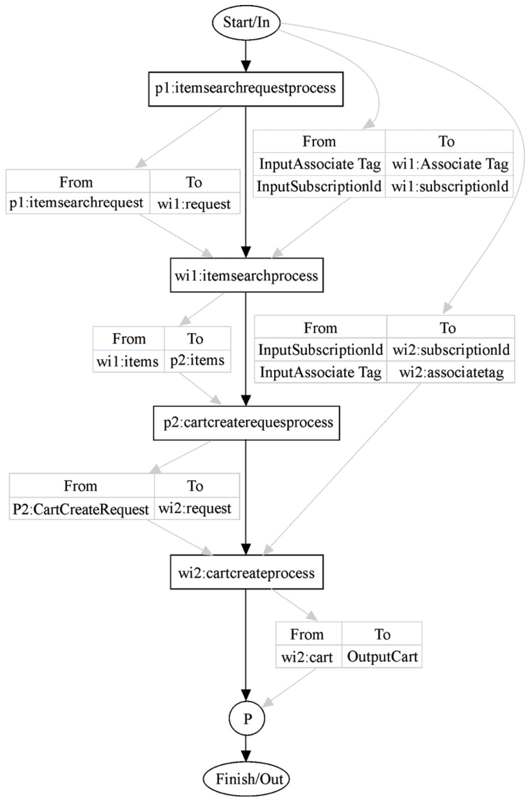

Start/In
p1:itemsearchrequestprocess
From | To
InputAssociate Tag | wi1:Associate Tag
InputSubscriptionId | wi1:subscriptionId
From | To
p1:itemsearchrequest | wi1:request
wi1:itemsearchprocess
From | To
InputSubscriptionId | wi2:subscriptionId
InputAssociate Tag | wi2:associatetag
From | To
wi1:items | p2:items
p2:cartcreaterequesprocess
From | To
P2:CartCreateRequest | wi2:request
wi2:cartcreateprocess
From | To
wi2:cart | OutputCart
P
Finish/Out

Figure 7. Structure of the web service process

```
vars ws F : Oid .
var state : State .
var attrs : AttributeSet .
var conf : Configuration .
var cartCreateRequestProcess : cartCreateRequestProcess' .

eq state | conf initial(<ws : cartCreateRequestProcess |
father : F,attrs>)call(F,ws)
=state | conf<ws : cartCreateRequestProcess |
initialized : true,father : F,input : ws.'index || ws.'items,
output : ws.'cartCreateRequest,
precondition : #not(swrlb:length(ws.'items) #< (ws.'index)),
result:noEff>call(F,ws) .
```

Figure 8. Initial equation of atomic process

'items' for 'index', the input of third atomic process "cart CreateRequestProcess", Maude will display the result in Figure 10. In other cases the result is like Figure 11.

We have done more works concerning this framework, because of space limitation, details are ignored:

(1) Test framework: although the directly mapping from OWL-S SOS to rewrite logic gives the consistency, some web services have been constructed to test the eight control structures and nested ones, including the execution of atomic web services. The results are the same as expected.

(2) Model checking and analysis: several cases are constructed including "philosopher dining" which not only concern control flow but also get a deadlock because of data sharing in the dataflow; and "online shopping" which concerns an error in dataflow. These errors can be found by the Maude analysis tools.

```
eq initial( < WS : MainProcess | father : F, attrs> )
    = < WS : MainProcess | initialized : true, father : F, input : ws .
    'InputAssociateTag || (ws . 'InputAssociateTag || (ws . 'InputSub-
    scriptionId,output : ws . 'OutputCart, precondition : true, result :
      noEff, binding : fromto( 'Perform_Search . 'SubscriptionId, ws .
                                                   'InputSubscriptionId)
 fromto('Perform_Search . 'AssociateTag, ws .'InputAssociateTag)
 fromto('Perform_Search . 'Request, ws . 'ItemSearchRequest)
 fromto('Perform_Search . 'Request, 'Perform_SearchReq . 'Item-
   SearchRequest)
 fromto('Perform_CreateReq . 'items, 'Perform_Search . 'Items)
 fromto('Perform_Create . 'SubscriptionId, ws . 'InputSubscrip-
   tionId)
 fromto('Perform_Create . 'AssociateTag, ws . 'InputAssociateTag)
 fromto('Perform_Create . 'Request, 'Perform_CreateReq . 'Cart-
   CreateRequest) ,
struc : sequence('Perform_SearchReq ; 'Perform_Search; 'Per-
  form_CreateReq; 'Perform_Create),
nest : null, wait : nilo, blockwait : niln, waitbag : noo >
 < 'Perform_SearchReq : itemSearchRequestProcess | initialized :
  false, father : ws,   precondition : true >
 < 'Perform_Search : ItemSearchProcess | initialized : false, fa-
  ther : ws, precondition : ture>
 < 'Perform_CreateReq : ItemSearchProcess | initialized : false,
  father : ws,   precondition : swrlb:length( 'Perform_CreateReq .
  'items) #>= ('Perform_CreateReq .'index) >
 < 'Perform_Create : CartCreateProcess | initialized : false, fa-
  ther : ws, precondition :   true>.
```

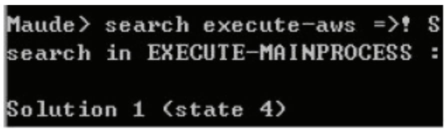
Figure 9. Initializing equation of composite process

```
omod EXEC-MAINPROCESS is
 pr MAINPROCESS .

 op MainProcess : -> MainProcess' .
 op execute-aws : -> Superstate .

 eq exec-MainProcess = loc(1, nilv) loc(2, 'SubscriptionId')
loc(3, 'AssociateTag') loc(4, nilv) loc(5, Keywords-1) loc(6,
SearchIndex-1) loc(7, nilv) loc(8, nilv) loc(9, nilv) loc(10,
index-1) loc(11, nilv) loc(12, nilv) loc(13, index-1)  loc(14,
nilv) loc(15, nilv) loc(16, index-1) loc(17, nilv) loc(18,nilv)
mem('mainProcess    .    'InputSubscriptionId,    2)
mem('MainProcess . 'InputAssociateTag, 3)
mem('Perform_SearchReq    .    'ItemSearchRequest,    4)
mem('Perform_Req      .      'Keywords,      5)
mem('Perform_SearchReq     .     'SearchIndex,     6)
mem('Perform_Search . 'Items, 7)
mem('Perform_Search       .'OperationRequest,       8)
mem('Perform_Search . 'Shared, 13)
mem('Perform_Search . 'Validate, 14) mem('Perform_Search .
  'XMLEscaping, 15) mem('Perform_CreateReq . 'CartCre-
    ateRequest, 9) mem('Perform_CreateReq . 'index, 10)
  mem('Perform_Create . 'Cart, 11) mem('Perform_Create .
 'OperationRequest, 12) mem('Perform_Create . 'Shared, 16)
 mem('Perform_Create . 'Validate, 17) mem('Perform_Create .
  'XMLEscaping, 18) | < 'MainProcess : MainProcess | initial-
          ized : false, father : ', precondition : true >
  call(' , 'MainProcess).
endom
```

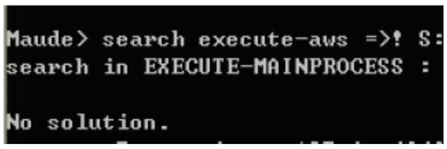
Figure 10. Executing environment module



```
Maude> search execute-aws =>! S
search in EXECUTE-MAINPROCESS :

Solution 1 (state 4)
```

Figure 11. Process can finish



```
Maude> search execute-aws =>! S:
search in EXECUTE-MAINPROCESS :

No solution.
```

Figure 12. Process can not finish

8. Conclusions

This paper gives a formal semantics for OWL-S sub-set by rewrite logic, including abstraction, syntax, static and dynamic semantics. Compared with related researches, the contribution of this paper gives a translation consistency and benefited with formal specification, dataflow can be analyzed deeply, which makes formal verification, and reliability evaluation of software based on SOA possible.

The undergoing future works include: “Precondition” and “Effect” in SWRL format; WSDL and grounding information; and a more complex application analysis.

9. Acknowledgement

This work has been greatly helped by Prof. Meseguer. Thanks to Michael Katelman, Feng Chen and Joe Hendrix in Formal Systems Lab of UIUC, and all the developers of the shared software.

REFERENCES

[1] X. Fu, T. Bultan, and J. W. Su, “Analysis of interacting bpel web services,” in Proceedings of the 13th International Conference on World Wide Web, New York, NY,USA, pp. 621–630, May 2004.

[2] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. R. Payne, E. Sirin, N. Srinivasan, and K. Sycara, “OWL-S: Semantic markup for web services,” Technical Report UNSPECIFIED, Member Submission, W3C, http://www.w3.org/Submission/ OWL-S/, 2004.

[3] H. Huang, W. T. Tsai, R. Paul, and Y. N. Chen, “Automated model checking and testing for composite web services,” in Proceedings Eighth IEEE International Sympo sium on Object-Oriented Real-Time Distributed Com puting, Washington, DC, USA, pp. 300–307, May 2005.

[4] S. Narayanan and S. A. Mcllraith, “Simulation, verification and automated composition of web services,” in Proceedings 11th International Conference on World Wide Web, Honolulu, Hawaii, USA, pp. 77–88, May 2002.

[5] A. Ankolekar, M. Paolucci, and K. Sycara, “Spinning the OWL-S process model-toward the verification of the OWL-S process models,” in Proceedings International Semantic Web Conference 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Hiroshima, Japan, 2004.

[6] H. H. Wang, A. Saleh, T. Payne, and N. Gibbins, “Formal specification of OWL-S with Object-Z: The static aspect,” in Proceedings IEEE/WIC/ACM International Conference on Web Intelligence, Washington, DC, USA, pp. 431–434, November 2007.

[7] J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang, “Verifying DAML+OIL and beyond in Z/EVES,” in Proceedings the 26th International Conference on Software Engineering, Washington, DC, USA, pp. 201–210, May 2004.

[8] T. F. Serbanuta, F. Rosu, and J. Meseguer, “A rewriting logic approach to operational semantics (Extended Abstract),” Electronic Notes Theoretical Computer Science, Vol. 192, No. 1, pp. 125–141, October 2007.

[9] H. Huang and R. A. Mason, “Model checking technologies for web services,” in Proceedings the 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance, Wanshington, DC, USA, pp. 217–224, April 2006.

[10] A. Verdejo and N. Marti-Oliet, “Executable structural operational semantics in maude,” Journal of Logic and Algebraic Programming, Vol. 67, No. 1–No. 2, pp. 226–293, April–May 2006.

[11] M. Birna van Riemsdijk, Frank S. de Boer, M. Dastani, and John-Jules Meyer, “Prototyping 3APL in the maude term rewriting language,” in Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, Hakodate, Hokkaido, Japan, pp. 1279–1281, May 2006.

[12] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott, “All about maude-A high-performance logical framework,” Springer-Verlag New York, Inc., 2007.

[13] J. Meseguer and G. Rou, “The rewriting logic semantics project,” Theoretical Computer Science, Vol. 373, No. 3, pp. 213–237, April 2007.

[14] M. Clavel, F. Duran, etc., “Maude mannual,” Department of Computer Science University of Illinois at Urbana-Champaign, 2007, http://maude.cs.uiuc.edu.