A comparative study between WSCI, WS-CDL, and OWL-S [*]

María Emilia Cambronero, Gregorio Díaz, Enrique Martínez, and Valentín Valero
Escuela Politécnica Superior de Albacete. Universidad de Castilla-La Mancha
Departamento de Sistemas Informáticos
Campus Universitario s/n. 02071. Albacete, SPAIN
{MEmilia.Cambronero,Gregorio.Diaz,Valentin.Valero}@uclm.es
{emartinez}@dsi.uclm.es

Abstract

Choreography languages allow us to describe Web Services compositions from a global viewpoint in Service Oriented Architectures (SOA). However, none of the existing languages has achieved the status of de facto standard for that purpose until now. In this paper we compare three existing proposals to specify Web Services choreographies: WSCI, WS-CDL, and OWL-S. First, we describe the main characteristics of each one of these languages, and after that we compare the different structures of the three languages. Finally, we present some conclusions of our work.

1. Introduction

The importance of Service-Oriented Architectures (SOA) has grown in the last years because they allow the integration of software applications between different organizations. In these architectures, applications are exposed as services, and these services are interconnected through the use of a set of standards (SOAP, WSDL, UDDI, WS-Security,...). This is the reason because standardization is one of the main aspects of SOA. While a certain level of maturity has been achieved in the adoption of standards to interconnect and describe Web Services, there are still challenges related to the business processes executed by Web Services compositions.

The terms *orchestration* and *choreography* refer to two different ways of describing Web Services compositions. Orchestration languages always represent the composition from the viewpoint of the parties involved in this composition. WS-BPEL (Web Service Business Process Execution Language, [1]) is the most adopted language for that purpose.

On the other hand, the target of choreography languages is the coordination of long-running interactions between multiple distributed parties, where each one of the parties uses Web Services to offer his externally accesible operations. Choreography languages depict the composition from a global viewpoint, showing the interchange of messages between the involved parties. However, there is not an only standard that has been widely adopted for that purpose until now.

Our goal with this paper, then, is to present a comparative study of three existing languages to specify Web Services choreographies: Web Service Choreography Interface (WSCI, [2]), Web Service Choreography Description Language (WS-CDL, [4]), and Ontology Web Language for Services (OWL-S, [5]).

The rest of the paper is structured as follows: Section 2 shows a general description of WSCI language. Section 3 explains the main features of WS-CDL language. Section 4 provides a brief description of the DAML program and the OWL-S language. Section 5 is devoted to the comparison of the different structures of these languages. Finally, in Section 6, some conclusions are presented.

2. Web Service Choreography Interface (WSCI)

The Web Service Choreography Interface (WSCI, [2]) is an XML-based language to describe the interface of a Web Service participating in a choreographed interaction with other services. This interface shows the flow of messages exchanged by the Web Service. The language has been developed by companies like Sun, SAP, BEA and Intalio.

A WSCI interface describes the observable behavior of only one Web Service. This behavior is expressed by means of temporal and logical dependencies in the flow of mes-

[*]Supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PEII09-0232-7745. The third author is supported by the European Social Fund and the Junta de Comunidades de Castilla-La Mancha (Spain).

sages. For that purpose WSCI includes sequencing rules, correlation, exception handling, and transactions. WSCI also describes the collective message exchange among the Web Services participating in the choreography, providing a global view of the interactions. Therefore, a WSCI choreography consists of a set of interfaces, one for each Web Service taking par of it.

In Figure 1 we can see the architecture of WSCI choreography, where we have a different interface for each Web Service.

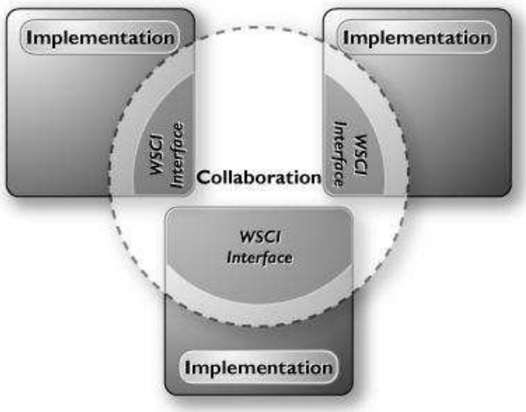

Figure 1. WSCI architecture

The internal implementation of the services is not addressed by WSCI, i.e., how they work and how the message are interchanged. The goal of this language is describing the observable behavior of a set of services through interfaces depicting the flow of messages.

WSCI is able to work with other Web Services description languages, specifically Web Services Description Language (WSDL, [3]). In that way, WSDL describes the static interfaces of the services while WSCI depicts the dynamic interfaces, that is, how the services interact with other services. In the most common scenario, a Web Service has only one WSDL interface and multiple WSCI interfaces for multiple contexts. In Figure 2 we can see the relationship between WSCI and WSDL.

WSCI depicts the behavior of the services by means of choreographed activities. These activities can be atomic or complex. Atomic activities are, e.g., waiting a specific amount of time or sending a message. Complex activities are composed of other activities, defining the choreography for this activities (sequential, parallel, conditional,. . .).

WSCI also allows the use of processes. A process in WSCI is a portion of behavior labeled with a name. We can reuse these processes by referencing his name. We can define two kinds of processes: *Top-level processes* that can be referenced from everywhere in the interface, and *nested*

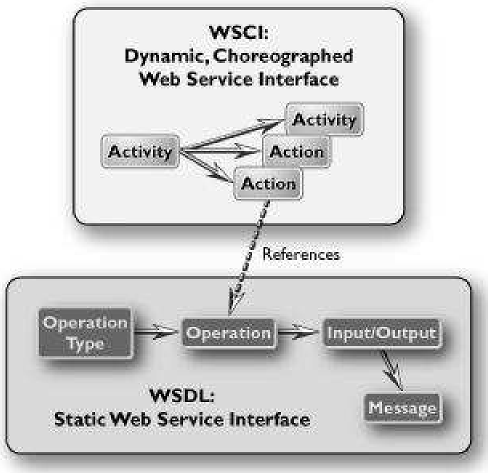

Figure 2. Relationship with WSDL

processes that are defined inside a complex activity and can be referenced only inside this activity.

Contexts are used to describe the environment within which a set of activities is executed. These contexts include the set of declarations available to the activities, the set of possible exceptions and the behavior related to these exceptions, and the transactional properties associated with the activities, including the compensations to undo these activities.

WSCI uses a mechanism called correlation to associate a message with a concrete conversation. Multiple conversations can be distinguished through the use of different correlation instances. Properties of a concrete correlation are communicated as part of messages exchanges.

To sum up, WSCI allows us to specify the observable behavior of a Web Service in a concrete interaction with other services. To obtain a global model with WSCI, we have to consider a set of interfaces (one for each service) and the mapping between the operations existing in each interface.

3. Web Service Choreography Description Language (WS-CDL)

The Web Service Choreography Description Language (WS-CDL, [4]) is an XML-based language to describe peer-to-peer collaborations of Web Services taking part in a choreography. This description defines, from a global viewpoint, the common behavior of the services, and the ordered message interchanges make reaching a common business goal possible.

The goal of specifying Web Services choreographies is composing peer-to-peer interactions between any kind of services, regardless of the programming language or the environment that host the service. In Figure 3 we can see a model of Web Services integration using WS-CDL.

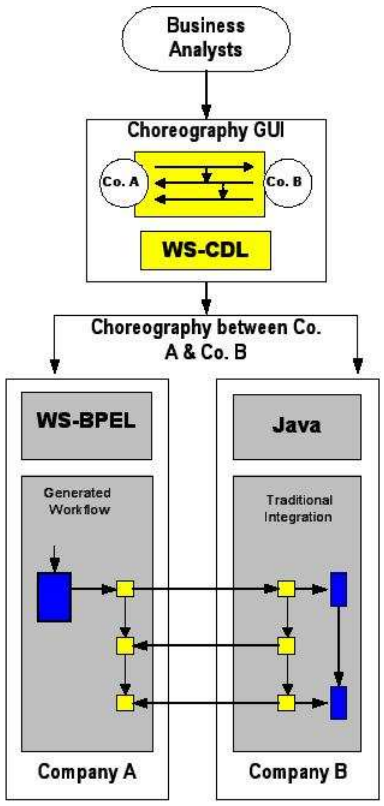

Figure 3. Integration of Web Services using WS-CDL

In WS-CDL the collaboration between Web Services takes place within a set of agreements about the ordering and constraint rules, through which messages are exchanged between participants.

Choreography modeling with WS-CDL consists of the following elements:

- *Roles, Relationships and Participants*: Information is always exchanged between participants within a choreography. A participant groups all the parts of the collaboration that must be implemented by the same entity. A relationship identifies the mutual obligations that must be fulfilled in a collaboration to succeed. A role enumerates a potential behavior of a participant within an interaction.

- *Types, Variables and Tokens*: A type defines the kind of information corresponding to a variable or a token. A variable contains information about the common objects in a collaboration. A token is an alias to reference part of a variable.

- *Choreographies*: A Choreography defines collaborations between participants using the following means:

  – *Choreography Composition*: It allows the creation of new choreographies by means of reusing already defined choreographies.

  – *Choreography Life-Line*: It defines the development of a collaboration. A choreography starts within a business process, then some work is performed and, finally, it finishes normally or abnormally.

  – *Choreography Recovery*: It consists of *exception blocks*, that specify additional interactions that must be performed when an abnormal behavior happens, and *finilizer blocks*, that specify additional interactions which could modify or undo the effect of a previously executed choreography.

- *Channels*: A channel is a point of collaboration between participants specifying where and how information is exchanged.

- *WorkUnits*: A WorkUnit describes the constraints that must be fulfilled to execute some interactions.

- *Interactions*: An interaction is the base of a choreography, describing message interchanges between participants and any related synchronizations of states and variables.

- *Activities and Ordering Structures*: An activity (including interactions) is the lowest level element of a choreography that performs some work. An ordering structure combines activities and other ordering structures to build complex activities. Ordering structures include sequence, choice, and parallel.

- *Semantics*: Semantics allow the creation of descriptions with the semantic definitions of any component of the model.

WS-CDL also includes support to reference WSDL definitions of Web Services.

4. Ontology Web Language for Services (OWL-S)

The Ontology Web Language for Services (OWL-S, [5]) was originally known as DAML-S. The objective of the DARPA Agent markup Language (DAML) program is the development of a language and tools that facilitate the concept of Semantic Web [6]. As part of this program, the Web Services ontology OWL-S has been developed. The aim of this ontology is to automate the discovery, invocation, composition, interoperation and monitoring of Web Services. This ontology has been developed by Carnegie Mellon University, Nokia, Stanford University, SRI International, Yale University,. . .

In Figure 4 we can see the ontology for Web Services proposed by OWL-S. This ontology is based on providing three essential kinds of information about the services:

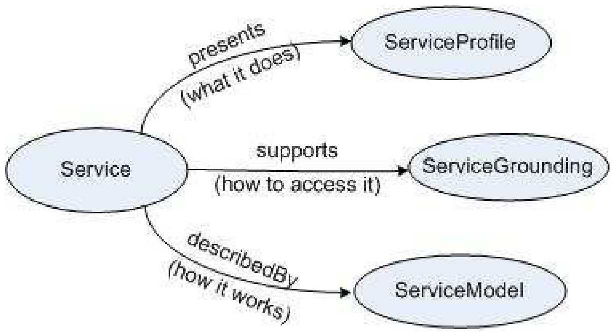

Figure 4. Web Services ontology

- *What does the service provide?* This information is given by the **Service Profile**.
- *How is the service used?* This information is given by the **Service Process Model**.
- *How to access the service?* This information is provided by the **Service Grounding**.

Briefly, the Service Profile provides the information that agents need to discover the service, while the Service Process Model and the Service Grounding give the information that agents need to use the service.

Although OWL-S defines an ontology for each one of these three areas, it also allows the definition of alternative approaches. The default approaches are only basic approaches but are useful in the majority of cases.

In OWL-S each service is considered as a set of atomic processes with inputs and outputs associated. In that way, when the mapping from abstract definition to concrete utilization must be done, OWL-S is complemented with the use of WSDL for the concrete definition of services. In Figure 5 we can see the relationship between OWL-S and WSDL.

Apart from the ontologies described before, OWL-S defines another ontology for the required resources. This ontology covers the description of physical resources, temporal resources and computational resources related to the services.

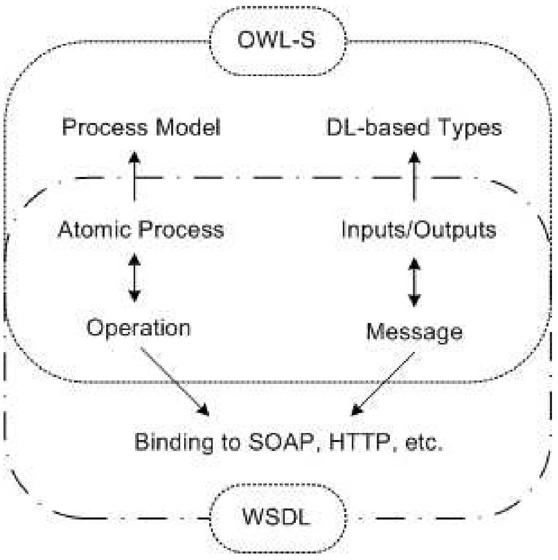

Figure 5. Relationship to WSDL

To sum up, the objective of OWL-S is to provide a base ontology for the description of Web Services within the DAML program. This ontology will be the starting point to automate the discovery, invocation, composition and monitoring of Web Services.

5 Comparison

5.1 Basic Structures

WSCI In WSCI the basic activities are called *atomic activities* and the *action* element is the main one. This element describes the way in which Web Services use an elementary operation within a context, e.g., the exchange of a message with another Web Service. The syntax of an action is:

```
<action
   name = NCName
   operation = QName/NCName
   role = QName
   {any attribute with non-WSCI namespace}>
   Content: (documentation?,correlate*,call?,
            {any element with non-WSCI namespace}*)
</action>
```

The *operation* attribute can be used to reference a WSDL operation that the action performs. An action can be associated with one of the following kinds of WSDL operations:

- **Notification**: The service sends a message to another service.
- **Solicit-response**: The service sends a message to another service and waits for a response.

- **One-way**: The service receives a message.

- **Request-response**: The service receives a message and sends a response.

The *role* attribute is an optional attribute that associates an action with a role name. It can be used to reference the definition of a role given by some other specification.

The *correlate* element is used to relate an action to a correlation definition. It serves to indicate in which particular execution context is performed the action, allowing us to correlate a message with a particular conversation. The syntax of this element is:

```
<correlate correlation = QName
      instantiation = (true|false):false />
```

The *correlation* attribute is mandatory and it references a correlation specification, while the *instantiation* attribute is optional, and can have value **true** (the correlation properties forming the correlation identity will be used to identify the current execution context in all subsequent message exchanges) or value **false** (the correlation properties are used to identify a previously established execution context in which the action should be performed).

The *call* element is used to indicate the activities that will happen while an action that handles a request-response operation is performed by a Web Service. The syntax of this element is the following:

```
<call
   process = NCName>
   Content: (documentation?)
</call>
```

This element is forbidden for all WSDL operations apart from request-response.

Finally, the extensibility of the action element allows us to refer to operations defined in a specification other than WSDL. This can be done by using extension attributes provided by WSCI.

WS-CDL In WS-CDL the basic building block of a choreography is the *interaction* element. It indicates information exchanges between participants, possibly including the synchronization of some information values. These interactions are performed when one participant sends a message to another participant in the choreography. When the message exchanges complete successfully, the interaction completes normally. The syntax of the interaction is the following:

```
<interaction  name="NCName"
              channelVariable="QName"
              operation="NCName"
              align="true"|"false"?
              initiate="true"|"false"?>

   <participate  relationshipType="QName"
                 fromRoleTypeRef="QName"
                 toRoleTypeRef="QName"/>

   <exchange  name="NCName"
              faultName="QName"?
              informationType="QName"?|
                  channelType="QName"?
              action="request"|"respond">
     <send      variable="XPath-expression"?
                recordReference="list of NCName"?
                causeException="QName"?/>
     <receive   variable="XPath-expression"?
                recordReference="list of NCName"?
                causeException="QName"?/>
   </exchange>*

   <timeout time-to-complete="XPath-expression"
            fromRoleTypeRecordRef="list of NCName"?
            toRoleTypeRecordRef="list of NCName"?/>?

   <record  name="NCName"
            when="before"|"after"|"timeout"
            causeException="QName"? >
     <source  variable="XPath-expression"? |
              expression="XPath-expression"?/>
     <target  variable="XPath-expression"/>
   </record>*
</interaction>
```

First, we take a look at the fifth initial attributes:

- The *name* attribute specifies the name of the interaction.

- The *channelVariable* attribute specifies the channel variable used to do the communication during the interaction. It contains information about the participants in the interaction.

- The *operation* attribute specifies the name of operation that is associated with the interaction.

- The *align* attribute, if **true**, indicates that after the interaction both participants act on the basis of their shared understanding for the messages exchanged and the information recorded.

- The *initiate* attribute, if **true**, indicates that the interaction is a choreography initiator.

The *participate* element specifies the relationship type the interaction participates in, and the requesting and accepting participants.

The *exchange* element is used to exchange information during the interaction. It includes the following attributes:

- The *name* attribute specifies the name of the exchange.

- The *faultName* attribute, if specified, indicates the exchange as a fault exchange with the given name.

- The *informationType* attribute and the *channelType* attribute specify the information type or the channel type of the information exchanged between the two participants.

- The *action* attribute specifies the direction of the exchanged information, i.e., **request** or **respond**.

The *send* element and the *receive* element inside the exchange element indicate that information is sent from a participant or information is received at a participant respectively. These elements can also specify the variables exchanged, and if an exception must be thrown.

The *timeout* element allows us to specify the maximum amount of time to complete an interaction, by means of the *time-to-complete* attribute. When this time is exceeded, a timeout occurs. This element also allows us to modify some records in both participants when the timeout occurs.

Finally, the *record* element is used to create or change the value of one or more variables. This element includes the following attributes:

- The *name* attribute specifies the name of the record.

- The *when* attribute specifies when the recording happens (before an exchange, after an exchange, or when a timeout happens).

- The *causeException* attribute, if specified, indicates that an exception may be caused and the value of the attribute identifies the exception that may be caused.

The *source* and *target* elements within the record specify the recordings of information happening in the interaction. The target is always a variable, while the source can be a variable or an expression.

OWL-S In OWL-S services are modeled as processes. These processes are specifications of the ways clients may interact with services. For that purpose, OWL-S includes a subclass of the Service Model called *Process*. This class is defined as follows:

```
<owl:Class rdf:ID="Process">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#AtomicProcess"/>
    <owl:Class rdf:about="#SimpleProcess"/>
    <owl:Class rdf:about="#CompositeProcess"/>
  </owl:unionOf>
</owl:Class>
```

We can distinguish three different kinds of processes: atomic processes, simple processes, and composite processes. *Composite processes* correspond to activities that require multiple service interactions, so we only talk about atomic and simple processes in this section.

Atomic processes are executed in a single step and never have subprocesses. They just receive an input message, do some work, and finally send an output message. There are always only two participants for that kind of process, the **client** and the **server**:

```
<owl:Class rdf:ID="AtomicProcess">
  <owl:subClassOf rdf:resource="#Process"/>
</owl:Class>

<owl:Class rdf:about="#AtomicProcess">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasClient"/>
        <owl:hasValue rdf:resource="#TheClient"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#performedBy"/>
        <owl:hasValue rdf:resource="#TheServer"/>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Simple processes have also a single step execution. They are used as abstractions, providing a view of some atomic process. In this case, the simple process is **realizedBy** the atomic process:

```
<owl:Class rdf:ID="SimpleProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="realizedBy">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#AtomicProcess"/>
  <owl:inverseOf rdf:resource="#realizes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="realizes">
  <rdfs:domain rdf:resource="#AtomicProcess"/>
  <rdfs:range rdf:resource="#SimpleProcess"/>
  <owl:inverseOf rdf:resource="#realizedBy"/>
</owl:ObjectProperty>
```

In Figure 6 we can see the relation between the simple processes and the atomic processes.

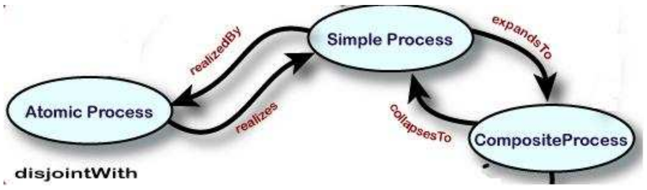

Figure 6. Relationship between processes

Finally, we must take into account that processes can have two different goals:

1. They can return some new information based on some given information. These processes are based on **inputs** and **outputs**.

2. They can produce a change in their environment. These processes are based on **preconditions** and **effects**.

There are several classes defined in the OWL-S model related to these four elements (inputs, outputs, preconditions, and effects).

Discussion The three languages have basic structures to describe the message exchange between parties in a composition, but there are several differences between the elements used for that purpose. While the *interaction* element in WS-CDL allows us to exchange multiple messages between two parties (in both directions), the *action* element in WSCI and the *atomic process* in OWL-S refer to a single exchange.

The *interaction* element in WS-CDL pays special attention to the variables exchange between the different parties in each *exchange* element, while the *action* element in WSCI only specifies the operation performed by the message. In OWL-S there is a list of inputs and outputs related to each *atomic process*.

Finally, each *action* element in WSCI only specifies one of the roles participating in the exchange, the sender or the receiver. The *connect* element is used in the global model to relate a send message to a receive message from different interfaces. The *interaction* element in WS-CDL specifies both roles, indicating which one is the requesting participant and which one is the accepting participant. In OWL-S the *atomic processes* always have two properties to indicate which role is the client of the service and which role is the server. In this aspect, WS-CDL and OWL-S are more powerful than WSCI, in the sense that they need less code to express a collaboration between two parties.

5.2 Complex Structures

WSCI In WSCI *complex activities* contain a set of activities and define the order in which these activities are performed. A complex activity can contain one or multiple activity sets. All the complex activities are based on a common definition with the following syntax:

```
<{activity type}
   name = NCName>
   Content: (documentation?)
</{activity type}>
```

Activities are always performed in some context and the attribute *name* is an optional attribute that can be used to distinguish this activity from any other activity in the same context.

Next, we are going to see in more detail each one of the complex activities that are used to determine the order in which a set of activities is performed:

- The *all* activity performs the whole set of activities that contains in any order, possibly in parallel. It has the following syntax:

```
<all
   name = NCName>
   Content: (documentation?,context?,
            {any activity}*)
</all>
```

- The *sequence* activity performs all the activity set in sequential order. It has the following syntax:

```
<sequence
   name = NCName>
   Content: (documentation?,context?,
            {any activity}+)
</sequence>
```

- The *choice* activity performs only one activity set from the collection of multiple activity sets within this complex activity. The decision is made based on events. The event can be the reception of a message, the expiration of a timeout, or the throwing of a fault. When multiple events overlap, there is no way to know which one of the possible activity sets is executed. The choice activity has the following syntax:

```
<choice
   name = NCName>
   Content: (documentation?,
          (onMessage|onTimeout|onFault){2,n})
</choice>

<onMessage>
   Content: (documentation?,action,
            context?,{any activity}+)
</onMessage>

<onTimeout
   property = QName
   type = (duration|dateTime) : duration
   reference = QName>
   Content: (documentation?,context?,
            {any activity}+)
</onTimeout>

<onFault
   code = QName>
   Content: (documentation?,context?,
            {any activity}+)
</onFault>
```

- The *foreach* activity executes the activity sets within repeatedly. It has the following syntax:

```
<foreach
   name = NCName
   select = expression>
   Content: (documentation?,context?,
            {any activity}+)
</foreach>
```

The *select* attribute is an XPATH expression that evaluates to a list of items. The activity set is repeated once for each item in this list. If the list is empty, the activity set is not performed.

- The *switch* activity selects one activity set from the collection of multiple activity sets within this complex activity based on the evaluation of conditions. It has the following syntax:

```
<switch
   name = NCName>
   Content: (documentation?,case+,default?)
</switch>

<case>
   Content: (documentation?,condition,
          context?,{any activity}+)
</case>

<default>
   Content: (documentation?,context?,
            {any activity}+)
</default>

<condition
   {extension attribute}>
   Content: {expression}
</condition>
```

All the *case* elements are mutually exclusive, selecting the corresponding activity set if the value of the condition is true for that case. Only one case can be executed, so if multiple case elements can be performed, the first one in the definition has the biggest priority. If no other condition is fulfilled, the activity set within the *default* element is performed.

The *condition* element is an XPATH expression that evaluates to a Boolean value. This condition is evaluated in the context of the switch activity.

- The *until* activity performs the activity set that contains repeatedly based on a Boolean condition. The until activity is repeated one or more times because the condition is evaluated after each iteration of the activity set. If false the activity set is repeated, otherwise the activity ends. It has the following syntax:

```
<until
name = NCName>
Content: (documentation?,condition,
          context?,{any activity}+)
</until>
```

- The *while* activity performs the activity set that contains repeatedly based on a Boolean condition. The while activity is repeated zero or more times because the condition is evaluated before each iteration of the activity set. If true the activity set is executed, otherwise the activity ends. It has the following syntax:

```
<while
name = NCName>
Content: (documentation?,condition,
          context?,{any activity}+)
</while>
```

WS-CDL In WS-CDL we can distinguish two different kinds of *complex activities* inside a choreography: the workunit element and the ordering structures.

The *workunit* element specifies a condition that must be fulfilled in order to perform some work and/or the repetition of some work. It completes successfully when the set of activities inside completes successfully. This element has the following syntax:

```
<workunit  name="NCName"
      guard="xsd:boolean XPath-expression"?
      repeat="xsd:boolean XPath-expression"?
      block="true|false"? >

      Activity-Notation

</workunit>
```

The *Activity-Notation* refers to the set of activities performed within the workunit.

The optional attribute *guard* is an XPATH expression that specifies the condition that must be fulfilled to perform the workunit.

The optional attribute *block*, with false value as default, indicates whether the element have to block waiting for the "true" evaluation of the guard condition or it skips the activities inside when the guard condition evaluates to "false".

The optional attribute *repeat* is also an XPATH expression that specifies the repetition condition of the workunit. It is always not blocking.

In this way, when there is not guard condition specified then it is considered to be always true while when there is not repetition condition specified then the workunit is not considered to be executed again after one execution.

Ordering structures are used to combine basic activities and other complex activities in a nested way, expressing the order in which actions are performed within the choreography. There are three ordering structures:

- The *sequence* ordering structure expresses that the set of activities inside must be executed sequentially. It has the following syntax:

```
<sequence>
    Activity-Notation+
</sequence>
```

- The *parallel* ordering structure indicates that the set of activities inside must be executed concurrently. It completes successfully when all the concurrent activities complete successfully. The syntax of this ordering structure is:

```
<parallel>
    Activity-Notation+
</parallel>
```

- The *choice* ordering structure specifies that only one of multiple activities can be executed. If the choice have workunits inside, only the first one in lexical order with a "true" guard condition is selected. If there are other activities, there is no way to know which one is selected; it is considered as a non-observable decision. The choice has the following syntax:

```
<choice>
    Activity-Notation+
</choice>
```

OWL-S As we have seen in Section 5.1, Web Services in OWL-S are modeled as processes and there are three different kinds: atomic processes, simple processes, and composite processes. *Composite processes* contain other processes of any kind in a nested way. They also specify the way in which their contents are executed, such as sequence or choice.

Composite processes are specified as follows:

```
<owl:Class rdf:ID="CompositeProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
  <owl:disjointWith rdf:resource="#SimpleProcess"/>
  <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Process"/>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#composedOf"/>
         <owl:cardinality rdf:datatype="...">
                  1</owl:cardinality>
      </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

The **composedOf** property is used to specify the control construct corresponding to the composite process, i.e., the way in which their contents are executed. It is also used to associate a composite process with its nested processes.

Next, we are going to see the different control constructs provided by OWL-S:

- The *Sequence* control construct specifies a list of subprocesses to be executed in a row. It has the following definition:

```
<owl:Class rdf:ID="Sequence">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructList"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

  The *ControlConstructList* contains the list of subprocesses to be executed in sequence.

- The *Split* control construct specifies a set of subprocesses to be executed concurrently. This process completes as soon as all his subprocesses has begun their execution. Split has the following definition:

```
<owl:Class rdf:ID="Split">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

  The *ControlConstructBag* contains the set of subprocesses to be executed in parallel.

- The *Split+Join* control construct also specifies a set of subprocesses to be executed concurrently, but in this case the process completes when all its subprocesses have finished. This control construct has the following definition:

```
<owl:Class rdf:ID="Split-Join">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

  Again, the *ControlConstructBag* contains the set of subprocesses to be executed in parallel.

- The *Any-Order* control construct specifies a set of subprocesses to be executed in any order but not concurrently. The process completes when all its subprocesses have finished. Any-Order has the following definition:

```
<owl:Class rdf:ID="Any-Order">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This control construct also uses the *ControlConstruct-Bag* to specify the subprocesses that contains.

- The *Choice* control construct specifies a set of subprocesses and only one of them is executed. The selection criteria are is non-observable, so any of the subprocesses can be chosen. Choice has the following definition:

```
<owl:Class rdf:ID="Choice">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This kind of composite process also uses the *Control-ConstructBag* to specifiy all the posible subprocesses.

- The *If-Then-Else* control construct executes different subprocesses depending on the value of a condition. It has the following definition:

```
<owl:Class rdf:ID="If-Then-Else">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="ifCondition">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="then">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="else">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

The **ifCondition** property specifies the condition we have to test. The **then** property specifies the subprocess we execute if the condition is “true”, whereas the **else** property specifies the subprocess we execute when the condition is “false”.

- The *Iterate* control construct repeats the execution of its components an undetermined number of times. It is only used as a superclass of Repeat-While and Repeat-Until constructs. Iterate has the following definition:

```
<owl:Class rdf:ID="Iterate">
  <rdfs:subClassOf
   rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty
       rdf:resource="#components"/>
      <owl:allValuesFrom
       rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

- The *Repeat-While* control construct iterates the execution of a subprocess while a condition evaluates to true. This condition is always evaluated before the execution. This control construct has the following definition:

```
<owl:Class rdf:ID="Repeat-While">
 <rdfs:subClassOf rdf:resource="#Iterate"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="whileCondition">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="whileProcess">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

The **whileCondition** property specifies the condition we test, and the **whileProcess** property specifies the subprocess we execute repeatedly.

- The *Repeat-Until* control construct iterates the execution of a subprocess as long as a condition evaluates to true. This condition is always evaluated after the execution, so at least one execution is done. This control construct has the following definition:

```
<owl:Class rdf:ID="Repeat-Until">
  <rdfs:subClassOf rdf:resource="#Iterate"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="untilCondition">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="untilProcess">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

The **untilCondition** property specifies the condition we test after each execution, and the **untilProcess** property specifies the subprocess we execute repeatedly.

Discussion In Table 1 we show the equivalences between the different complex structures we have described corresponding to each language.

The *sequence* construction exists in the three languages with the same meaning, the sequential execution of the activities or processes within the construction.

The *choice* construction also exists in the three languages but there are some differences. While in WSCI the selection is based on the triggering of an event, in WS-CDL and OWL-S the selection criteria for the activities inside are non-observable. WS-CDL also allows us to use workunits within the choice, restricting the possible selections to the workunits that match their guard condition.

A construction to indicate the concurrent execution of several activities also exists in the three languages, but with different names. The *all* construction in WSCI indicates that the activities are executed in parallel or in any order (but not concurrently). The *parallel* construction in WS-CDL indicates that the activities are executed concurrently (the specification says nothing about a possible execution in any order but not concurrently). Last, in OWL-S we have three different constructions for that purpose: The *Split* process indicates the concurrent execution of all its subprocesses without waiting for the completion of these subprocesses, the *Split+Join* process also indicates the concurrent execution of all its subprocesses but it waits for the completion of them, and the *Any-Order* process indicates the execution in an undefined order of the subprocesses but not concurrently.

The *switch* construction, that selects one activity from a collection, only exists in WSCI, but it can be emulated in WS-CDL and OWL-S by using some other construction. In WS-CDL we can use multiple non-blocking *workunits* with the guard condition specified, while in OWL-S we have the *If-Then-Else* process and with several of these processes nested we can achieve the same behavior.

Both, the *until* construction in WSCI and the *Repeat-Until* process in OWL-S, indicate the repetition of the contents based on the truth value of a condition, where the condition is evaluated at the end of each iteration. This behavior can be emulated in WS-CDL by using a *workunit* with a repeat condition specified.

We also have the *while* construction in WSCI and the *Repeat-While* process in OWL-S indicating the repetition of the contents based on the truth value of a condition, but with the condition evaluated before each iteration. In this case, an emulation in WS-CDL can be done by using a *workunit* with a guard and a repeat condition specified (both evaluating the same condition).

Finally, the *foreach* construction only exists in WSCI and there is not any equivalent construction in the other two languages.

5.3 Exception handling and compensation

WSCI In WSCI we can specify exceptional behavior that can be reached from any point of the Web Service choreography. The declaration of this behavior is part of the context definition and it includes the activities that the Web Service performs in response to each exception.

WSCI allows us to declare three kinds of exception:

- The reception of a message considered as exceptional.
- The occurrence of a fault.
- The expiration of a timeout.

After the exceptional activities have been performed, the current context terminates and the parent context is resumed. In this way, WSCI allows us to treat exceptions that do not cause the termination of the overall choreography (such as the throw/catch in Java programs). If there is not any treatment for an exception in the current context, this context terminates and the exception is raised to the parent context for treatment (such as the propagation of exceptions to parent classes in Java programs).

The *fault* activity is used to trigger a fault in the current context. It has the following syntax:

```
<fault
   name = NCName
   code = QName>
   Content: (documentation?)
</fault>
```

The *name* attribute specifies the name of the fault and the *code* attribute is used to specify the code corresponding to the fault. The exception element which onFault event handler matches the name attribute of this fault will be triggered when the fault occurs. If the fault is not handled by any exception handler, in the current context or in any parent context, the process terminates with the fault code.

WSCI	**WS-CDL**	**OWL-S**
sequence	*sequence*	*Sequence*
choice (events)	*choice* (non-observable)	*Choice* (non-observable)
all (concurrent or unspecified order)	*parallel* (concurrent)	*Split*, *Split+Join*, and *Any-Order*
switch	Multiple *workunits* with guard conditions	*If-Then-Else*
Until	*workunit* with repeat condition	*Repeat-Until*
While	*workunit* with guard and repeat conditions	*Repeat-While*
foreach	–	–

Table 1. Equivalences between complex structures

The *exception* element is used to handle exceptional behavior. This element must specify one or more event handlers, each one of them defining the event handled and the activity set to perform when this event occurs. It has the following syntax:

```
<exception>
   Content: ((onMessage | onTimeout | onFault){+})
</exception>
```

We can distinguish three different kinds of event handlers:

- The *onMessage* event handler is triggered by an incoming message and its initial action indicates the event that triggers this event handler.
- The *onTimeout* event handler is triggered when a timeout expires.
- The *onFault* event handler is triggered when a fault occurs. It has an optional attribute *code* specifying the fault code. If absent, the event handler will be triggered by all faults for which no other event handler has been specified.

When an exception has occurred, WSCI allows us to undo some work by using the *compensation* element in a transaction. It has the following syntax:

```
<compensation>
   Content: (documentation?, context?,
             {any activity}+)
</compensation>
```

The *compensate* element is used inside the exception handler to reference the compensation we want to execute. This element is defined as follows:

```
<compensate
   name = NCName
   transaction = NCName>
   Content: (documentation?)
</compensate>
```

The *transaction* attribute is used to indicate the name of the transaction whose compensation we want to execute.

WS-CDL Different types of exceptions are considered in WS-CDL. The exceptions considered include the following categories:

- **Interaction failures**: E.g. the sending of a message fails.
- **Protocol based exchange failures**: E.g. no acknowledgement is received as part of the behavior of a protocol.
- **Security failures**: E.g. a message is rejected because it has not valid digital signature.
- **Timeout errors**: E.g. an interaction is not completed in the specified amount of time.
- **Validation errors**: E.g. an XML message is not well formed.
- **Application failures**: E.g. an Internet purchase service is out of stock of a product offered.

Exception workunits can be defined to handle all these exceptions. They may also be used as the mechanism to recover from the exceptions. The exception workunits are defined within the *exceptionBlock* element of a choreography. It has the following syntax:

```
<exceptionBlock  name="NCName">
   WorkUnit-Notation+
</exceptionBlock>?
```

At least one exception workunit must be defined. The guard of the workunit can be used to specify the particular type of exception we want to handle through the use of the *hasExceptionOccurred* function. The exception workunit with no guard condition is called the default exception workunit and only one is allowed within an exception block.

Only one exception workunit can match each exception. If multiple exception workunits are defined, the order of evaluating them is based on the order in which the workunits have been defined. When the matching happens, the actions of the matched workunit are executed. If no matching happens and a default exception workunit exists, then the actions of this workunit are executed. Otherwise, the exception is raised in the parent choreography.

WS-CDL also allows us to define finalization actions within a choreography that can confirm or cancel the effects of this choreography, so we can use this actions for compensation. This finalization is done by means of the *finalizerBlock* element. It has the following syntax:

```
<finalizerBlock  name="NCName">
   Activity-Notation
</finalizerBlock>
```

Multiple finalizer blocks can be defined with different *name* attributes, but only one of them is executed when the choreography completes successfully.

Finally, if an exception occurs, the choreography completes unsuccessfully and the actions within it are completed abnormally. Furthermore, the finalizer blocks of the choreography are not executed.

OWL-S The OWL-S specification does not provide any explicit support for exception handling. However, some work about exception handling in Semantic Web Services is being developed as part of the Darpa program. In [7] an approach for the specification of exception handling and recovery of Semantic Web Services based on OWL-S is presented. Here, we therefore describe this specification.

The following failure categories can be distinguished when executing a Web Service composition, according to [7]:

- **Service invocation errors**: Communication failures, response timeout, ...
- **OWL-S processing errors**: Problems with the syntax or the structure of OWL-S files.
- **Process level execution errors**: Erroneous situations caused by inconsistencies on the process model that may occur during the execution.
- **Application level errors**: Erroneous states specific to the application logic of a Web Service.
- **Constraint violations**: Violations of constrains established in e-contracts between parties.

Several actions can be taken as a response to a failure. We have the following action categories:

- **Neutral actions**: Actions that do not have any effect on the state of a failed process.
- **Recovery actions**: Actions that provide means for restoring the state of a failed process.
- **Fault emitting actions**: Actions that throw an exception as a response to a failure.
- **Termination actions**: Actions that can be used to terminate a process.
- **Adaptation actions**: Actions that modify the execution flow of a service.

A list of *fault handlers* that are used as responses to failures can be defined for each process. These handlers have the following form, in a simple abstract syntax:

```
FaultHandler(FaultType [faultVariable])
  { actions }
```

Fault handlers are strictly local to the process for which they are defined. Multiple handlers can be defined for a process but only the first one that matches in top-down order is executed. If there is not any fault handler that matches, the fault is propagated to the parent process. The optional *faultVariable* can be used to access the fault occurrence and its value.

A list of *constraint violation handlers (CV-handlers)* can be defined in a process to detect hard constraint violations considered as failures. A CV-handler has the following abstract syntax:

```
CV-Handler(event-expression [eventVariable])
  { actions }
```

CV-handlers are active in their own process and in all embedded processes. Triggering a CV-handler causes the termination of the process corresponding to this CV-handler and changes the state of this process to failed state. The optional *eventVariable* can be used to access the event occurrence and its value.

We can also define a list of *event handlers* for each process. These handlers can be used to express soft constraints, i.e., constraints that do not necessarily lead to an erroneous state, and to define the responses to these constraints. An event handler has the following form:

```
EventHandler(event-expression [eventVariable])
  { actions }
```

Triggering an event handler does not lead to the termination of the process for which this event handler is defined. Event handlers are also active for all the embedded processes. When one event happens, all the event handlers that match are executed in top-down order. The optional *eventVariable* can be used to access the event occurrence and its value.

Finally, the *compensation* construct is used in a process to define the actions that can be executed for undoing the effects of this process. It has the following abstract syntax:

```
Compensation { actions }
```

A compensation can be activated by calling the *compensate* action or the *compensateProcess* action. It is only executed when the process has finished successfully, i.e., the process is not in failed state. If there is not compensation specified for a composite process, it is compensated by executing compensations of embedded processes finished in the reverse chronological order of their original invocation. If there is not compensation for an atomic process, it is skipped during the compensation execution, but a specific event called *NoCompensationForAProcess* is thrown.

Discussion Exception handling and compensation are included as part of WSCI and WS-CDL specifications, but the latest version of OWL-S does not include any of these aspects. We know that a specification about exception handling and compensation is being developed as part of the Darpa program, but this specification is still in its infancy.

These three languages propose an exception treatment based on events and the bottom-up propagation of exceptions in nested structures. WS-CDL does not make any distinction in the treatment of the different kinds of event that can cause the exception, while WSCI differentiates between three kinds of event (message, timeout, and fault) and the specification related to OWL-S differentiates between another three kinds of event (faults, hard constraint violations, and soft constraint violations). However, the three languages take into account the same types of exception (timeout, abnormal behavior of a service, violation of a constraint, . . .).

Finally, the compensation process is very similar in the three languages. The main difference is that compensation in WS-CDL can only be defined at the choreography level, whereas in WSCI we can define a compensation for each transaction and the specification related to OWL-S allows us to define a compensation for each process.

6. Final Discussion

Both, WSCI and WS-CDL, are W3C proposals, but WSCI last update was released in 2002, so it has not got received any attention in the last years. On the other hand, WS-CDL is the ongoing standardization initiative for Web Service choreography, but it has not achieved the status of being accepted as the *de facto* standard for that purpose. Apart from these two proposals, we have the OWL-S language as a part of the emerging Semantic Web, so its success is closely related to the consolidation of this framework worldwide in the future, which is not clear now.

As we can see in [8], some people think that the reason because none of the choreography standardization efforts has been adopted by a wide user base is that Service-Oriented Architectures (SOA) has not gained enough maturity until now. They think that some issues have to be solved before we reach the adoption of a SOA infrastructure that integrates choreography (the identification of patterns for service interactions, the definition of a service interaction meta-model, . . .). However, the current choreography languages can be seen as a starting point to reach these goals. For example, the elements of a service interaction meta-model will be very similar to the elements we have in WS-CDL.

Timing restrictions are used very often in the composition of Web Services, being a critical issue in real-time systems. For example, we want to indicate the amount of time we wait for the confirmation of a purchase order. In WS-CDL and WSCI time constraints can be specified by using the timeout element and the timeout event, respectively, but the specification of OWL-S says nothing about these restrictions. Nevertheless, several efforts have been devoted to extend OWL-S with a time ontology [9, 10].

The use of a formal language to describe a Web Service choreography facilitates the validation of compositions by applying validation techniques already defined for this formalism. Only WS-CDL of the three languages we are comparing is based on a formal language (π-calculus) [11], but there is not a clear translation from all the elements of WS-CDL into π-calculus, so we cannot apply any validation technique directly. The scientific community has developed multiple translations of these three languages into different formal representations [12, 13, 14]. However, all these proposals only take into account a subset of the elements of each language, so they cannot guarantee full correctness of the given specifications.

Concerning the relation with other standards, these three choreography languages are XML-based and can work together with the WSDL language, using this well-established standard to describe the Web Services participating in the composition. WS-CDL and WSCI do not cover the description and execution of the workflow corresponding to each service in the composition, so we are free to use different mechanism for each one of these services, such as WSFL (Web Services Flow Language, [15]) and WS-BPEL. On the other hand, OWL-S intends to cover this work and extensions like OWL-WS (OWL for Workflow and Services, [16]) has been developed for that purpose. Finally, we also have to take into account that OWL-S builds on OWL (Ontology Web Language, [17]), so it makes use of some of the ontologies defined by this language.

References

[1] T. Andrews et al. *Business Process Execution Language for Web Services (version 1.1).* Technical report, may 2003.

[2] A. Arkin et al. *Web Service Choreography Interface (WSCI) 1.0.* http://www.w3.org/TR/wsci/.

[3] E. Christensen et al. *Web Services Description Language (WSDL) 1.1.* http://www.w3.org/TR/wsdl.

[4] N. Kavantzas et al. *Web Service Choreography Description Language (WSCDL) 1.0.* http://www.w3.org/TR/ws-cdl-10/.

[5] D. Martin et al. *OWL-S: Semantic Markup for Web Services.* http://www.w3.org/Submission/OWL-S/.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. *The Semantic Web.* Scientific American, 284(5):34–43, 2001.

[7] Roman Vaculín, Kevin Wiesner, and Katia Sycara. *Exception handling and recovery of semantic web services.* In Fourth International Conference on Networking and Services. IEEE Computer Society Press, 2008.

[8] A. Barros, M. Dumas, and P. Oaks. *Standards for Web Service Choreography and Orchestration: Status and Perspectives.* In Proceedings of the 1st International Workshop on Web Service Choreography and Orchestration for Business Process Management at the BPM 2005, Nancy, France, 2005.

[9] F. Pan and J. R. Hobbs. *Time Ontology in OWL.* http://www.w3.org/2001/sw/BestPractices/OEP/Time-Ontology.

[10] F. Pan and J. R. Hobbs. *Time in OWL-S.* In Proceedings of AAAI-04 Spring Symposium on SemanticWeb Services, Stanford University, California, 2004.

[11] S. Ross-Talbot. *Web Services Choreography and Process Algebra.* SWSL Committee: Working Materials, 2004.

[12] , G. Díaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero *Automatic Translation of WS-CDL Choreographies to Timed Automata.* In Proceedings of WS-FM, Versalles, September, 2005.

[13] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. *Formalizing Web Service Choreographies.* In Proceedings of First International Workshop on Web Services and Formal Methods. Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[14] JunFeng Wu and HuaiKou Miao. *A Rewriting Logic Approach to OWL-S Composite Process Formal Specification.* APSCC,pp.343-348, 2008 IEEE Asia-Pacific Services Computing Conference, 2008.

[15] Frank Leymann. *Web Services Flow Language (WSFL) Version 1.0.* IBM Software Group, May, 2001.

[16] Stefano Beco, Barbara Cantalupo, Ludovico Giammarino, Mike Surridge, and Nikolaos Matskanis. *OWLWS: A Workflow Ontology for Dynamic Grid Service Composition.* In 1st IEEE InternationalConference on e-Science and Grid Computin, Melbourne, December, 2005.

[17] S. Bechhofer et al. *OWL Web Ontology Language.* http://www.w3.org/TR/owl-ref/.