

A Model of Interactions about Actions for Active and Semantic Web Services

Yasmine Charif¹ Nicolas Sabouret²

¹CRIP5 45, rue des Saints Pères, 75270 Paris Cedex 06
Yasmine.Charif@etu.univ-paris5.fr

²LIP6 8, rue du Capitaine Scott, 75015 Paris
Nicolas.Sabouret@lip6.fr

Abstract This paper proposes an interaction model to define online web services that can interoperate with other web services in order to exchange information about their actions. These web services are known as semantic since they integrate in a single representation both the structural data description and the action description. Thus, they are able to answer formal requests about their own functioning. We show how this model can integrate currently emergent standards in web services description (WSDL, SOAP, *etc.*) and take into account the proactive nature of active components in the semantic web. We then propose an interoperability mechanism based on reasoning about action's operational semantics.

Keywords: Semantic web services, Interaction architecture for web services, Language for process modeling, Reasoning about actions, Ontological engagement, VDL.

1 Presentation of the problem

1.1 Agents and web services in the semantic web

The Semantic Web [2] aims at defining an extension of the current Web in which information is given well-defined meaning, better enabling agents to assist human users in information processing. In this vision, users interact with services that manipulate the information. Such services are characterized by a *functioning* [9] and, unlike 'CGI scripts' which simply process, manipulate or produce information in the web, this functioning is *part of the information*. As a consequence, the user must be able to request the service about its actions (figure 1). Moreover, services themselves must be able to reason about their actions and to interact with other services to query them about their functionalities, behavior and activity. All this requires the service to be able to access a semantic description of its own actions.

Semantic Web Services (SWS) aim at providing such information about actions. The emergence of standards for SWS description, publication, invocation and composition (WSDL [3], WSMF [6] or OWL-S [4]) makes services interoperable. However, SWS do not take into account the need for user assistance as

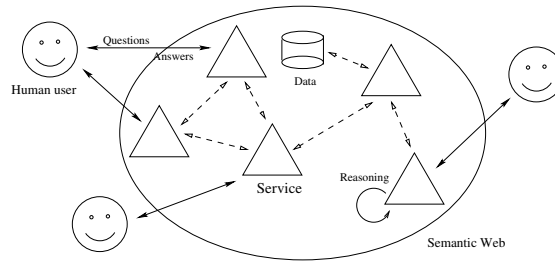


Figure 1. Interactions between users and service in the Semantic Web.

mentioned above. The challenge of Semantic Web Services *in user assistance* is therefore to allow agents to explain to human or agent users *what they do, how they do it* and *why*.

In this paper, we try to propose an architecture for web services that takes into account interaction with both human users and other services, based on an execution model for agents [15]. This raises two main issues with respect to interoperability of services. At the communication level, we want our model to be compatible web services standards such as WSDL. At the semantic level, we want services to be able to understand data and action descriptions sent by other services using ontologies.

1.2 Previous work

In [15], we proposed a programming model and an execution platform for Active and Semantic Web Services (ASWS), *i.e.* software components on line in the Internet (compatible with web services standards) provided with a representation of their own operation (and of its semantics). These services are known as *active* because, like agents, they can run without any interaction (as opposed to traditional services which are only reactive). Our aim is to define online services that interact through HTML web pages with the user and with other services using web services description standards. These services must also be able to answer in natural language to the user's questions (figure 2). From the human user's point of view, the ASWS are thus active web pages.

This model is based on the View Design Language (VDL) [14], an ASWS programming language in XML. This language integrates within a single representation the component's program (action and interaction capabilities), the data it manipulates and its semantics. Thus, the formalism that describes the operational part of such a component is also the model in which it handles its internal representation. This allows human users to question directly the underlying structures of the program, without having to know by advance this tool's possibilities.

In this context, a *query language* was defined to model a wide class of questions that the human users can ask (through a *chatbox*) about the state, activity and behavior of the ASWS [16].

1.3 Plan of the paper

In the next section, we present the models our work is based upon. Section 3 shows existing standards and some related work about these two problems. We present on section 3 how WSDL is used to publish ASWS capacities and allow them to exchange information about their actions. Section 4 shows how we solved intercomprehension problems by producing automatically actions ontologies. Section 5 concludes and gives current research perspectives.

2 Related work

2.1 The VDL model

The execution model ASWS are defined using a specific model: VDL (which stands for *View Design Language*). VDL is an AI-oriented language that aims to programme active software components (a.k.a agents) capable of:

1. interacting with both human users (through a graphical interface, e.g. a web page) and other agents (using web services standards).
2. representing their own actions and reasoning about them at runtime in order to explain to human or agent users what they do, how they do it and why.

The VDL model is based on XML-tree rewriting. The agent's code is an XML tree where some nodes define actions to perform, preconditions or data to manipulate. We call this tree the agent's view: A view is both the state of the VDL agent at time t and the program of this agent. At each execution step, the VDL interpreter looks within the tree for a specific elements corresponding to actions description. It then rewrites the tree according to these elements.

It is possible to provide VDL actions with two kinds of preconditions corresponding to two kind of behavior:

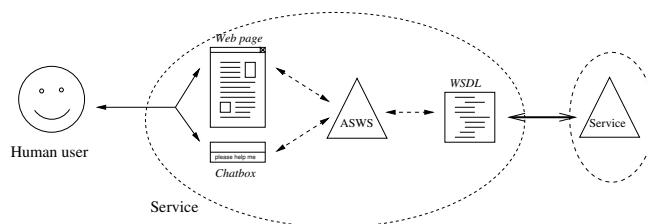


Figure 2. Interaction model for human users and ASWS.

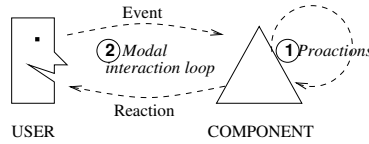


Figure 3. Modeless and Modal execution.

- Reaction (modal behavior): the service performs operations in response to interactions from the user (a typical example is a start/stop operation).
- Proaction (modeless behavior): the service run independently from any user interaction (e.g. a cooking process, once launched, does not need further prompts to work).

Figure 3 illustrates this principle.

We showed in [14] that three basic actions are required to programme all possible transformations on a VDL tree: **add** to append elements to the content of a given element in a tree; **put** to replace the content of an element; **del** to remove an element from the tree. Moreover, these actions can be provided with two kinds of preconditions corresponding to the two execution modes: the default mode is proaction but the process can be provided with boolean guards. For reaction, we use external events which are XML elements sent *at runtime* to the agent or service.

Programming interactions in VDL The programmer can define three kinds of interactions in VDL using specific basic actions:

1. External events to trigger a reaction from another ASWS, using **send-event**. When an ASWS *A* wants to send an external event *E* to another service *B*, it must contain the following code:

```
<send-event>
  <dest> B </dest>
  <event> E </event>
</send-event>
```

The *dest* element contains the receiver’s address. This address is composed of:

1. The ASWS local identifier. It must be declared using a **name** element;
2. The user’s identifier, corresponding to its ASWS cookie as explained in [15]. It is declared using the **user** element;
3. The URL of the server the ASWS runs into. It is declared within a **server** element.

Moreover, an ASWS can send an event to all the services running on the stated servers with the stated users using the specific name “*all*”. One or more *dest*

and *event* elements can be declared within the same *send-event* basic action (in which case all stated recipient will receive all stated external events).

2. Actions to send information about actions, using **send-action**. When an ASWS *A* wants to send the action *C* to a service *B*, it must use the following basic action:

```
<send-action> <dest>B</dest> <action>C</action> </send-action>
```

C is a syntactically correct VDL action that can contain any basic action, precondition or sub-action, *i.e.* even another message sending action. Several *dest* and *action* elements can be declared.

3. Requests to question another ASWS about its data or actions or to answer a request [16]. When an ASWS *A* wants to send the request *R* to a service *B*, it must use the following basic action:

```
<send-request> <dest>B</dest> <request>R</request> </send-request>
```

The request *R* is the XML representation of a formal request as defined in [16], *i.e.* it is a sextuple $\langle \alpha, \tau, \nu, \sigma, \omega, \delta \rangle$ where α is the speech act performative [17], τ defines the kind of behavior considered (reaction, processes, *etc*), σ is the request's subject and ω the object (arguments). For example, the formal representation of the question “*What are you doing?*” is as follows:

```
<request>
  <act> what </act>
  <type> do </type>
  <subject> view </subject>
</request>
```

The VDL envelope For ASWS communication, we defined a specific VDL envelope inspired from KQML [7] and FIPA-ACL [1]. The VDL envelope is the basic structure emitted while interactions between VDL agents. It allows its recipient to better identify the message it contains (in terms of its transmitter, the interaction identity, *etc.*):

```
( sender : the sender url,
  receiver : the receiver url,
  content : the message content,
  reply_with : single identifier allotted to the interaction,
  in-reply-to : identifier of the interaction at the origin of this one
)
```

where the field *in-reply-to* makes the distinction between an unsolicited assertion and an assertion made in reply to a query. To make ASWS interoperable, the VDL envelope they use for message sending must be included in a standard protocol such as SOAP and WSDL, as presented in section 3.

Example Here is a simple example of an ASWS in VDL that we will use in this paper. It prepares your bath at home when requested and warns you (or your assistant agent) when the bath is ready.

```

<view><name>Bath-tub </name>
  <description>Prepares you a bath at home</description>
  <bath-must-run/> <sender/>
  <bath-level>0</bath-level>
  <bath-temperature>35</bath-temperature>
  <bath-ready>false</bath-ready>
  <action>
    <name>Start bath </name>
    <event><start-bath/></event>
    <add><path><sender/></path> <get-senders/></add>
    <put><path><bath-must-run/></path> <value>true</value></put>
  </action>
  <action>
    <name>Fill bath-tub </name>
    <guard><get><bath-must-run/></get></guard>
    <guard><not><get><bath-ready/></get></not></guard>
    <put><path><bath-level/></path>
      <plus><get><bath-level/></get><value>1</value></plus>
    </put>
    <guard><equals><get><bath-level/></get><value>15</value></equals></guard>
    <put><path><bath-ready></path><value>true</value></put>
  </action>
  <action>
    <name>Inform that bath is ready </name>
    <guard><get><bath-ready/></get></guard>
    <send-event>
      <event><bath-ready/></event>
      <dest><get><sender/></get></dest>
    </send-event>
  </action>
</view>

```

The agent *Bath-tub* contains:

- four variables *bath-must-run*, *bath-level*, *bath-temperature*, *bath-ready* and *list-of-bath-requesters*;
- a *Start bath* action that records in *list-of-bath-requesters* each requester sent the event *<start-bath/>* and sets *bath-must-run* to *true*;

The specific keyword *get-senders* is used by the VDL interpreter to capture the VDL address of external events senders.

- a *Fill bath-tub* action that increases *bath-level* at each execution step, until 15, whenever *bath-must-run* is evaluated to *true* and *bath-ready* is evaluated to *false*. When *bath-level* reaches 15, *bath-ready* is put at *true*;

– and an *Inform that bath is ready* action that sends the event `<bath-ready/>`, as soon as *bath-ready* is evaluated to *true*, to all the requesters recorded in *list-of-bath-requesters*.

2.2 Web service description in WSDL

Our aim is to define ASWS that interact with other services using web services description standards. As a consequence, the interface of this web service must be described in WSDL (*Web Service Description Language*) [3] in order to be published onto UDDI¹ repository. WSDL is a language based on XML to define the services operations and to describe how to reach them. It is used to describe the service web, to specify its localization and to describe the operations that it exposes (and finally to facilitate its interoperability). It introduces the **port**, **binding** and **message** concepts to describe the services:

- **WSDL Port** are logical name that identify a group of operations and describe how to reach the service web using a specific protocol. SOAP [18] is the protocol more often coupled with WSDL to provide to the client application the necessary tools to call upon distant services.
- **WSDL Binding** are specification of the access type, the transport protocol used and the parameters coding in the messages for a given port type.
- **WSDL Message** defines the XML data structure for the interaction, either in input (the client calls an operation) or in output (the service answers).

Thus, a VDL action will be described in the *WSDL port* part, and the detailed description of its in/out events in the *WSDL message* part.

2.3 Ontologies for service interaction

Supposing they have the same vocabulary, two services may use different identifiers for what is in fact the same concept. A solution to this issue is to provide services with collections of information called *ontologies*. An ontology is an explicit knowledge-level specification of a shared conceptualization, *i.e.* it is a set of the distinctions that are meaningful for the service or process. Exhaustive typologies of ontologies were developed, in particular by [8,10]. We will retain the two main categories that [12] distinguished:

- **General/Common ontologies** which include the vocabulary related to the objects, the events, time, space, function, behavior, causality, *etc.*
- **Task ontologies** which provide a vocabulary of the terms used to solve problems associated to tasks that can belong or not to the same field.

We want to use ontologies in our interaction model in VDL so as to solve the inter-comprehension constraint between services when they manipulate concepts and actions. We need to provide *every ASWS* with a task ontology. Section 4 shows how our interaction model takes into account this constraint, why and how it handles actions using task ontologies and how it maintains the service code optimal during actions exchanges.

¹ <http://www.uddi.org/>

3 Interactions between ASWS

Our first aim is to make ASWS interoperable with other web services. In order to do this, we need to build a WSDL description of the service interaction capabilities automatically from its VDL code.

3.1 Using WSDL to publish VDL agent capacities

As soon as the service instance² begins to run, the VDL interpreter generates a WSDL description of the ASWS interaction capabilities. This description is specific to the instance and made accessible to other services. Since the description of these functionalities is integrated within the VDL code, they can be modified at runtime.

The general algorithm for producing a WSDL description is as follows:

1. In the VDL code of the service, the VDL interpreter looks for the actions that are triggered by external events (**event** precondition) and that send messages in response (**send-event**, **send-action** or **send-request** basic action).
2. In no action corresponds to these two criteria, the VDL interpreter looks for actions that merely send messages. Let A_f be such an action. The interpreter tries to link A_f to other actions that are triggered by external events and, in response, that activate A_f at term. This is done using backward chaining on the actions effects. The interpreter thus builds a set of sets $\{A_s, A_1, A_2, \dots, A_f\}_k$ that will be seen as a general WSDL operation whose input are the precondition events and the output the sent messages.

The backward chaining algorithm is too complex to be presented in detail here. The general idea is to look for all variables used within the boolean preconditions of A_i and then search the whole tree for actions that modifies it, that is to say the variable appear within a **path** element³. This makes the possible A_{i-1} .

Each possible $\{A_s, \dots, A_f\}_k$ defines a WSDL operation.

3. The WSDL description is generated and added to the WSDL file as follows:
 - The action with its input and outputs is declared in **WSDL:portType**. Input and outputs are defined in **WSDL:message** specifying their type (event, action or request). Types are declared in **WSDL:type**.
 - For each action (operation), the corresponding SOAP action is defined in **WSDL:binding** and the encoding of the input and output are specified.

3.2 Example

As an example, let us consider the *bath-tub* agent defined in section 2.1. The *Inform that bath is ready* action produces an event sending as a result. Its precondition depends on *bath-ready* which is handled by the *Fill bath-tub* action.

² Each human or agent user interacts with its own instance of the ASWS. However, difference instances can interact together.

³ A preliminary algorithm was presented in [14].

This one is started by an external event, but the VDL interpreter tries nevertheless to look for other actions approaching the source. It follows the chaining with the variable *bath-must-run* used in the *Fill bath-tub* precondition, and handled by the *Start bath* action that is started by an external event. Now that no action remains, the interpreter deduced a general action that is started by the event `<start-bath/>` and that sends the event `<bath-ready/>` as a result. This corresponds to the following WSDL code:

```

<wsdl:definitions name="Bath-tub"
  targetNamespace="http://example.com/Bath-tub.wsdl"
  xmlns... (xsd,vql,soap,wsdl + bath-tub (wns)) >
  <!-- Input: event <start-bath/>, Output: event <bath-ready/> -->
  <wsdl:message name="BathCall">
    <part name="start-bath" type="vql:event"/>
  </wsdl:message>
  <wsdl:message name="BathReturn">
    <part name="bath-ready" type="vql:event"/>
  </wsdl:message>
  <!-- Definition of the general action "to prepare a bath" -->
  <wsdl:portType name="Bath-tubPortType">
    <wsdl:operation name="StartBath">
      <wsdl:input message="wns:BathCall"/>
      <wsdl:output message="wns:BathReturn"/>
    </wsdl:operation>
  </wsdl:portType>
  <!-- Binding definition -->
  <wsdl:binding> ... </wsdl:binding>
  <!-- Service definition -->
  <wsdl:service name="Bath-tubService">
    <wsdl:documentation>Prepares you a bath at home</wsdl:documentation>
    <wsdl:port name="Bath-tubPort" binding="wns:Bath-tubBinding">
      <soap:adress location="http://example.com/Bath-tub"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

3.3 Publishing requests and actions inputs

In addition to the usual capacity declaration that defines the set of VDL external events the service can process, we want the ASWS to publish the fact that it can:

1. receive actions from another agent to learn new behaviors;
2. process requests about its own actions, behavior and activity [16].

Since these capabilities are managed at the architecture level, they do not appear in the VDL code. As a consequence, the produced WSDL description does not mention them.

The publication of these capacities by the WSDL description generator consists in adding two operations to the WSDL description: "*Send-Action*" for receiving actions and "*Send-Request*" for receiving requests and returning answers⁴:

```

<!-- Receiving an Action -->
<wsdl:message name="ActionCall">
  <part name="Action" element="vql:Action"/>
</wsdl:message>

<!-- Receiving a Request and answering by a Request -->
<wsdl:message name="RequestCall">
  <part ref="Request"/>
</wsdl:message>
<wsdl:message name="RequestReturn">
  <part ref="Request"/>
</wsdl:message>

<!-- Definition of Send-Action and Send-Request capacities -->
<wsdl:portType name="VDLPortType">
  <wsdl:operation name="Send-Action">
    <wsdl:input message="wns:ActionCall"/>
  </wsdl:operation>
  <wsdl:operation name="Send-Request">
    <wsdl:input message="wns:RequestCall"/>
    <wsdl:output message="wns:RequestReturn"/>
  </wsdl:operation>
</wsdl:portType>

```

Thus, other services know they can send to an ASWS a VDL action or a request (in addition to the ASWS own operations) and that they will be answered with another request (with `Act="Assert"`).

3.4 Including the VDL envelope in SOAP messages

From a WSDL-compatible service's point of view, the VDL envelope (section 2.1) is embedded within SOAP messages, so that the *sender*, *reply-with* and *in-reply-to* values are declared in the SOAP message header, and the message *content* (*i.e.* the external event, VDL action description or VDL request about actions, depending on the kind of interaction) in the SOAP message body.

Here is an example of a SOAP message sent by a service S_1 to ask the *bath-tub* service to prepare a bath:

```

POST /bath-tub HTTP/1.1
Host: http://www.bath-tub.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: http://www.bath-tub.com/Bath-tub#Bath

```

⁴ The *request* type definition, corresponding to the model described in [16], is given in Appendix.

```

<env:Envelope xmlns:env="http://www.w3.org/2001/09/soap-envelope">
  <env:Header>
    <i:interaction xmlns:i="http://example.org/2001/06/tx"
      env:mustUnderstand="1">
      <sender>Coco</sender>
      <receiver>Mike</receiver>
      <reply-with>req-122</reply-with>
      <in-reply-to/>
    </i:interaction>
  </env:Header>
  <env:Body>
    <m:BathCall xmlns:m="http://www.bath-tub.com/bath-tub">
      <start-bath/>
    </m:BathCall>
  </env:Body>
</env:Envelope>

```

Since the message's content is defined as an *vgl:event* type in the WSDL description, this content (`<start-bath/>`) is interpreted as an external event. Thus, the (tomcat) server sends this event to the *bath-tub* ASWS for processing. This event is added to the event's queue and is processed by next execution step, which starts the *start-bath* action.

Minutes later, the *bath-tub* service sends the following message to warn S_1 that the bath is ready:

```

HTTP/1.1 200 OK    [...]
<env:Envelope xmlns:env="http://www.w3.org/2001/09/soap-envelope">
  <env:Header>
    <i:interaction xmlns:i="..." env:mustUnderstand="1">
      <sender>Mike</sender>
      <receiver>Coco</receiver>
      <reply-with>req-583</reply-with>
      <in-reply-to>req-122</in-reply-to>
    </i:interaction>
  </env:Header>
  <env:Body>
    <m:BathReturn xmlns:m="http://www.bath-tub.com/bath-tub">
      <bath-ready/>
    </m:BathReturn>
  </env:Body>
</env:Envelope>

```

This SOAP message is built by the VDL interpreter using the *send-event* VDL instruction in the ASWS's code.

4 Ontologies and actions in VDL

In previous section, we added a WSDL layer on the top of our VDL interaction model and used the SOAP protocol to be compatible and interoperable with other services. Now, we aim at defining *semantic* services in order to enable automatic composition and interaction between services. To this purpose, many approaches in the SWS domain propose to describe actions in a domain-specific ontology (OWL-S [4], WSMF [6], *etc*). On the contrary, we want to respect a *minimal ontological engagement* [8] to avoid the service programmers to define action ontologies but rather generate them later automatically from the VDL description.

In this section, we propose an interaction model based on the fact that every ASWS is provided with a *task* ontology (*see* section 2.3) that is used to understand not only data that is exchanged but also other service's *actions* described in VDL.

4.1 Ontological engagement

A minimal ontological engagement means:

- an agreement on the shared vocabulary used in a coherent and consistent way,
- that requests and assertions must use the vocabulary defined in the ontology,
- that the ontological engagement can be minimized by specifying the weakest theory and by defining only the essential terms for coherent knowledge communication with the theory.

The VDL programmer must then define a task ontology (in our case with Protégé⁵) for each VDL service. Each ontology must gather all the data the service manipulates. The software application we are developing authorize the programmer to load his ASWS onto the server if and only if he correctly built its ontology (*i.e.* all the concepts handled by the service must be defined in its ontology). The service's ontology is defined using Protégé and must have the same identifier as the ASWS, except it ends with *.pont* (**P**rotégé **O**ntology) instead of *.vdl*. No knowledge of the VDL language is required when defining the task ontology. However, the VDL programmer must know of the concepts it can use in its service's definition.

When two ASWS interact with each other, they need to share the same ontology so as to understand each other. Current research in the domain propose to build this ontology by integrating both ontologies into a single one [5,11,13]. However, this currently remains an open problem even when restraining to data ontologies as we proposed. For the purpose of our study, we will suppose that concepts handled by VDL services have the same names if and only if they have the same meaning. Concrete controls we make are done, for the moment, on the

⁵ Protégé <http://protege.stanford.edu>

correspondence level between the concepts of an ASWS (*i.e.* the XML elements it manipulates) and those of its ontology.

Our minimal ontological engagement states that services do not agree on actions' names. However, this must not prevent actions to remain comprehensible for other ASWS, especially when sending VDL code through the *send-action* and *send-request* basic actions. Actions are apprehended by the *operational semantics* of the basic actions and preconditions they contain, rather than by the *name* the VDL programmer gave them.

4.2 Actions matching

When receiving an action description in VDL (*e.g.* after a *send-action* or a *send-request* basic action), an ASWS must also be able to adapt its own code so as to take into account the modifications implied by the action. In our interaction model, we aim at maintaining the VDL code optimal during interactions. When an ASWS receives an action after a *send-action*, it must add the action to its own code. However, we would like it to first try to match this action with its own actions. If it finds correspondences, it must pairs the two considered actions.

We say that an element t_1 matches an other t_2 (noted $t_1 \supseteq t_2$) if and only if t_2 appears within the direct or indirect children of t_1 , or t_2 is t_1 itself. Let consider two actions A_1 and A_2 with preconditions $precond_1$ (resp. $precond_2$) and sub-actions $action_1$ (resp. $action_2$). When the ASWS compares the received action A_2 with its own action A_1 , it can find correspondences between either the preconditions, the sub-actions or both. It then builds a new VDL description to replace the action A_1 (otherwise, the ASWS adds directly the received action), according to the following algorithm:

- If $precond_1 = precond_2$, A_1 will be replaced by:

```
<action>
  <precond1/>
  <action1/>
  <action2/>
</action>
```

- If $precond_1 \supseteq precond_2$, A_1 will be replaced by:

```
<action>
  <precond2/>
  <action2/>
  <action> <precond1/><action1/> </action>
</action>
```

- If $action_1 = action_2$, A_1 will be replaced by:

```
<action>
  <guard> <or><precond1/><precond2/></or> </guard>
  <action1/>
</action>
```

4.3 Example

Suppose we want to write a service S that asks the bath-tub to send the bath temperature when it is ready. This can be easily done by sending the following VDL action:

```
<send-action> <dest><name>bath-tub</name></dest>
  <action> <guard><get><bath-ready/></get></guard>
    <send-event> <dest><name>S</name></dest>
      <event><get><bath-temperature/></get></event>
    </send-event>
  </action> </send-action>
```

Using the action matching algorithm, the *bath-tub* service detects that the new action has the same preconditions as the *Inform that bath is ready* action. As a consequence, it will simply add the “send-event” basic action to the content of this action.

5 Conclusions and perspectives

This paper proposes a programming model for active and semantic web services capable of interoperation with other online web services or agents in order to exchange information about their functioning. These agents interact both with human users (through HTML web pages) and with other agents (through WSDL description) for both command (by sending events) and control (by sending actions or requests that model questions about actions, activity and behavior of other services or agents) included in SOAP messages. We are currently implementing this model in the already existing ASWS platform in VDL. The source code of the VDL interpreter, demonstrations about VDL web services and documentation are available on line⁶.

Our contribution is only a preliminary work that opens many wide perspectives and challenges for web services and agents in the semantic web:

- We first intend to use UDDI repositories for service discovery.
- We must also validate the task ontologies produced by our model *w.r.t.* different web services.
- We want to use the operational semantics of VDL so as to produce OWL-S actions descriptions for any ASWS for automatic composition with services defined using other models.
- We would like VDL agents to systematically re-send a request to which they did not have answer when there is a change in their environment (for example, the arrival of a new agent).

Our aim is to use the functionalities offered by the ASWS model (proactive interaction with human users or services using the web) together with the possibility to question the agents about their actions (using the VDL request model) in order to ease collaboration between web services for a task realization, as stated in Tim Berner’s Lee vision.

⁶ <http://www-poleia.lip6.fr/~sabouret/demos>

References

1. The FIPA ACL Message Structure Specifications. <http://www.fipa.org/specs/fipa00061/>, 2002.
2. T. Berners-Lee. *Weaving the Web*. Harper San Francisco, 1999.
3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). <http://www.w3.org/TR/wsdl>, 2001.
4. DAML-S Coalition. *DAML-S: Web Service Description for the Semantic Web*, 2002.
5. A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to Map between Ontologies on the Semantic Web. In *Proc. 11th International WWW Conference*, 2002.
6. D. Fensel and C. Bussler. The Web Services Modeling Framework WSMF. In *1st meeting of the "Semantic Web enabled Web Services workgroup"*, 2002.
7. T. Finin, R. Fritzson, and D. McKay. An overview of KQML : A Knowledge Query and Manipulation Language. Technical report, University of Maryland Baltimore Country, 1992.
8. N. Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organisation, Extraction, and Integration. *A multidisciplinary approach to an emerging information technology. International summer school*, pages 139–170, 1997.
9. Z. Guessoum and J.P. Briot. From Active Objects to Autonomous Agents. *IEEE Concurrency*, 7(3):68–76, 1999.
10. G. Van Heijst, A. Schreiber, and B. Wielinga. Using Explicit Ontologies in KBS development. *Internatinal Journal Human-Computer Studies*, 46:183–292, 1997.
11. M. Klein. Combining and Relating Ontologies: an Analysis of Problems and Solutions. In *Proc. 17th International Joint Conference on Artificial Intelligence, Workshop: Ontologies and Information Sharing, Seattle, USA*, 2001.
12. R. Mizoguchi, J. Vanwelkenhuysen, and M. Ikeda. Task ontology for reuse of problem solving knowledge. In *Proc. 2nd international conference on building and sharing of very large-scale knowledge bases*, 1995.
13. N. Noy and M. Musen. Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proc. AAAI-00 Conference. Austin. USA*, 2000.
14. N. Sabouret. *Étude de modèles de représentation, de requêtes et de raisonnement sur le fonctionnement des composants actifs pour l'interaction homme-machine*. PhD thesis, Université Paris-Sud, 2002.
15. N. Sabouret. Active Semantic Web Services: A programming model for agents in the semantic web. In *Proc. EUMAS*, 2003.
16. N. Sabouret and J.P. Sansonnet. Automated Answers to Questions about a Running Process. In *Proc. CommonSense 2001*, pages 217–227, 2001.
17. J.R. Searle. *Speech Acts*. Cambridge University Press, 1969.
18. W3C. Simple object access protocol (soap). <http://www.w3.org/TR/SOAP>, 2000.

Appendix

The following XML schema defines the *request* type for use within WSDL description.

```
<wsdl:type><xsd:schema>

  <!-- Definition of a Request -->
  <element name="Request"><complexType>
    <element name="Act" element="vql:Act"/>
    <element name="Type" element="vql:Type"/>
    <element name="Subject" element="vql:Subject"/>
    <element name="Object" element="vql:Object"/>
    <element name="Mark" element="vql:Mark"/>
    <element name="Date" element="vql:Date"/>
  </complexType></element>

  <!-- Definition of Act -->
  <element name="Act"><complexType><choice>
    <element name="What"/>
    <element name="How"/>
    <element name="Why"/>
    <element name="Ask"/>
    <element name="Assert"/>
  </choice></complexType></element>

  <!-- Definition of a Type -->
  <element name="Type"><complexType><choice>
    <element name="is"/>
    <element name="do"/>
    <element name="can"/>
    <element name="order"/>
  </choice></complexType></element>

  <!-- Definition of a Subject -->
  <element name="Subject" type="elementOnly"/>

  <!-- Definition of a Object -->
  <element name="Object" type="elementOnly"/>

  <!-- Definition of a Mark -->
  <element name="Mark"><complexType><choice>
    <element name="true"/>
    <element name="false"/>
  </choice></complexType></element>

  <!-- Definition of a Date -->
  <element name="Date" type="elementOnly"/>

</xsd:schema></wsdl:type>
```