OWL-S Ontology Framework Extension for Dynamic Web Service Composition

Jing Dong
Computer Science Department
University of Texas at Dallas
Richardson, TX 75083, USA
jdong@utdallas.edu

Yongtao Sun
American Airline
4333 Amon Carter Blvd
Fort Worth, TX 76155, USA
Yongtao.Sun@aa.com

Sheng Yang
Computer Science Department
University of Texas at Dallas
Richardson, TX 75083, USA
syang@utdallas.edu

Abstract

Composing existing web services for enterprise applications may enable higher level of reuse. However the composition processes are mostly static and lack of support for runtime redesign. In this paper, we describe our approach to the extension of the OWL-S ontology framework for dynamic web service composition. We raise the level of abstraction and propose an abstract service layer so that web services can be composed at the abstract service level instead of the concrete level. Each abstract service is attached with an instance pool including all instances of the abstract service to facilitate fail-over and dynamic compositions.

1. Introduction

Recent advance on web service computing enables building business processes and systems through the discovery and integration of the existing services. Current web services are typically described in terms of atomic and composite web services, using languages like BPEL [5] or OWL-S [2] which provide mechanisms for web service compositions. However, the processes of web service compositions tend to be static in the sense that these processes are normally generated off-line. Any changes to the part of a process may result in the reconfiguration of the whole process. It lacks the support of the capability of fail-over and dynamical redesign. This is especially critical for real-time enterprises since the systems cannot afford to stop, reconfigure, and restart. If some web services of a composition fail or the requirements change, the system needs to be able to change locally and reconfigure on-the-fly.

Among the numerous available web services, there are many that provide similar services. Even with different implementations, most of them present a similar interface to the end user. However, there is currently little effort on abstracting these similar services into high-level common services. Although the OWL-S language provides a way to describe the hierarchical relationship between services, the recommended ontology framework is still limited to one root-level abstract service. Raising the level of abstraction and capturing similar services as a service pool are important, especially for dealing with fail-over and dynamic redesign in real-time enterprises and e-business.

In this paper, we propose an extension to the ontology framework based on OWL-S, which enables defining the composite services at the abstract service level. We provide new constructs to specify such higher level of abstraction. Our approach also includes a service instance pool that allows filtering and plugging in candidate services at runtime. In addition, we offer a planner prototype based on Java Theorem Prover (JTP) [10] that can automatically generate the composition processes on-the-fly.

The rest of this paper is structured as follows. Section 2 gives an architecture overview of our approach. Section 3 defines our extension to the OWL-S Ontology Framework. We describe a prototype planner in Section 4. The last two sections present the related work and conclusions.

2. Architecture Overview

Figure 1 presents an architecture overview of our approach that focuses on the definition of abstract services based on both existing web service instances and the user's expected goals. We define an extension to the OWL-S recommended ontology framework for this purpose. We first define an abstract service hierarchy by grouping the available concrete services into different categories based on their functional characteristics, such as input and output, and specific nonfunctional service parameters. Multiple-level inheritance hierarchy is enabled to represent the hierarchical relationships among these abstract services. A concrete service (existing OWL-S service) can be plugged into the appropriate levels. For example, a concrete flight service AA301 (a flight from American Airline which departs from New York JFK airport and arrives in Chicago ORD airport) is a type of both the abstract American Airline Flight service and abstract JFK-ORD Flight Service which is in turn a type of the abstract Flight service. Since there are thousands of web services already deployed in the Internet and new services available every day which may not be aware of the abstract service hierarchy, we also develop a backend utility program to register a concrete service into a domain service hierarchy and dynamically

update the service ontology when detecting a change. The newly introduced abstract service layer does not require the concrete service to completely conform to the definition of abstract service because our backend utility can automatically detect the equivalent relation between semantic concepts. For example, if one specific abstract Flight service has a “maximumLoad” serviceParameter and a concrete flight service has a “loadLimit” serviceParameter, our utility program considers it a match if the “sameas” relationship has been defined for the “maximumLoad” and “loadLimit” serviceParameters in the ontology. In this paper, we assume that all concrete services of the same abstract service share the same interface information, i.e., they all have the same IOPE parameters, so that we can focus on the abstract service hierarchy. We plan to address different interface mapping in the future.

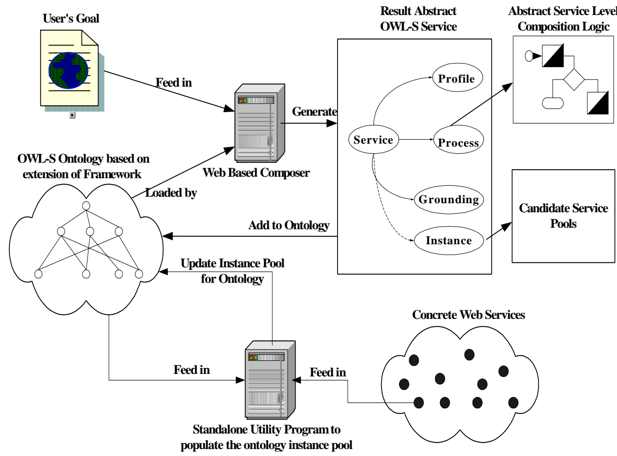

Figure 1 . Architecture Overview

If a composition of services is requested for business or enterprise applications, either an existing abstract service can be matched or a new abstract service is defined with its Profile section containing the input/output and the semantic properties. The Process and new Instance sections can be generated by our composition planner in two steps. First, based on the ontology hierarchy of all available abstract services for existing services and the definition of the user’s goals including both the functional requirement and nonfunctional service parameters, we can obtain the composition process (the Process section of the abstract service that satisfies the user’s goals) using our composition planner based on JTP [10]. This generated composite service matches all the input/output and flow-related semantic requirements. This can narrow down the candidate scope by filtering out all other unrelated web services. Second, other functional service parameters describing the user’s goals are recorded in the Profile section of the abstract service. They are used to identify the actual candidate service pool for composite service and fill in the new Instance section if a direct concrete service candidate is found.

3. Extension to OWL-S Ontology Framework

In this section, we describe our proposed extension to the OWL-S recommended framework. This extension includes the information on the input/output, flow control, semantic property, and candidate instance pool of the abstract service in the ontology hierarchy. This new ontology framework contains the following features:

A new “Instance” section is added to the OWL-S recommended ontology framework. This new section provides the information about the candidate concrete service instances for this abstract service. These service instances can be of a standard web service, an OWL-S or a BPEL service. This Instance section is different from the Grounding section since it does not provide a binding to a physical web service. Instead, it contains a collection of references to the available candidate service instances. If the Instance section of an abstract service is not empty, there is at least one concrete service available for direct invocation. We call all these available concrete services the *instance pool* of the corresponding abstract service. At runtime, the system can pick up a candidate service from the pool and invoke it via its own binding information (like Grounding in OWL-S or WSDL in standard web service). If an abstract service does not have any candidate service instance, it may obtain its instance(s) from the Process section at runtime. For example, a traveler may want a service that allows him to fly from New York to Paris, with a stop at Moscow, called a “NewYork-Moscow-Paris Flight” service, which is not a typical connection flight available from any airline. Thus, the Instance section of the “NewYork-Moscow-Paris Flight” abstract service is empty. The Process section of “NewYork-Moscow-Paris Flight” can specify that the flow of “NewYork-Moscow-Paris Flight” is just a “NewYork-Moscow” service plus a “Moscow-Paris” service. Suppose both “NewYork-Moscow” and “Moscow-Paris” have a number of candidate services in their Instance sections. In this way, a composite concrete service for the abstract service “NewYork-Moscow-Paris Flight ” can be obtained by picking one candidate service instance from the instance pool of “NewYork-Moscow” and another from that of “Moscow-Paris”. The new Instance section and its relationships to other sections are illustrated in Figure 2. More details about the Instance section is presented in Section 3.2.

New constructs are added to the Process section of the recommended framework. In the Process section of the current OWL-S framework, each OWL-S service can only be an Atomic process, a Simple process, or a Composite process. No matter which type of service, it

can only contain one work flow logic. We define a new "Abstract Process" type, which has an "abstractComposedOf" attribute to specify all possible composition processes it can have. In our approach, we can define a collection of composite processes containing multiple work flow logics in the new abstract service. For example, the work flow logics of both "JFK-DFW-LAX" and "JFK-ORD-LAX" services are included in the new abstract "JFK-LAX" service. Either can be used to fulfill a flight service from NewYork JFK airport to Los Angels LAX airport. More details are presented in Section 3.3.

An abstract service does not contain the Grounding information since it is not mapped to any physical service. Instead, it gets the candidate instance from its Instance section which is like a resource pool.

In the existing OWL-S ontology framework, each service in the hierarchy is still a concrete service. With the above two framework extensions, we can define an abstract service layer, so that future service compositions can be made at the abstract level.

From software architecture and design perspective, our approach defines a service architecture that has more complex structure than the simple "subClassOf" relationship in the current OWL-S framework. With the support of abstract services, it is possible to define more complex service structures based on, e.g., inheritance, information sharing, and polymorphism. Therefore, the Profile, Process or Instance of an abstract service is subclasses of the Profile, Process or Instance of its parent service. In the following sections, we present more details of our extension.

3.1 Extension to Root Level OWL-S Framework Ontology

In order to connect the Instance section to its Service, a new predicate "implementedBy" is introduced to the root level OWL-S ontology framework. This links an Instance Class to its Service Class. The Instance Class contains the instance pool information for the abstract Service. Additionally, some new constructs (see Section 3.3) are introduced in ServiceModel which is the parent class of the Process. The new Service class with our extension is shown as follows. Figure 2 shows the visual RDF schema of the Service.owl[1], where our extensions are marked.

```
<!-- Service Implementation -->
<owl:Class rdf:ID="ServiceInstance">
<rdfs:label>ServiceInstance</rdfs:label></owl:Class><owl:Ob
jectProperty rdf:ID="implementedBy">
   <rdfs:domain rdf:resource="&service;#Service"/>
   <rdfs:range rdf:resource="&service;#ServiceInstance"/>
   <owl:inverseOf rdf:resource="&service;#implements"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="implements">
   <rdfs:domain rdf:resource="&service;#ServiceInstance"/>
   <rdfs:range rdf:resource="&service;#Service"/>
   <owl:inverseOf
rdf:resource="&service;#implementedBy"/>
</owl:ObjectProperty>
```

1 In the remainder of this paper, we only show the visual RDF schema without the corresponding RDF file to save space and for better visualization.

- **ServiceInstance**: the root class that represents the instance pool of an abstract service. Its subclass contains all reference information of the candidate instances for that particular abstract service.
- **implementedBy**: An object property extended for a Service class. The resource of this property points to a ServiceInstance.
- **implements**: An object property of a ServiceInstance class. It is an inverse property of implementedBy.

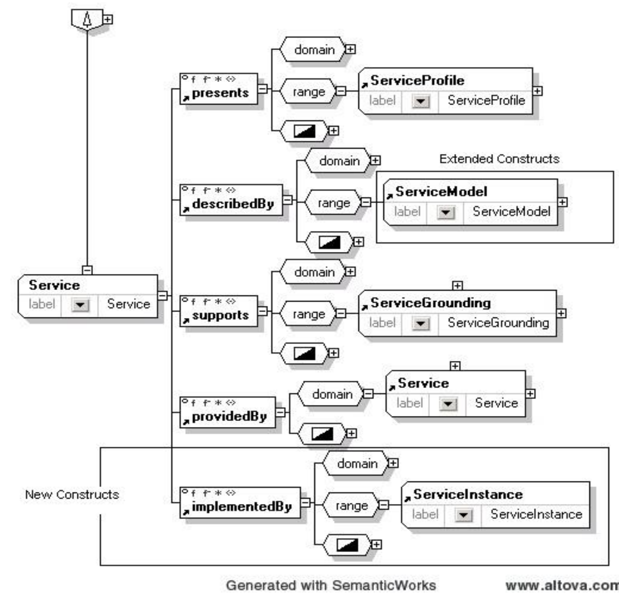

Figure 2 .Visual RDF Schema for Extended OWL-S Upper Ontology

3.2 Instance Class

As discussed previously, the Instance class has a brand new set of information introduced to describe the information of the instance pool of the available concrete service for the corresponding abstract service. It is a subclass of "ServiceInstance" shown in Figure 2. Figure 3 shows the visual RDF schema for the Instance class:

- **Instance**: A subclass of the ServiceInstance shown in Figure 2. It is the root class for all abstract service "Instance". The overall relationship to other ontology framework objects can be found in Figure 2.
- **preferenceOrder**: A data property of the Instance class. It specifies the user's preference when choosing the candidate service. It currently has four

possible values: **Sequential, RoundRobin, Random** and **PriorityCode.**

- **ProcessCandidate**: A class that represents an implementation instance. Each ProcessCandidate points to a real web service in the Internet, such as the "AA301" service mentioned in Section 3.
- **businessOwner**: a data property of the ProcessCandidate class that specifies the owner of concrete web service.
- **priorityCode**: a data property of the ProcessCandidate class that specifies the priorityCode assigned. The value of priorityCode is greater than or equal to 0 with 0 being the highest priority. If the value of preferenceOrder is set to PriorityCode, the system takes this property value to determine which candidate will be chosen at runtime.
- **processRefID**: a data property of the ProcessCandidate class that specifies the id of a concrete web service. It is an optional reference to the physical web service.
- **serviceType**: a data property of the ProcessCandidate class. Currently three types exist: Simple, BPEL and OWL-S.
- **AccessPoint**: A class that represents the access method of a concrete web service.
- **protocolType**: a data property of the AccessPoint class. The possible value can be "HTTP", "FTP" etc.
- **accessLocation**: a data property of the AccessPoint class that specifies the access address of a concrete web service. The system can use this location to retrieve the detailed information of a concrete service.
- **processCandidate:** An object property for an Instance Class. The resource of this property points to a ProcessCandidate class.
- **accessPoint:** An object property for a ProcessCandidate class. The resource of this property points to an AccessPoint class.

3.3 New Constructs in Process.owl

For each abstract service, the composition logic may not be unique. The abstract service needs to have the capability to represent multiple composition logics, which is not possible with the current "Process" class. Therefore, we introduce a new "AbstractProcess" class in the Process section as marked in Figure 4. This class provides a collection of Process logics, like different itineraries from New York to Los Angeles flight example in Section 3. It has an "abstractComposeOf" property pointing to a ControlConstruct with different flow logics. The ControlConstruct maintains a collection of available composition solutions, which can be one process (like an AtomicProcess, a SimpleProcess or a CompositeProcess) or a combination of processes and control constructs.

- **AbstractProcess**: A class represents the possible composition logics. In current OWL-S framework, an actual Process is a subclass of the union of AtomicProcess, SimpleProcess and CompositeProcess. With the introduction of this new AbstractProcess, we are able to describe multiple composition solutions in the Process section of an abstract service.
- **abstractComposedOf:** an object property for AbstractProcess. A collection of ControlConstructs are linked to an AbstractProcess via this abstractComposedOf.
- **ControlConstruct:** relocated from the CompositeProcess of the existing OWL-S Framework which can be used to describe a composition flow for one unique solution. Moving this construct from CompositeProcess to here allows us to describe a collection of possible solutions. In the "New York to Los Angeles" Process, for example, it may use a sequence list (one subclass of ControlConstruct) to contain all possible itineraries.

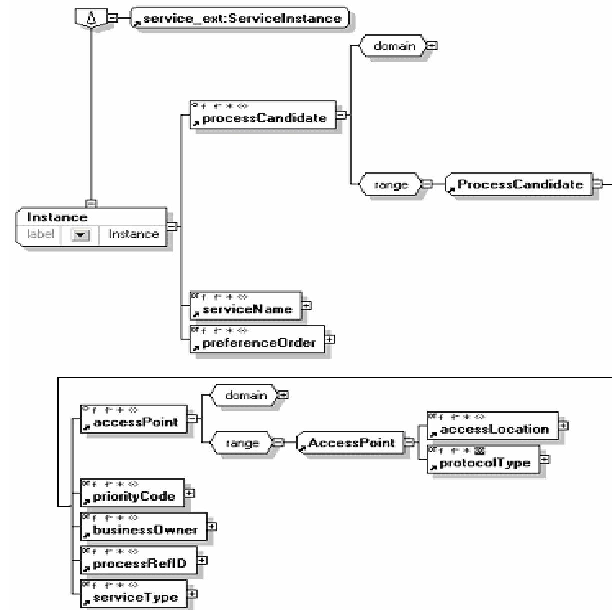

Figure 3 . Visual RDF Schema for "Instance" Class

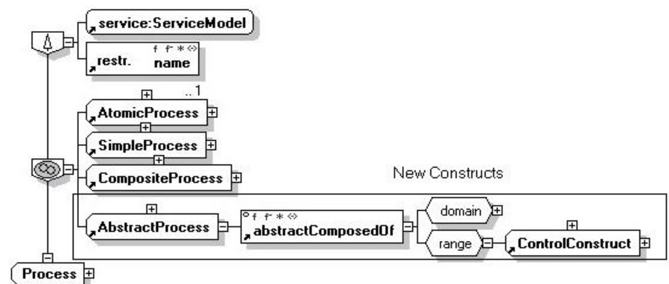

Figure 4 .Visual RDF Schema for "Process" Class

4. Planner Prototype

In order to facilitate the generation of different composite processes, we develop a web-based composer. It utilizes the J2EE web architecture and the Embedded JTP server. The Prototype contains a web interface to gather user's goals. Users can load their special ontologies while the default OWL-S ontology is automatically loaded in the planner. The customer's goals are converted into RDF entries and fed into the JTP inference engine. Consequently, the detailed inference steps are shown on the screen, and the resulting service files are generated when the user finishes the query and clicks the "Generate Result Service" button. Our prototype tool can be used for other service compositions, although we show flight service composition as an example in this paper.

One of the main differences between our extension and the original OWL-S is that we propose abstract service layers. Each specific abstract service needs to connect to its own profile, process and instance while maintaining the inheritance relationship with its parent class. To maintain the correct relationship for different parts of same service, each abstract service uses the "someValuesFrom" "allValuesFrom" and "hasValue" restrictions. Like the following example, AA-Flight_Service, the abstract flight service for American Airline, is connected to AA-Flight_Profile, instead of the Root level Profile Class.

```
<owl:Class rdf:about="#AA-Flight_Service">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:someValuesFrom>
        <owl:Class rdf:about="#AA-Flight_Profile"/>
      </owl:someValuesFrom>
      <owl:onProperty rdf:resource="http://www.daml
      .org/services/owl-s/1.1/Service.owl#presents"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.daml
      .org/services/owl-s/1.1/Service.owl#presents"/>
      <owl:allValuesFrom>
        <owl:Class rdf:about="#AA-Flight_Profile"/>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  ……..
```

Similarly, we use these restrictions to specify the IOPE for the profile and process of each abstract service. For instance, the profile of "AA-Flight_Service" can be defined with a "hasValue" restriction that sets the serviceParameter property to "aaServiceParameter", which means AA-Flight_Profile has a non-functional service parameter that is set to American Airline. Likewise, "JFK-ORD-Flight_Profile" may have a "hasInput" property with the "hasValue" restriction set to airport "JFK" and a "hasOutput" property with the "hasValue" restriction set to "ORD". Our planner prototype converts the user's goals into backend JTP query string and generates the Process based on the query result. The "and" and "or" operators are used on different levels to combine each JTP query string. Currently the prototype tool specifies three levels where level 1 is the top level. Below shows an example of the functional requirement of the user's goal to get an abstract service which has the departure airport "JFK" (input) and the arrival airport "ORD" (output). The non-functional requirements of the user's goal are that the flight service belongs to American Airlines and has either BusinessCabin or CoachOnlyCabin. The following is the generated JTP queries:

```
(and
  (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|hasInput|
    ?myFlight |http://127.0.0.1:8080/Parameters.owl#|::|JFK|)
  (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|hasOutput|
    ?myFlight |http://127.0.0.1:8080/Parameters.owl#|::|ORD|)
  (|http://www.daml.org/services/owl-s/1.1/Profile.owl#|::|servicePara
    meter| ?myFlight
    |http://127.0.0.1:8080/Parameters.owl#|::|aaServiceParameter|)
  (or
    (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|servicePa
      rameter| ?myFlight |http://127.0.0.1:8080/Parameters.owl#|
      ::|businessCabin|)
    (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|servicePa
      rameter| ?myFlight |http://127.0.0.1:8080/Parameters.owl#|
      ::|coachOnlyCabin|)
  )
)
```

Users can keep refining their requirements by adding/modifying their goals via our prototype tool. The query results are shown on the screen for review. When users are satisfied with the result, the files containing the final result OWL-S service are generated and added back to the ontology.

Currently, our planner prototype can only handle sequential compositions. When the JTP queries the result for the service components that satisfy the user's goals, the prototype will populate the Process abstractComposedOf property only with the "sequence" control construct. Suppose the user requests a flight service (JFK-LAX) from New York (JFK) to Los Angeles (LAX), for instance, the "JFK-ORD" and "ORD-LAX" services are the query result. We will add support for the other control constructs in the future.

5. Related Work

A number of languages/frameworks have been developed based on the standard W3C web service language to support web service composition. Among

them, two major efforts are the BPEL4WS [5] and OWL-S [2] which define a standard for concrete composite web services. Our work is an extension of OWL-S at the abstract service level.

There are several different approaches in web service composition area. Rao et al. [9] propose architecture for web service composition using the linear logic theorem proving. Both the service profile and customer goals are translated into the propositional linear logic and fed into the Jena planner [6]. The goal realization is based on the individual concrete web service. Similarly, Traverso et al. [11] use the EaGle language and the MBP Planner to generate composition result in π-calculus which is transformed to BPEL4WS. Mandell et al. [8] provide an automatic runtime discovery, composition and execution environment by integrating the BPEL4WS and OWL-S. These approaches focus on concrete web service composition; whereas our approach concentrates on the dynamic optimization and run-time fail over capability.

A transactional approach for web service composition is proposed in [1], where the accepted termination states are defined to allow the user to specify the required failure atomicity level. In contract, our approach focuses on fail-over and dynamic composition instead of failure atomicity.

WSMO (Web Service Modeling Ontology) [3] is another effort to address the semantic web services. WSMO relies on four core components: Ontology, Web Services, Goals and Mediators. There are several differences between the OWL-S and WSMO [7], e.g., separation of provider and requester point of view in WSMO and explicitly use of mediators to link the loose coupling core components. WSMO also describes similar concepts of OWL-S. For example, the OWL-S service profile can be expressed by the combination of the WSMO goal, the WSMO Web Service capability, and the Web Service non-functional properties [7]. Based on WSMO, implementation like WSMX [4] provides an execution environment for semantic web service, which enables the service registration, client discovery and invocation. The service compositions in WSMO, however, are still achieved at the concrete service level. In contrast, our approach focuses on abstract level for fail-over and dynamic optimization.

6. Conclusions

This paper has proposed an extension to the OWL-S ontology framework for dynamic web service composition at abstract service level. Rooted from OWL-S, our approach inherits the capability of semantic clarity. New abstract service concept is extended to the OWL-S. Each abstract service has an instance pool. Our planner prototype can take the user's goals and the service ontology and feed them into the backend inference engine to generate the results that are abstract services instead of concrete services. Each of the resulting abstract services has an instance pool of all possible concrete service solutions. Our planner prototype is based on the embedded JTP, which has been developed for the demonstration purpose.

In the future, we aim to continue enhancing the OWL-S ontology framework to support more complex optimization. We also plan to explore case studies related to both the abstract level and instance level semantic constraints, and the user's goals with "must have" and "good to have" semantic constraints. Furthermore, we intend to integrate our composer prototype with an existing ontology server.

References

[1] Sami Bhiri, Claude Godart, Olivier Perrin: Reliable Web services composition using a transactional approach, in Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service, Hong Kong, March 2005.

[2] M. Dean, D. Connolly, F. Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL Web Ontology Language 1.0 Reference. http://www.w3.org/TR/2002/WD-owl-ref-20020729/

[3] C. Feier, D. Roman, A. Polleres, J. Domingue, M. Stollberg and D. Fensel. Towards Intelligent Web Service Modeling Ontology (WSMO). In Proceedings of the International Conference on Intelligent Computing (ICIC) 2005, Hefei, China, August, 2005.

[4] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX - a semantic service-oriented architecture. In Proceedings of the International Conference on Web Services (ICWS 2005), Orlando, Florida (USA), July 2005.

[5] IBM. BPWS4J. http://www.alphaWorks.ibm.com/tech/bpws4j

[6] Jena - semantic web framework for java. Online: http://jena.sourceforge.net

[7] R. Lara, D. Roman, A. Pollers, and D. Fensel. A conceptual comparison of WSMO and OWL-S. European Conference on Web Services (ECOWS 2004), Erfurt, Germany, September, 2004, pages 254-269.

[8] D. Mandell and S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. Proceedings of the Second International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, 2003.

[9] J. Rao, P. Kungas, M. Matskin. Application of linear logic to web service composition. The First International Conference on Web Services, Las Vegas, USA, June 2003. CSREA Press.

[10] Stanford KSL. JTP. http://www.ksl.stanford.edu

[11] P. Traverso, M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. Technical report. # T04-06-08. Istituto Trentino di Cultura, 2004.