Denotation of Semantic Web Services Operations through OWL-S

Marco Luca Sbodio
Italy Innovation Center
Hewlett Packard Italiana
C.so Trapani 16, 10139 Torino, Italy
marco.sbodio@hp.com

Claude Moulin
University of Compiègne, CNRS, Heudiasyc
Centre de Recherches de Royallieu
60205 Compiègne, France
claude.moulin@utc.fr

Abstract

Emerging semantic web service description formalisms, such as OWL-S, allow for a definition of the semantic of services. Describing input and output types is not sufficient to declaratively and unambiguously denote the operations offered by a web service. Two services may have the same input and output types and have completely different semantics of their operation.

In this paper we present an approach for the specification of a web service denotation based on OWL-S capabilities, and an algorithm for dynamic discovery of services exploiting their denotation. We show how preconditions and results of the OWL-S formalism can be used to constrain the actual denotation of a service, and we describe how an agent can perform dynamic discovery of services exploiting their denotation. In our scenario, an agent has to search for the appropriate service, and verify that this service is able to produce the information that the agent needs.

1. Introduction

Web services constitute the building blocks of service oriented architectures. They offer modularity, flexibility and interoperability. Web services standards ensure the definitions of platform and language independent functional interfaces, and enforce the decoupling between interfaces and implementation. Although the WSDL description of a web service is a precise definition of its functional interface, it does not declaratively and unambiguously denote the semantics of the operations offered by the web service.

The *semantic web services* vision [1] is pursued by several emerging formalisms and frameworks, such as WSMO [20], SWSF [21] and OWL-S [15]. The Web Service Modeling Ontology (WSMO) defines an explicit conceptual model for Semantic Web Services [18]. It provides a framework for the description of Semantic Web Services that enables seamless business integration through formal descriptions [16, 17]. Although the aims of both WSMO and OWL-S are the same, they present some differences in their approach; a detailed comparison between OWL-S and WSMO is presented in [19].

Our work is based on OWL-S, and it explores how the denotation of a web service can be unambiguously specified using OWL-S Process ontology, through the definition of input and output types, and the declaration of preconditions and results.

The paper is structured as follows. In Section 2 we give an overview of the approach. In Section 3 we introduce a simple reference scenario, which will be used throughout the paper to illustrate our approach. Section 4 gives an overview of the OWL-S features that allow for a complete denotation of a service; in Section 5 we show how OWL-S features are used to describe the services in our reference scenario, and we explain how we use them to achieve a full denotation of the services. In Section 6 we present an algorithm that shows how an agent can perform dynamic discovery of services exploiting their denotation. We conclude with a comparison on related works in Section 7.

2. Overview

We aim at automating the dynamic discovery of web services performed by an agent seeking to satisfy some goal. In the discovery process it is necessary to use the full semantic denotation of the web service operations (input/output types, preconditions and results) in order to assess if a service is appropriate to fulfill the agent's goal. Specifically, we use preconditions and results to declare constraints among inputs and output of a web service in order to disambiguate its operations.

Given a set of web services, we assume that their OWL-S descriptions are available through a *semantic* *service registry.* We do not bind to any specific implementation of semantic service registry, but we simply assume that the semantic service registry works as an RDF store that can be queried using SPARQL [25] or RDQL [26].

The discovery process is carried out by an agent, which tries to fulfill a goal represented by an RDQL query. The agent has its own knowledge base (an RDF/OWL model), which is used in the following ways:

- it contains instances of some ontology classes, which can be used as inputs for web services
- it is augmented during the discovery process with knowledge inferred from the OWL-S descriptions
- it is queried to check if the goal is fulfilled

Our current focus is on *information production* web services, which usually generates or returns some kind of information based on information given as input and (possibly) the world state. This kind of web services usually do not produce changes in the state of the world (effects), which is a peculiarity of the *world transition* web services. Information production web services are very common in the e-Government domain, which is the domain of the TERREGOV project: Impact of e-Government on Territorial Government Service. TERREGOV addresses the issue of interoperability of e-Government services for local and regional governments (see Section 8). We show how preconditions and results may be exploited to disambiguate the denotation of the operations of information production web services.

3. Reference Scenario

We illustrate our approach through the following reference scenario, which is a simplification of actual processes occurring in e-Government applications. We refer to a simple domain ontology (see figure 1)[1], which defines the class `Person` with three object properties (`hasPassport`, `hasSocialSecurityCard`, `hasMother`).

Properties `hasPassport` and `hasSocialSecurityCard` represent the link between a `Person` and respectively `Passport` and `SocialSecurityCard`, which represent two identifiers for the same person (it is often the case in public administration processes that people have different identifiers according to service's domain). Property `hasMother` represents the parental relationship between two persons.

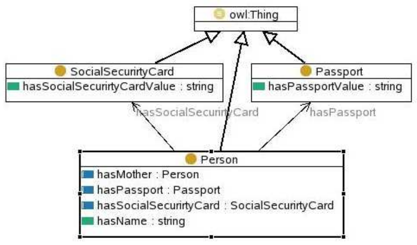

Figure 1. Ontology schema.

A `Person` has also a datatype property called `hasName`. This domain ontology is used to define the inputs and outputs types of three web services, which are informally described here:

- WS1: given the instance of `Passport` of a `Person`, WS1 returns the instance of `SocialSecurityCard` of the same `Person`.
- WS2: given the instance of `SocialSecurityCard` of a `Person`, WS2 returns the value of `hasName` of the same `Person`.
- WS3: given the instance of `Passport` of a `Person`, WS3 returns the instance of `SocialSecurityCard` of the `Person`'s mother.

WS1 and WS3 have the same input (an instance of Passport) and the same output (an instance of SocialSecurityCard). However, the purpose of WS1 is completely different from the purpose of WS3. Furthermore, both WS1 and WS3 can be composed with WS2 (output type of both WS1 and WS3 matches with the input type of WS2), but the semantics of the composition is quite different.

4. Specification of Service Denotation with OWL-S

The formal denotation of a service is expressed by the declarative specification of all its characteristics: input/output types, preconditions and results (and the possible relationships among them). OWL-S ontologies provide appropriate constructs to formally specify all these elements. In this work we refer to the version 1.2 of OWL-S [15], which is not yet finalized. Specifically, we use the information provided through the OWL-S Process ontology, which in OWL-S version 1.2 has been substantially revised and enriched.

[1]The figure has been built with the TopBraid Composer (http://www.topbraid.com)

OWL-S Process ontology have specialized constructs that allow the declarative definition of inputs and output types (through the object properties `hasInput` and `hasOutput`) and of preconditions and results (through the object properties `hasPrecondition` and `hasResult`).

Specifically, `hasPrecondition` defines the conditions that must hold true before the service can be invoked, and `hasResult` defines the results that are produced by the service. An OWL-S Process may have several results with corresponding outputs. Each result can be associated to *result conditions* that specify when that specific result can occur. It is assumed that such conditions are mutually exclusive, so that only one result applies in any single situation.

The results conditions are specified through the object property `inCondition`. Furthermore, the results may also specify some effects, i.e. changes in the state of the world. Inputs, Output, Preconditions and Results of a service are often referred to with the acronym **IOPR**.

In the OWL-S Process ontology the range of `hasPrecondition` and `inCondition` is `Condition`, which is a sub-class of `Expression`; an `Expression` is defined in some logical language (SWRL [22], DRS [23], KIF [24], SPARQL [25], RDQL [26]). Notice that OWL-S has introduced the use of RDQL and SPARQL only in late release (see OWL-S 1.2 Pre-Release [15]). This variety of logical languages offers a high degree of flexibility, even if this may pose some issues in terms of interoperability. Notice that we currently use RDQL for OWL-S conditions and query over knowledge bases, but everything could be easily expressed with the emerging SPARQL standard.

The OWL-S Process ontology says that `hasPrecondition` and `inCondition` properties refer to conditions that are tested in specific contexts. Preconditions are evaluated with respect to the client environment before the process is invoked; result conditions are effectively meant to be evaluated in the server context after the process has executed, which is impossible in case of service discovery.

Our interpretation of result conditions is that they are also part of the client context, in the sense that the client obtains the result *iff* the corresponding condition is true (this interpretation is further clarified in section 5).

Finally, OWL-S provides also specialized constructs (`VariableBinding`) to define a correspondence between a variable mentioned in a logical expression (used in `hasPrecondition` and `inCondition`) and OWL instances, or instances of OWL-S `process:Parameter`.

5. Service Denotation with IOPR

We show here how to denote web services using OWL-S Inputs, Outputs, Preconditions and Results, and we describe our interpretation of results' conditions during the service discovery phase. In our description we refer to the web services WS1 and WS3 introduced in our reference scenario (see Section 3).

The following is an excerpt of OWL-S definition of WS1 (we use the more compact N3 [2] format); `ns1` is the namespace of the OWL-S definition of WS1, `process` is the namespace of the OWL-S Process ontology, and `ex` is the namespace of the domain ontology illustrated in section 3).

```
ns1:WS1
   a   process:AtomicProcess ;
   process:hasInput ns1:PassportIn ;
   process:hasOutput ns1:SocialSecurityCardOut ;
   process:hasPrecondition ns1:WS1Precondition ;
   process:hasResult ns1:WS1Result .

ns1:PassportIn
   a   process:Input ;
   process:parameterType "ex:Passport"^^xsd:anyURI .

ns1:SocialSecurityCardOut
   a   process:Output ;
   process:parameterType
      "ex:SocialSecurityCard"^^xsd:anyURI .

ns1:WS1Precondition
   a   expr:RDQL-Condition ;
   expr:expressionData
      "(?p ex:hasPassport ?pass)"^^xsd:string ;
   expr:variableBinding ns1:VariableBinding_p ,
      ns1:VariableBinding_pass .

ns1:WS1Result
   a   process:Result ;
   process:inCondition ns1:WS1ResultCondition .

ns1:WS1ResultCondition
   a   expr:RDQL-Condition ;
   expr:expressionData
      "(?p ex:hasSocialSecurityCard ?ssc)"^^xsd:string ;
   expr:variableBinding
   ns1:VariableBinding_p , ns1:VariableBinding_ssc .

ns1:P
   a   process:Existential ;
   process:parameterType "ex:Person"^^xsd:anyURI .

ns1:VariableBinding_p
   a   expr:VariableBinding ;
   expr:theObject ns1:P ;
   expr:theVariable "?p"^^xsd:string .

ns1:VariableBinding_pass
   a   expr:VariableBinding ;
   expr:theObject ns1:PassportIn ;
   expr:theVariable "?pass"^^xsd:string .

ns1:VariableBinding_ssc
```

```
    a   expr:VariableBinding ;
    expr:theObject ns1:SocialSecurityCardOut ;
    expr:theVariable "?ssc"^^xsd:string .
```

The definition starts with the declaration of instances of `process:Input` and `process:Output` (respectively `ns1:PassportIn` and `ns1:SocialSecurityCardOut`).

Following is the declaration of the precondition `ns1:WS1Precondition`, which asserts the RDQL clause `(?p ex:hasPassport ?pass)`: the variable `?p` is bound to the instance `ns1:P`, and the variable `?pass` is bound to the instance `ns1:PassportIn` (i.e. the `process:Input`). Notice that `ns1:P` is an instance of `process:Existential`, a special OWL-S construct for declaring variables with process scope, so that when they are bound in preconditions, they can be referenced also in results.

Finally there is the declaration of a `process:Result` with the associate condition `ns1:WS1ResultCondition`, which asserts the RDQL clause `(?p ex:hasSocialSecurityCard ?ssc)`: the variable `?ssc` is bound to the instance `ns1:SocialSecurityCardOut` (i.e. the `process:Output`), and the variable `?p` is bound as described above. The information provided through the `process:Existential` instance (`ns1:P`) and the bound logical expressions contribute to the denotation of the service: WS1 returns the social security card of a person whose passport is given as input. Notice that an agent can infer that the instance (`ns1:P`) is referring to (`ex:Person`) through the value of the `process:parameterType` property.

`ns1:WS1ResultCondition` represents a result conditions: our interpretation of result conditions is that they are evaluated in the server context, but can also be used in the agent context during the discovery process, in the sense that the agent receives the result *iff* the corresponding result condition is true.. We interpret `ns1:WS1ResultCondition` in the following way:

- if the condition `ns1:WS1ResultCondition` is true, then the service's result is `ns1:WS1Result` (server context)
- if the agent receives the `ns1:WS1Result`, then the condition `ns1:WS1ResultCondition` is true (agent context)

The agent may be unable to verify the condition `ns1:WS1ResultCondition`, which possibly requires some knowledge available only in the server context. Nevertheless, the agent may reason over the service OWL-S description during service discovery, while trying to check if any of the possible service's result help in achieving its goal. This reasoning is based on the following steps:

- The agent assumes that it receives from the service a specific result (in our example `ns1:WS1Result`).
- Under this assumption, the corresponding condition is assumed to be true (in our example `ns1:WS1ResultCondition`).
- Using each `expr:VariableBinding` specified in the condition, the agent can transform the corresponding RDQL clause into an RDF statement, whose subject (or object) is an instance of the OWL class specified by the `process:parameterType` property of the `process:processVar` bound to the variable. In our example the new statement would be `(:P_X ex:hasSocialSecurityCard :SSC_X)`, where the subject is an instance of `ex:Person`, and the object is an instance of class `ex:SocialSecurityCard`.
- The new statement becomes a fact added to the agent knowledge base, and the agent can now check if this additional knowledge allows the fulfillment of its goal.

The OWL-S Process description of the other web services in our reference scenario is similar to the one of WS1. For comparison we provide only a small fragment of the OWL-S process description of WS3 (`ns3` is the namespace of the definition of WS1, `process` is the namespace of the OWL-S Process ontology, and `ex` is the namespace of the domain ontology illustrated in section 3):

```
ns3:WS3
   a   process:AtomicProcess ;
   process:hasInput ns3:PassportIn ;
   process:hasOutput ns3:SocialSecurityCardOut ;
   process:hasPrecondition ns3:WS3Precondition ;
   process:hasResult ns3:WS3Result .

...

ns3:WS3Precondition
   a   expr:RDQL-Condition ;
   expr:expressionData
      "(?p ex:hasPassport ?pass)"^^xsd:string ;
   expr:variableBinding ns3:VariableBinding_p ,
      ns3:VariableBinding_pass .

ns3:WS3ResultCondition
   a   expr:RDQL-Condition ;
   expr:expressionData
      "(?p ex:hasMother ?m),
      (?m ex:hasSocialSecurityCard ?ssc)"^^xsd:string ;
   expr:variableBinding ns3:VariableBinding_p ,
      ns3:VariableBinding_m , ns3:VariableBinding_ssc .
```

```
ns3:VariableBinding_m
   a   expr:VariableBinding ;
   expr:theObject ns3:M ;
   expr:theVariable "?m"^^xsd:string .

ns3:M
   a   process:ResultVar ;
   process:parameterType "ex:Person"^^xsd:anyURI .
```

The precondition `ns3:WS3Precondition` is the same as `ns1:WS1Precondition`, but the result condition `ns3:WS3ResultCondition` is more complex than `ns1:WS1ResultCondition`. The variable `?p` is bound to an instance of `process:Existential` (whose `process:parameterType` refers to `ex:Person`), and the variable `?ssc` is bound to an instance of `process:Output` (whose `process:parameterType` refers to `ex:SocialSecurityCard`).

`ns3:WS3ResultCondition` uses also an additional variable `?m`, which is bound to an instance of `process:ResultVar`, and which is used to tie together the variable `?p` and the variable `?ssc`. The information provided through `ns3:WS3Precondition` and `ns3:WS3ResultCondition` contribute to the denotation of the service: WS3 returns the social security card of the mother of a person whose passport is given as input.

6. Automated Discovery of Services based on IOPR

We present the high level steps of an algorithm that an agent may use to select a service based on its goal and the service IOPR definition expressed in the OWL-S description of the service. The basic idea underlying the algorithm is the use of preconditions and result conditions to create a connection between OWL-S bindings and the knowledge base (an RDF/OWL model) maintained by the agent.

The agent dynamically checks the OWL-S descriptions of the services available in the semantic service registry; it checks if it can provide appropriate inputs, and satisfy the service's preconditions. For those services that can be invoked, the agent performs case reasoning on the service's results: it assumes that it receives a specific result, and therefore that the associate conditions are true; using the conditions and the variable bindings the agent infer additional knowledge as described in Section 5.

The agent adds the additional knowledge to its knowledge base, and checks if its goal is now satisfiable (that is the RDQL query representing the goal returns some answer when executed over the agent's knowledge base). If the goal is satisfiable, then the specific service is potentially useful to fulfill the agent's goal.

The following pseudo-code represents a more formal definition of the algorithm:

```
KBA : agent's knowledge base

GOAL = RDQL query expressing the required
       output type and potential
       constraints

CANDIDATES : list of <X,Y> where
             X is an OWL-S Process and Y
             is a Result of X that
             potentially fulfills GOAL

GOT = output type expressed by GOAL

C = findProcessWithCompatibleOutputType(GOT)

foreach (Process P in C) {

  P_PRECONDS = Preconditions of P
               and corresponding Variable
               bindings
  P_INPUTS = Input types of P

  if ( canInvoke(P_PRECOND, P_INPUTS) ) {

    P_RESULTS = Results of Process P

    foreach (Result R in P_RESULTS) {

      R_COND = Result Conditions of R
               and corresponding Variable
               bindings
      KBA_1 = augmentKB ( R_COND )

      S = checkGoal (KBA_1)

      if ( S is not empty) {
        add <P,R> to CANDIDATES
      }
    }
  }
}
```

In the following we discuss the algorithm providing some examples based on the reference scenario introduced in Section 3. `KBA` is the agent's knowledge base; to illustrate the algorithm we assume that `KBA` is the following:

```
:Marco
  a ex:Person ;
  ex:hasPassport :MarcosPassport .

:MarcosPassport
  a ex:Passport ;
  ex:hasPassportValue "A33Y55"^^xsd:string .
```

The agent knows an instance of `ex:Person` and his `ex:Passpart`.

`GOAL` is the agent's goal. We define the goal with an RDQL query that expresses the required output type and potential constraints; to illustrate the algorithm we assume that the agent has the following GOAL:

```
SELECT ?y
WHERE
(:Marco ex:hasSocialSecurityCard ?y)
(?y rdf:type ex:SocialSecurityCard)
USING
ex FOR <http://www.example.com/domainOnt#>
```

The goal of the agent is to find information about the ex:SocialSecurityCard of the instance of ex:Person contained in its knowledge base.

The first step of the algorithm invokes the procedure findProcessWithCompatibleOutputType. This procedure queries the semantic service registry, and finds all the services whose OWL-S Process description declares an output type compatible with the one declared in GOAL.

We say *compatible* output type, because we do not restrict to exact output type match, but we also accept weaker matches obtained through subsumption between the GOAL output type and the output type declared in the OWL-S. A detailed discussion of such matching between types is provided in [4]. The procedure findProcessWithCompatibleOutputType returns a set (C) of matching OWL-S Processes. The algorithm then examines each Process in C to asses both that the agent can invoke the Process, and that the Process is adequate to fulfill GOAL. In our reference scenario we have C = {WS1, WS3}.

For each Process P in C the algorithm assigns to P_PRECONDS its preconditions and to P_INPUTS its inputs. The algorithm then invokes the procedure canInvoke, which checks if the agent can provide the required inputs and satisfy the preconditions to invoke the Process P. The agent takes each preconditions and corresponding variable bindings and use it to build clauses of an RDQL query; this query is executed over KBA, and if it returns results (i.e. all unbound variables are bound by the query execution), then the agent has enough knowledge to invoke the process P. Assuming that P is WS1 from our reference scenario the RDQL query built from its precondition is the following:

```
SELECT ?p ?pass
WHERE
(?p ex:hasPassport ?pass)
(?pass rdf:type ex:Passport)
(?p rdf:type ex:Person)
USING
ex FOR <http://www.example.com/domainOnt#>
```

The three clauses of the above RDQL query are obtained as following: the first is the logical expression asserted in the precondition ns1:WS1Precondition; the second and third clauses are inferred from the properties process:parameterType of the instances of process:processVar bound to the variables in the precondition. The execution of the query over KBA binds ?p to :Marco and ?pass to :MarcosPassport: the agent can therefore invoke WS1.

In the next steps the agent performs case reasoning over the possible results of Process P, to check if any of them yields to the fulfillment of GOAL. The agent assumes that it receives a result R, and therfore that the corresponding conditions R_COND are true. Assuming that P is WS1 from our reference scenario, then R_COND is (?p ex:hasSocialSecurityCard ?ssc). The procedure augmentKB uses the result's conditions and their variable bindings to infer additional knowledge as described in Section 5. Notice that if a variable appear both in preconditions and result's condition with the same expr:VariableBinding, then the agent can reuse the binding obtained when checking preconditions over KBA. In our example the agent can infer the following statements:

```
:Marco ex:hasSocialSecurityCard :SSC_X .
:SSC_X a ex:SocialSecurityCard .
```

The variable ?p is bound to :Marco because it retain the same binding obtained while checking preconditions; the variable ?ssc is bound to a dynamically generated instance of ex:SocialSecurityCard. The above two statements are added to KBA; the augmented knowledge base KBA_1 represents what the agent knows when it receives the result R. The last step of the algorithm consists in checkering if in this case (P is invoked and it returns result R) the agent's goal is fulfilled.

The procedure checkGoal executes the RDQL query GOAL over KBA_1; if the result set S is not empty, then GOAL can be fulfilled. In our example it is straightforward to see that the agent can use WS1 to fulfill its goal, but not WS3: although KBA contains appropriate knowledge to invoke WS3, the Process's result do not bring the information required to fulfill GOAL. For comparison we show below the statements inferred from OWL-S description of WS3, and specifically from its result condition ns3:WS3ResultCondition:

```
:Marco ex:hasMother :P_M .
:P_M ex:hasSocialSecurityCard :SSC_X .
:SSC_X a ex:SocialSecurityCard .
:P_M a ex:Person .
```

The knowledge base KBA_1 (obtained from KBA adding the above statements) do not allow answering the GOAL.

Notice that after executing the above algorithm the list CANDIDATES contains pairs of <X,Y>, where each X is an identifier of an OWL-S Process that the agent can potentially invoke to fulfill its goal, and Y is the corresponding result. Nevertheless it must be checked at run-time that the actual Result obtained by invocation of X corresponds to Y. If actual invocation of X produces at run-time a result Z different from Y, then the goal is not fulfilled.

7. Related Works and Conclusions

The selection of web service based on OWL-S (previously DAML-S) description has also been explored by M. Paolucci et al. in [4]. This work explores an approach to evaluate similarity among service advertisements (OWL-S descriptions) and services requests based on the semantic match among inputs/outputs types of a service advertisement and the inputs/outputs types of a service request.

According to the algorithm presented in [4], an advertisement matches a request when all the outputs of the request are matched by the outputs of the advertisement, and all the inputs of the advertisement are matched by the inputs of the request.

Our approach is built on the same idea, but expands it with the use of preconditions and result condition, which allows for a more precise denotation of the service, and allows an agent to discriminate among services having the same inputs/outputs types (possibly matching the ones in its goal), but with (very) different semantics associated to their operations.

Our approach is based on the interpretation of results' condition in the agent context, and the use of preconditions and results' conditions to create a connection between OWL-S bindings and the knowledge base maintained by the agent; this connection allows the agent to augment its knowledge base, and perform case reasoning over the possible results.

The automation of services selection based on their semantic is often the first step while performing services composition: [3] describes an approach for building a system used for an interactive composition of web services. Another interesting approach to web services composition has been presented by [5]. This work focuses on the use of Hierarchical Task Networks (HTN) and planning techniques for service composition. It is based on previous works presented in [6] and [7]. It also present the notion of *query*, which is close to our approach of expressing the agent goal with an RDQL query, and the notion of *information sources*, which are a kind of abstraction for information production web services (that are the ones we currently work with).

Among the various works on automated web service compositions ([8, 9, 10]) an interesting one is discussed in [10], where the authors present the design and implementation concepts of Plængine, a software system that support composition and enactment of services (a research worked supported by the Adaptive Service Grid [11]). Among the challenges related to service's composition, the authors of [10] report the fact that it is not sufficient to specify the elements of composition by their names and inputs/outputs, but it's also necessary to specify the functionalities through semantic annotations, which can be accomplished through preconditions and effects. This is the same idea underpinning our approach.

In this paper we have shown how OWL-S provides sufficient expressiveness to specify a declarative and unambiguous denotation of a service. We have described how result conditions can be interred also in the agent context while performing case reasoning over the possible service results. Finally we have presented an algorithm that the agent can use to dynamically select services that can be used to fulfill its goal.

8. Acknowledgments

This work has been developed within the TERREGOV project, an integrated project cofunded by the European Commission[2] under the IST (Information Society Technologies) Program, e-Government unit, under the reference IST-2002-507749.

References

[1] S. McIlraith, T.Son and H. Zeng *Semantic Web Services.* IEEE Intelligent Systems, Special Issue on the Semantic Web. 16(2):46–53, March/April, 2001.

[2] T. Berners-Lee. A readable language for data on the Web. N3 formalism http://www.w3.org/DesignIssues/Notation3.html

[3] E. Sirin, B. Parsia, and J. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[4] M. Paolucci et al. Semantic Matching of Web Services Capabilities *The Semantic Web-ISWC 2003: 1st International Semantic Web Conference*, LNCS 2342, Springer-Verlag, 2003.

[5] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler Information gathering during planning for web service composition *Proceedings of the Third Internatonal Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, November 2004.

[6] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau Automating DAML-S web services composition using SHOP2 *Proceedings of 2nd International Semantic Web*

[2]The content of this paper is the sole responsibility of the authors and in no way represents the views of the European Commission or its services.

Conference (ISWC2003), Sanibel Island, Florida, October 2003.

[7] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau HTN planning for web service composition using SHOP2 *Journal of Web Semantics*, 1(4):377-396, 2004.

[8] M. Pistore and P. Bertoli and F. Barbon and D. Shaparau and P. Traverso Planning and Monitoring Web Service Composition *Proceedings of ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[9] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions *Proc. of Web Services: Modeling, Architecture and Infrastructure*, Workshop in Conjunction with ICEIS2003, Angers, France, 2003.

[10] H. Overdick, H. Meyer, and M. Weske. Plaengine: A System for Automated Service Composition and Process Enactment *Proc. of WWW Service Composition with Semantic Web Services*, Compiegne, France, September 19, 2005.

[11] Adaptive Services Grid (ASG) An Integrated Project supported by the Sixth Framework Programme of the European Commission http://asg-platform.org/

[12] K. Sycara, S. Widoff, M. Klusch, J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace *Autonomous Agents and Multi-Agent Systems*, 5, pp. 173-203, 2002.

[13] D. Trastour, C. Bartolini, and J. Gonzalez-Castillo. A semantic web approach to service description for matchmaking of services *Proc. of the Intl. Semantic Web Working Symposium (SWWS)*, Stanford, CA, USA, 2001.

[14] J. Domingue, L. Cabral, and F. Hakimpour. IRS-III: a Platform and Infrastructure for Creating WSMO-based Semantic Web Services *WIW workshop on WSMO implementation*, Frankfurt, September 29-30th, 2004.

[15] OWL-S 1.2 Pre-Release
http://www.ai.sri.com/daml/services/owl-s/1.2/

[16] J. Domingue, D. Fensel, and D. Roman. Semantic Web Services with the Web Services Modeling Ontology (WSMO), *AgentLink News 19*, 2005.

[17] J. de Bruijn, D. Fensel, U. Keller, and R. Lara. Using the Web Service Modeling Ontology To Enable Semantic e-Business, *Communications of the ACM 48(12)*, 2005.

[18] Christoph Bussler, Dieter Fensel, Dumitru Roman, et al. Web service modeling ontology. *Applied Ontology Journal*, 1(1), 2005.

[19] R. Lara, A. Polleres, H. Lausen, D. Roman, J. de Bruijn, and D. Fensel. A conceptual comparison between WSMO and OWL-S, 2005. www.wsmo.org/TR/d4/d4.1/v0.1/.

[20] Web service modeling ontology
http://www.wsmo.org/

[21] Semantic Web Services Framework (SWSF)
http://www.w3.org/Submission/SWSF/

[22] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining owl and ruleml, 2003. http://www.daml.org/2003/11/swrl/.

[23] Drew McDermott DRS: A Set of Conventions for Representing Logical Languages in RDF. January 12, 2004. http://www.cs.yale.edu/homes/dvm/daml/DRSguide.pdf

[24] KIF. Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at http://logic.stanford.edu/kif/dpans.html.

[25] Eric Prud'hommeaux, Andy Seaborne. SPARQL Query Language for RDF. W3C Working Draft 4 October 2006. http://www.w3.org/TR/rdf-sparql-query/

[26] Andy Seaborne. RDQL - A Query Language for RDF. W3C Member Submission 9 January 2004. http://www.w3.org/Submission/RDQL/