AFRL-IF-RS-TR-2006-274
Final Technical Report
August 2006

SEMANTIC WEB SERVICES WITH WEB ONTOLOGY LANGUAGE (OWL-S) - SPECIFICATION OF AGENT-SERVICES FOR DARPA AGENT MARKUP LANGUAGE (DAML)

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K530/00

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2006-274 has been reviewed and is approved for publication.

APPROVED: /s/

ALBERT G. FRANTZ
Project Engineer

FOR THE DIRECTOR: /s/

JAMES W. CUSACK
Chief, Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* Aug 06	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jun 00 – Apr 06

4. TITLE AND SUBTITLE SEMANTIC WEB SERVICES WITH WEB ONTOLOGY LANGUAGE (OWL-S) - SPECIFICATION OF AGENT-SERVICES FOR DARPA AGENT MARKUP LANGUAGE (DAML)	**5a. CONTRACT NUMBER** F30602-00-0592
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62301E
6. AUTHOR(S) Katia P. Sycara	**5d. PROJECT NUMBER** DAML
	5e. TASK NUMBER 00
	5f. WORK UNIT NUMBER 14
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University, Box 2950, 700 Technology Drive, Pittsburgh, Pennsylvania 15213	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington Virginia 22203-1714; AFRL/IFSA, 525 Brooks Road, Rome New York 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-274

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA #06-584

13. SUPPLEMENTARY NOTES

14. ABSTRACT
CMU did research and development on semantic web services using OWL-S, the semantic web service language under the Defense Advanced Research Projects Agency- DARPA Agent Markup Language (DARPA-DAML) program. The objective was to develop a markup language to allow intelligent agents to communicate. The DAML program resulted in the Web Ontology Language (OWL) Markup as a W3 recommendation. OWL-S is a markup language based on OWL to create computer discoverable web services. The report includes documentation of research on the OWL-S profile, process model and grounding. The research also includes intelligent web service composition, brokering and an OWL-S virtual machine.

15. SUBJECT TERMS
Semantic Web, Semantic Web Services, Web Ontology Language, OWL-S, DAML, UDDI Registry, OWL-S Broker, Web Service Discovery, Composition and Mediation, OWL-S Virtual machine

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Albert G. Frantz
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	144	**19b. TELEPONE NUMBER** (*Include area code)*

Table of Contents

List of Figures

1. Introduction

The CMU work on DAML covered four areas:

- Involvement with standards for Web Services and Semantic Web Services
- Contributing to the development of DAML-S/OWL-S, the language for Semantic Web Services
- Development of tools for deployment of OWL-S based Web Services.
- Development of algorithms for Semantic Web Service Discovery, Matchmaking, Brokering, Invocation and Composition

2. Involvement with standards for Web Services and Semantic Web Services

W3C participation: The PI, Katia Sycara, served as an Invited Expert at the W3C Working Group on Web Services Architecture from 2002-2004. The Web Services Architecture specification document that resulted from the Working Group was accepted as a Note by the W3C in February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

Lessons Learned. The large majority of industrial participants at the W3C Working Group on Web Services Architecture were very skeptical about adding semantic annotations to Web Service descriptions and allowing automated semantic inference. The reasons were that, although they recognized that semantics would be useful, they were afraid of computational complexity of logic inference mechanisms and the lack of robust tools for performing such inference. Despite this skepticism, a small number of participants, including Dr. Sycara, managed to successfully propose and include in the specification document of the Web Services Architecture the need for semantics, need to automated discovery and invocation of services. In addition, CMU expressed to the group the need for expressing the Semantic Web Services Architecture specification in RDF and proceeded to do so. This became part of the submission of the Note http://www.w3.org/Submission/2004/07/. After OWL became a W3C standard in 2004, it seems that industry is more willing to consider the Semantic Web and semantic annotations on Web Services as a real viable and value-added proposition.

OASIS participation: The PI, Dr. Sycara also took part in the OASIS standards organization as a member of the UDDI Technical Committee. She participated in the specification of UDDI V3 release of the UDDI standard. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3

Lessons learned: The industrial participants of the Technical Committee were skeptical about inclusion of semantic annotations to the description of web services. In addition, they were skeptical on allowing automated and dynamic matching of service capabilities in UDDI. Despite this skepticism, a small number of members, including Dr. Sycara, proposed and wrote specifications for the enrichment of the capability descriptions of services in UDDI in terms of OWL and also specified a limited semantic search functionality. This will be part of the next UDDI release.

Semantic Web Services Initiative (SWSI): The PI, Dr. Sycara served as the US co-chair of the US-EU initiative on Semantic Web Services, which got initiated in 2003 by the Darpa PM. The initiative had two technical committees, Semantic Web Services Language (SWSL) and Semantic Web Services Architecture (SWSA). The Semantic Web Services Language technical committee submitted a specification for a language to W3C. http://www.w3.org/Submission/2005/SUBM-SWSF-20050909/

The SWSA committee's work also resulted in a document specification but it was not submitted to any standards body.

3. Development of OWL-S Language for Semantic Web Services

The PI, Katia Sycara and other members of the CMU DAML research group participated in the development of the OWL-S specification. The specification was accepted as a W3C Note.

http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

The Semantic Web Service vision is to describe the capabilities and content of Web services in an unambiguous, computer interpretable language, thus enabling automation of many of the Web service tasks currently performed manually by human beings. These include not only improved automation of Web service discovery and invocation, but also automated composition, interoperation, execution monitoring and recovery. To support this vision, Semantic Web Services provide more powerful Web Service development tools that enable, among other things, automated simulation and verification of Web service properties, and consistency checking and debugging features. Semantic Web services improve the quality and robustness of existing tasks such as Web Service discovery, while enabling a broad range of new automation tasks, heretofore performed by humans. A key component of the Semantic Web Services vision is the creation of a language for describing Web Services. OWL-S is such a language. Below, we give a brief conceptual overview of OWL-S. For more information, see the W3C submission.

3.1. Design Principles of OWL-S

The objective of OWL-S is to provide a Web service description language that supports *Automatic interaction and composition of Web services*. To support automatic interaction of Web services, OWL-S provides a way to extract the semantics of the messages that the agent exchanges with the Web service and to evaluate the consequences of those messages. To support automatic composition of Web services, OWL-S supports capability-based discovery and composition by providing a language that specifies what a Web service does and how it achieves its results.

Automatic interaction and composition of Web services greatly facilitates the applicability of Web services in two ways. First, the use of explicit semantics abstracts the specific syntax of messages exchanged by the agent and the Web service. Therefore, any operation on the message structure that would require human intervention can be handled automatically by OWL inference engines. As a consequence, Web services that use OWL-S interoperate more smoothly and are more resilient to changes requiring less maintenance of Web services that are based only of XML Schemata. Second, and more importantly, the use of OWL ontologies supports reasoning about the complete process of information exchange between the Web services and agent that allows agents to decide which is the best Web service that serves their needs, to derive how to interact with that Web service, and to actually interpret the information that the client and the Web service exchange. As a consequence OWL-S supports problem-driven automatic composition of Web services.

In a nutshell, the *design principles* behind the OWL-S ontologies are the following:

1. *Web services should be represented by their capabilities and the results that they produce*. In turn this representation allows means-end analysis and reasoning on what is the best Web service to achieve a given goal, and how to use that Web service achieve that goal.
2. *OWL-S is a language to describe Web services*, OWL-S should therefore be independent of any architectural design or implementation. The only architectural commitments that OWL-S makes are the commitments made by layering over Web service oriented languages such as WSDL.
3. *OWL-S builds on existing Web services standards*: rather than defining a new set of languages to describe Web services, OWL-S aims at enriching current Web services standards with semantic information. Therefore there should be a mapping between OWL-S specifications and Web services standards specifications such as WSDL and UDDI.
4. *OWL-S builds on existing Semantic Web standards*, rather than looking for the perfect logics for Web services, OWL-S aims at making as much use as possible of OWL. The representational problems of OWL-S can in turn motivate additional development of Semantic Web standards.

In the following, we will discuss these principles in more detail.

Support for Automatic Interaction with Web services

OWL-S is designed to enable the automation of a wide variety of activities related to services. A primary motivation is to *enable software agents to use services* without any ‘built-in’ pre-existing knowledge of their properties, capabilities, and protocols. OWL-S adopts the prospective that to support automatic interaction and composition of Web services, it needs to support capability based discovery, and goal driven interaction. The problem of capability-based discovery is solved by matching the specification of a problem, with the specification of the set of solutions that are provided by a Web service. The result of the discovery process is the set of Web services that can solve a given problem. The discovery process leads to finding the most appropriate Web services, but it does not provide the solution to the problem. Rather, the solution is found through the interaction with the Web service. Such an interaction may involve multiple exchanges of information, each resulting in the invocation of a different operation, in such a way that the combination of the operations results in the complete solution of the original problem.

The same distinction between discovery and interaction is present in the OWL-S ontology for Web services. Specifically, OWL-S distinguishes between the Profile ontology that is used to represent the capabilities of Web services, the OWL-S Process Model and Grounding that are used to manage the interaction between Web services.

The OWL-S Profile provides a representation of the capabilities of Web services and the requirements of applications that need to interact with the Web service. The capabilities of Web services are expressed in terms of the functional transformation produced by the Web service: specifically the web service produces an information transformation from the inputs that it expects to the outputs it generates, but also it produces a state transformation from a set of conditions that need to be true for the Web service to run correctly, to a set of effects that result from the invocation of the Web service. In addition to the functional specification, the OWL-S Service Profile also describes a set of qualitative properties such as quality of service provided, security guarantees etc.

The second use of the OWL-S Profile is to describe the requirements of an agent that looks for Web services. In this case, the agent uses the Service profile to describe the *ideal* service that could solve the agent's problems. One simple way in which this can be achieved is by using the outputs and effects to describe what state should result from the invocation of the Web service. For instance the application may need a stock quote, therefore it would compile a Profile description whose output is indeed a stock quote.

By using the same concepts to describe the capabilities of Web services as well as the problems of the application, OWL-S provides a uniform way to represent the information that is needed during the discovery process. As a consequence the matching process between the two pieces of information becomes easier, and the matching can be performed more efficiently reducing the risk of incurring precision and recall errors.

The other two models of the OWL-S ontology are the *Process Model* and *Grounding*. The Process Model describes how the Web service achieves the expected goals. Specifically, the OWL-S Process Model describes the workflow that would lead the Web service to its goals. Each step in the workflow is either an *atomic process* that corresponds to an exchange of information between the application and the Web service, or a *composite process* that corresponds to a sequence of processes organized following a specific control flow. For instance a composite process may specify that all the sub-processes are executed concurrently, or that they are executed sequentially, or that one such subprocess has to be non-deterministically selected and executed.

Atomic processes describe abstract information exchanges between agents and Web services. Each atomic process is characterized by a set of inputs that the Web service expects to perform the process, and a set of outputs that the Web service produces as a result of the execution of the process. The definition of inputs and outputs in the atomic process specify abstract information exchanges between the agent and the Web service. Specifically, inputs specify the semantics of the information that the agent sends to Web service, and conversely, the outputs specify the semantics of the information that the Web service sends to the agent. In addition, the atomic process is characterized by a set of preconditions to the successful execution of the process, and a set of effects that result from the execution of the process.

Finally, the *Grounding* specifies how the abstract information exchanges that are specified by the atomic processes are realized by concrete information exchanges between the agent and the Web service. The Grounding maps atomic processes to operations in the WSDL description of the Web service, and it specifies a correspondence between atomic processes and WSDL operations, as well as a mapping between the OWL descriptions of inputs and outputs and the XML serialization defined in the WSDL messages.

OWL-S descriptions allow the agent to use a *goal directed approach* to the interaction with a Web service. For example, using the Service Profile the agent can express the goals that it wants to achieve and the requirements that need to be satisfied. Such a description can be used by matching processes to locate the best service that can achieve the agent's goal and satisfy the agent's requirements. Once the Web service is identified, the agent uses the Process Model to interact with it. The interaction with the Web service is also a goal directed process where the agent tries to find out which sequence of information exchanges leads to the solution of its problem. In addition, each one of these exchanges may have additional requirements that the agent has to satisfy, and they may require the interaction with other Web services. The result of using OWL-S is a *composition of Web services*, where each Web service achieves a specific goal, and Web services are gathered using a means-end analysis process.

The interaction process described above bears a strong resemblance with the Service Oriented Architecture (SOA) interaction protocol that is typically produced by using UDDI-WSDL-SOAP. This is no surprise, since OWL-S aims at enriching existing standards with explicit semantics rather than replacing them. But, while OWL-S reproduces the same interaction process that is described by SOA, there is one important exception: OWL-S does not require a programmer querying UDDI, reading the WSDL models found there, and implementing those interfaces. The agent can take the responsibility for the interaction with the registry, the decision of which candidate services are most appropriate, the determination of the information required to invoke each service, and the interpretation and response to messages returned by the service.

OWL-S is a language to describe Web services

OWL-S defines a language and an ontology for Web service description. As a language it describes the type of statements that can be made about Web services. Often time those statements are specified using the OWL syntax. But OWL-S provides alternative syntaxes, such as the OWL-S surface syntax (see OWL-S W3C submission.) As an ontology, OWL-S describes which are the main concepts to describe a Web service. Such an ontology specifies concepts such as Service, Process, Profile, Grounding and many more.

Since OWL-S is a language it does not make any architectural commitments. Therefore OWL-S can be used in the context of the SOA architecture specification in which the agent discovers a Web service and then interacts with it directly. But OWL-S can also be used in other contexts, for example it can be used with a centralized broker. Similarly, it has been shown that OWL-S can be use in architectures that are characterized by mediators that translate between different languages or that adjust the interaction protocol of different types of applications.

As OWL-S does not make any architectural commitment, it does not make any commitment on the type of inference processes required to interpret and execute an OWL-S specification. By design, OWL-S is better suited to be used with systems that perform means-ends analysis and with systems that can perform inferences consistently with OWL semantics. But OWL-S has been used also to control Java-based clients. In general, OWL-S can be used for automatic interaction with Web services, or it can be used for hard-coded Web services and any degree in between.

OWL-S builds on existing Web services standards

OWL-S is not intended to replace existing Web services standards, but to augment them with an explicit specification of the semantics of the terms that are used to describe the Web service, and the semantics of the messages exchanged between the agent and the Web service. The first point of relation between Web services standards and OWL-S is in the Grounding. OWL-S explicitly builds on top of WSDL; OWL-S atomic processes map directly to WSDL operations and the grounding provides also a mapping from the XML Schema specifications of the messages expected and sent by the Web service. The direct mapping to WSDL allows it to use OWL-S to represent Web services that do not use OWL directly. Specifically, given any Web service which is described by a WSDL description, it is possible to construct an OWL-S description for the same service using the grounding.

OWL-S has also been mapped to the UDDI representation. Such a mapping allows it to use UDDI as a registry for discovery of OWL-S based web services. UDDI provides a language for the description of Web services, and a registry infrastructure to store and retrieve Web service descriptions. But UDDI does not provide much support for capability based representation and search of Web services. OWL-S has the complementary strength, it provides a language for the representation of capabilities of Web services, and algorithms for the discovery of Web services, but it does not provide the registry infrastructure provided by UDDI. Exploiting the mapping for OWL-S into a UDDI representation, it is possible to combine the better of the two worlds. Such a Semantic UDDI can provide both a rich and standard registry for Web services, as well as the capability search that allows the UDDI clients to find Web services on the basis of what they do.

OWL-S builds on existing Semantic Web standards

OWL-S relies on existing Semantic Web standards such as OWL as well as on emerging standards such as SPRQL and SWRL. OWL concepts are used to represent the types of service inputs and outputs and may be mentioned in preconditions and effects of services. Such concepts are typically domain-specific, and can draw on Semantic Web ontologies developed for by communities of interest for a diversity of purposes. OWL is also the language in which OWL-S concepts themselves are specified.

Languages such as SWRL and SPARQL may be used to represent the preconditions and effects of processes. These languages are used to overcome limitations of OWL for describing complex constraints on temporal activities, such as those that occur in rules and queries. Whereas OWL is effective for describing taxonomic categories and properties of things, the structures found in SWRL and SPARQL are better suited to describing conditions requiring the use of variables. OWL-S' use of these languages enables more complete characterizations of services. It should be noted, however, that there is not as yet a consensus standard for unifying these semantic frameworks together as OWL-S uses them; thus, a hybrid reasoning approach is needed.

3.2. Functionality of OWL-S.

Here, we provide several simple scenarios to illustrate some of the functionality provided by OWL-S.

Semantic discovery. Service discovery is the task of finding an appropriate service, given a description of the properties of the service that the requester is seeking. Most existing practice relies on human perusal, of service descriptions provided as keywords and text strings, supported by text string matching algorithms that can filter the candidates if given the right keywords to use. This process is labor-intensive, requiring too much user involvement at runtime, and is hard to accommodate at development time without knowing the precise set of keywords to utilize. When service descriptions are based on bare-bones taxonomies, keywords, and/or English descriptions, the forms of requests and the precision of matching techniques are severely limited. When service providers and requesters use different terminologies to describe services, string-based search will frequently fail to discover appropriate services. On the other hand, if service descriptions are based on shared ontologies on the Semantic Web, the hierarchical class structuring and associated reasoning and mediation techniques available with OWL can be applied to produce more flexible and effective service discovery with less need for direct user involvement at runtime.

As a simple illustrating example, consider a requester who would like to buy ‘cutlery’. Let us assume that there are service providers that categorize themselves as sellers of kitchenware (including ‘knives’), but none that advertise ‘cutlery’. With string-based discovery, the requester will not locate any service providers that match the specified objective. (Of course some trial-and-error by a human would likely lead to a match, but here we are aiming to support software agents and tools as requesters.) OWL-S relies on OWL ontologies to describe services, including domain-specific terms. As a result it can overcome the limitations of string-based discovery by providing semantic matching based on widely shared domain ontologies. In this example, even if there were no providers advertising the domain class of cutlery, a requester using OWL-S could use ontological knowledge to request providers of kitchenware, as cutlery would be known to be a subclass of kitchenware.

Ontology-based approaches allow the request to be a description at a different level of abstraction than the descriptions of the service providers (as the matcher can reason about the relationship between the classes), while still allowing the requester to specify important details that might eliminate some subclasses that would otherwise match. For example, the request might include the detail that the provider needed to be located in a particular state, or sell the item at or below a particular price.

Composition. Web service composition is the task of combining independently developed and supported Web services to achieve some user objective. Composition can be achieved manually by specifying a workflow and associated dataflow between the services; it can be done automatically by a computer program; or it can be done by adapting existing workflows interactively for a particular user and task. In any of these cases, the agent performing the composition needs to know the conditions under which the chosen Web services can be executed and what kinds of effects their execution will have on the agent’s state of knowledge and on the state of the world. Key to any automated composition is the availability of software-interpretable descriptions of service capabilities and interaction requirements, and a computer program capable of interpreting and reasoning about them. In addition to the semantic characterization of inputs and outputs, effective composition tools need to consider the preconditions required for successful use of each service, and the effects they will have in the world. “Nonfunctional” attributes of service behavior, such as resource requirements and quality-of-service guarantees, may also be needed.

As an example, consider a scenario that describes two people trying to take their mother to a physician for a series of treatments and follow-up meetings. The problem is to come up with a sequence of appointments that will fit everyone's schedules, preferences and constraints. Creating a composition involves discovering new services based on various requirements, e.g. finding all hospitals which provide the treatment and are approved by the health insurance company, coordinating data between heterogeneous services, e.g. personal schedule management software and hospital appointment services, and reasoning about the effects of services without executing those services, e.g. can I schedule all my hospital appointments if I go to the doctor at this date. Finding compatible services and automating the sequencing and data flow between these services can be done most effectively when the input and output parameter types are described as terms in a way that lets reasoning software determine that the transfer is semantically appropriate. This requirement can be met by using an ontology system such as OWL, and a means of describing service structure (preconditions and effects) and function (activity type), such as that provided by OWL-S.

Semantic data integration and service interoperation. Semantic data integration is the task of integrating data according to its meaning. It is a pervasive challenge in the Web. In the context of Web services, semantic data integration is critical to service interoperation. In particular, composing Web services requires matching the output(s) of one service to the input(s) of another, and/or matching the effect(s) of one service to the precondition(s) of another. Matching based on syntactic descriptions or natural language text descriptions of these properties is not adequate. The same string term (or document type) may be used to describe different things, and different terms (document types) may be used to refer to the same thing. Data may need to be manipulated or transformed to enable it to serve as input to a subsequent Web service. These transformations must be meaning preserving, and services may need to be provided to effect these transformations. Existing mechanisms for describing Web services do not provide rich enough descriptive mechanisms to automate semantic discovery, integration, or the kinds of semantics-preserving translations needed for dynamic interoperation.

For example, consider a loan scenario in which online lender services require electronic transmission of credit report scores. Assume a particular case where the applicant has moved from the UK to the US and only has a credit report from the UK, which uses a different scoring system, while the lending services require US credit scores. An automated composition system might supply the UK score when asked for a credit score, because both are numbers labeled as credit scores, but the applicant might be denied a loan because the credit score was too low or was invalid because it was too high. If however, a credit translator service is capable of taking a UK credit score and translating it to generate a US credit score, then the pre-requisites for the lender service can be met and the buyer may obtain a loan. To realize this semantic integration requires explicit representations of the requirements and capabilities of services in a machine interpretable form.

Pervasive computing. As the devices we use in our everyday life, like televisions and music players in our home or projectors and printers in our office, become more sophisticated and net-accessible, we gain the ability to automate aspects of their usage. The convergence between device and Web service standards (e.g., both WSDL-based Web service descriptions and UPnP-based device descriptions rely heavily on SOAP as the messaging protocol) makes it possible to describe the functionalities of devices as Web services.

As an example, consider a business meeting where one or more people are doing presentations. In such a setting, one might want to do several different tasks that will require

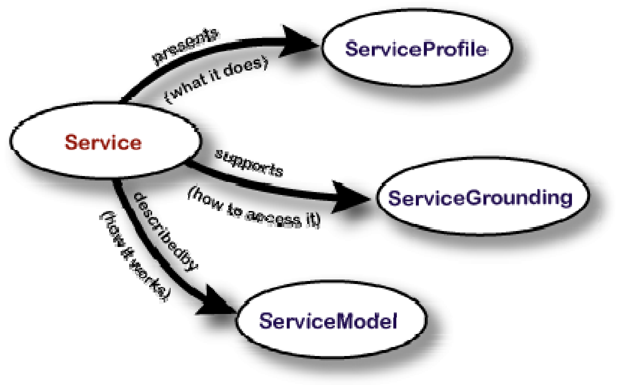


Figure 1. ***Top level of the service ontology.***

interacting with the devices and other people in the environment. For example, such tasks might include finding out the e-mail addresses of everyone in the conference room and e-mailing the presentation document, printing out the presentation handouts, adjusting screen resolution based on the capabilities of the projector, and controlling the lighting in the room when the presentation begins and ends. Realizing such scenarios in the real world not only requires description of the capabilities of the devices, but also requires description of security and privacy policies. Some assurance is needed that the service providing contact information will only disclose such information to appropriate requesters. Similarly, the printer or the projector service should not let anyone use those devices arbitrarily. The discovery and matchmaking functionality provided by OWL-S facilitates composing services in a way that meets these kinds of constraints as they occur in pervasive computing environments. The extensible nature of the OWL-S description language enables the individual devices in that environment to be described and also addresses critical security and privacy concerns.

The OWL-S Web site http://www.daml.org/services/owl-s/1.2/ includes pages listing related publications, tools, and use cases from the community at large. In addition, a number of open-source tools are hosted at www.semwebcentral.org. Discussion of issues related to OWL-S are conducted on the public mailing list of the W3C's Semantic Web Services Interest Group http://www.w3.org/2002/ws/swsig/.

3.3. Lessons Learned and Future Work

OWL-S has been used in many ways, to address a range of challenges, and numerous extensions have been proposed and demonstrated in prototype systems. We believe this indicates both the need for this kind of approach, and its flexibility and generality.

For OWL-S to become widely used will require that it become more tightly integrated with commercial Web services standards and that mature tool support makes it easier to use for non-expert developers. Further work on algorithms that effectively support discovery, selection, and composition of services in large-scale, complex settings is also needed. In addition, it will be important to conduct empirical evaluation of the applicability and benefits of OWL-S in developing and managing service-based systems in businesses and other settings.

There is a need for more complete guidelines and documentation of best practices regarding architectural configurations using OWL-S and the expected behavior of common components, such as execution engines, matchmakers, and service composers. Other future work includes fundamental ontology extensions to support error-handling, additional work on security and quality-of-service specifications and also extensions to allow for the specification of specific instances (executions) of process models, which are needed to support monitoring and recovery. Other fundamental extensions are needed to support negotiation and contracting.

Finally, there is a need to develop specializations of the profile for a variety of domains, and for broad categories of services. For example, for services that sell goods, ontology modules are needed for the specification of cost models and product guarantees.

We see OWL-S as having made a critical first step in the process of formulating the necessary declarative representational framework for fully specifying the capabilities and behavior of Web services, in a way that supports greater automation of service-related activities, such as discovery and composition. We believe that it should continue to be exercised and developed to meet the needs of the full array of such activities, including negotiation and contracting, monitoring and recovery, and so forth. We expect that the need to better support such activities in the world of Web services will lead to significant further evolution of OWL-S.

4. Development of OWL-S Tools

At CMU we have developed a set of tools that comprises all activities of developing, deploying, discovering and invoking OWL-S based Semantic Web Services. Our tools are are available from www.semwebcentral.org , also from our home page www.cs.cmu.edu/~softagents. The tools have been downloaded and used by hundreds of users. Here we give a brief description.

4.1. An Integrated Environment (IDE) for OWL-S Service Construction and Deployment

A Web Service developed engages in the following activities in order to implement a web service: first she needs to implement the Web service, then provide a WSDL description of the Web service interface, and finally address the OWL-S description of the Web Service creating a Process Model that is faithful to the actual Web service implementation, a Profile for discovery and finally a Grounding to map the Process Model to the WSDL description of the Web service. The compilation of OWL-S descriptions of Web services requires a considerable effort since it forces the developer to describe a Web service at different levels of abstraction.

At CMU we have performed these activities many times in our daily construction of Web services. We came to realize that to facilitate the use of OWL-S does not require only one tool, but a collection of tools that address the three main aspects of the OWL-S development: (1) the compilation of the Web service description, (2) the verification that the specification is correct and (3) the elimination of inconsistencies across modules of the description.

Programming tasks to develop and deploy a Semantic Web Service:

- Java coding to run the Web service (this is supported by Java2WSDL Tools –open source)
- Translate the resulting WSDL to OWL-S: Done through our CMU tool WSDL2OWL-S Converter
- Generate OWL-S
    - Select/Construct Ontologies
    - Generate/Edit OWL-S Profile (done through our CMU tool *OWL-S Profile Editor)*

    - Generate/Edit OWL-S Process Model (done through our *CMU Process Model Editor)*
    - Verify Process Model for executability and consistency checking (done through our *CMU OWL-S verification tool*)
- Deploy the Web Service (done through the open source AXIS tools)
- Advertise with OWL-S/UDDI Matchmaker (this is done automatically through our *CMU OWL-S2UDDI tools)*

Figure 2 provides a detailed view of the activities of a service provider, and of the our implemented IDE to support these activities. The first task of the developer is to implement the Web service and to generate a WSDL description for the Web service. In most cases, the generation of WSDL is supported by open source tools such as Java2WSDL. The second task of the developer is to generate the OWL-S description for the Web service. This task is partially supported by our WSDL2OWL-S, a tool that generates a virtually complete Grounding and a skeletal OWL-S Process Model and Profile exploiting the relation between WSDL and OWL-S. While WSDL2OWL-S greatly facilitates the developer activities by providing automated generation of the Grounding, Atomic Processes, and part of the Profile, much work is left to be done. Specifically, the developer has to add all the composite processes to the Process Model, also she has to complete the Profile non-functional parameters and finally add the entire security markup. The third task is the verification of the OWL-S Web service description to verify its correctness. To our knowledge, no tools are currently available to facilitate this task. The lack of tools prevents a thorough evaluation essentially pushing to run time problems that could have been detected at development time. As a consequence, complex OWL-S descriptions are likely to contain bugs. Note that the introduction of security requirements and Policies in OWL-S is bound to make the problem harder by increasing the number of features that the developer has to take care of. After the OWL-S description has been generated, the fourth task of the developer is to deploy the Web service. The deployment is often supported by existing tools and it requires very little effort. The fifth and final task is to register the Web service with UDDI. This task is supported by OWL-S2UDDI, a tool that we implemented at CMU, that transforms OWL-S Profiles into OWL-S /UDDI service descriptions and that registers the service description with OWL-S /UDDI. Subsequently, the OWL-S/UDDI Matchmaker can semantically match the OWL-S advertisement to service requests.

Our analysis shows that the two crucial types of tools that would greatly facilitate the use of OWL-S are editing tools and verification tools. In our vision, these tools should support the developer through the generation process by detecting inconsistencies in the description of the Web service, and, when possible, suggesting solutions. This process is similar to the development process supported by IDE tools for standard programming languages, which try to detect problems at development and compilation time, reducing the likelihood of execution time errors.

We have implemented CODE, an Integrated Development Environment for OWL-S that provides tools to address these issues. CODE extends the Eclipse Java IDE, which provides both drawing and text editing tools. Eclipse provides a plug-in mechanism that allows adding new functionalities, and it offers a platform to integrate all aspects of the Web service development from Java code generation to UDDI advertisement. CODE is available form www.semwebcentral.org.

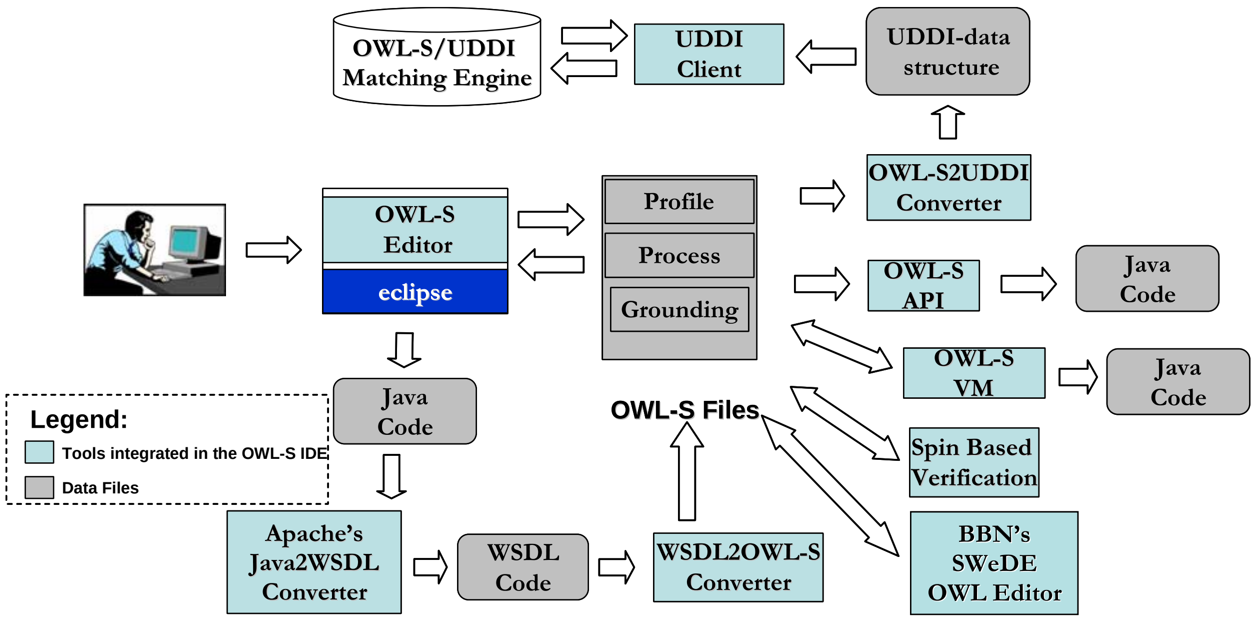


Figure 2. The implemented CMU IDE to support development and deployment of OWL-S based Web Services

4.2. Tools to Support Requesters (Clients)

OWL-S specifies the requirements of the provider and implicitly expects requesters to adhere with these requirements. The requester needs to be able to construct a request that specifies the capabilities that it needs, and post the request to a capability based registry such as the UDDI. Then, the OWL-S/UDDI matchmaker will find the appropriate providers and send them to the requester. The requester will select a provider and invoke it.

The OWL-S editor can be used to generate the capabilities requests, then the tools that we developed at CMU, such as OWL-S2UDDI and the OWL-S Virtual Machine, can be used to discover Web services that provide desired functionalities and to manage the rest of the interaction with them.

Below, we show the activities of a requester who would like to discover and involve a Semantic Web Service and the tools that we have developed in support of those activities.

- Generate Request OWL-S Profile (supported by CMU OWL-S Editor)
- Query OWL-S/UDDI Match maker (CMU OWL-S2UDDI tools)
- Match request to advertisements in the Matchmaker's knowledge base (CMU Matching Engine)
- Select a Web Service (internal process of requester)

- Load OWL-S Process Model and Grounding and Execute the Web Service (CMU OWL-S Virtual Machine)

4.2.1. Service Discovery

We have developed a variety of algorithms and tools for Semantic Web Services Discovery.

- OWL-S/UDDI Matchmaker
- OWL-S Broker
- Peer to Peer Discovery

OWL-S Matchmaker

Our current tool to support Web Service discovery is the OWL-S/UDDI matchmaker which integrates OWL-S with UDDI functionality. The decision to integrate OWL-S matching with UDDI was a strategic one: since UDDI is the emerging de facto industrial standard for Web Services, the OWL-S profile matching integration with UDDI allowed us to highlight the contribution of OWL-S to the growing industrial Web services infrastructure. The cost of this decision is that it also forces every OWL-S developer to use UDDI. This dependence on UDDI presents an OWL-S developer with many disadvantages. First of all, UDDI is quite heavy, and at times quite unwieldy to deal with. Furthermore, the UDDI API is limited only to advertisement and query for information. Nevertheless, since UDDI is an industry standard, we have developed the mapping of UDDI internal representation to OWL-S and integrated our semantic matching engine so that advertisements that are semantically annotated can be matched against incoming queries. Note also, that our construction of appropriate APIs allows both UDDI-only capable queries to be matched. Figure 3 presents the UDDI to OWL-S bindings and Figure 4 presents the overall Matchmaker architecture.

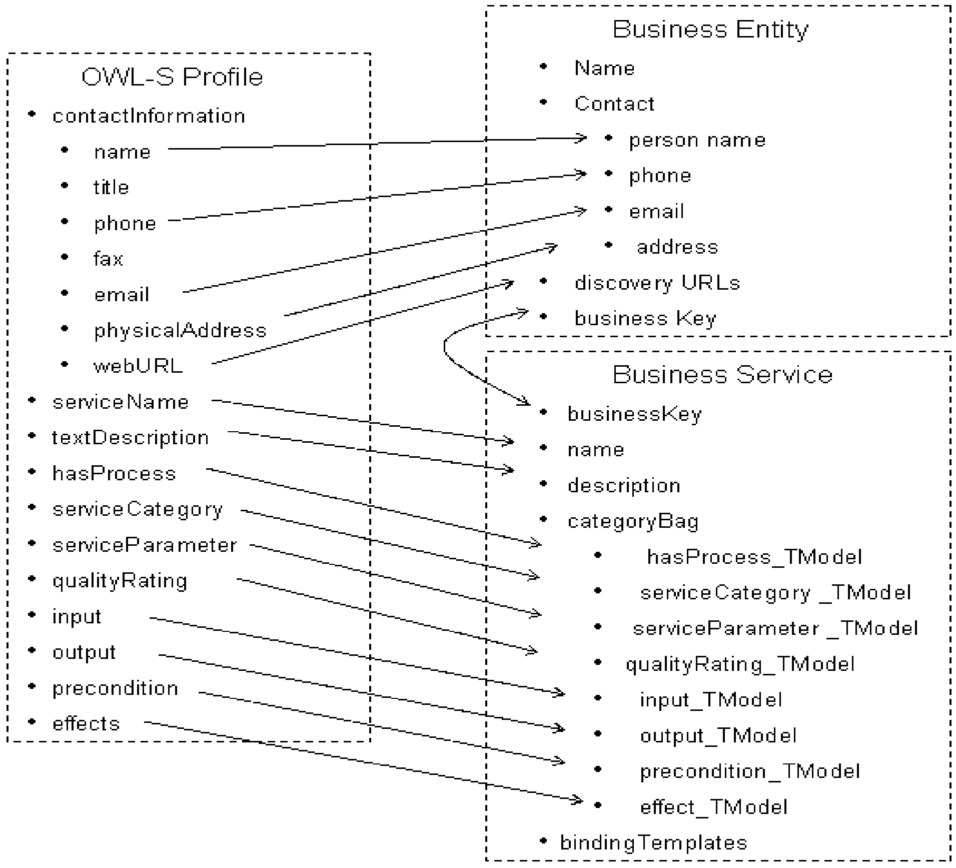


Figure 3. OWL-S to UDDI Mapping

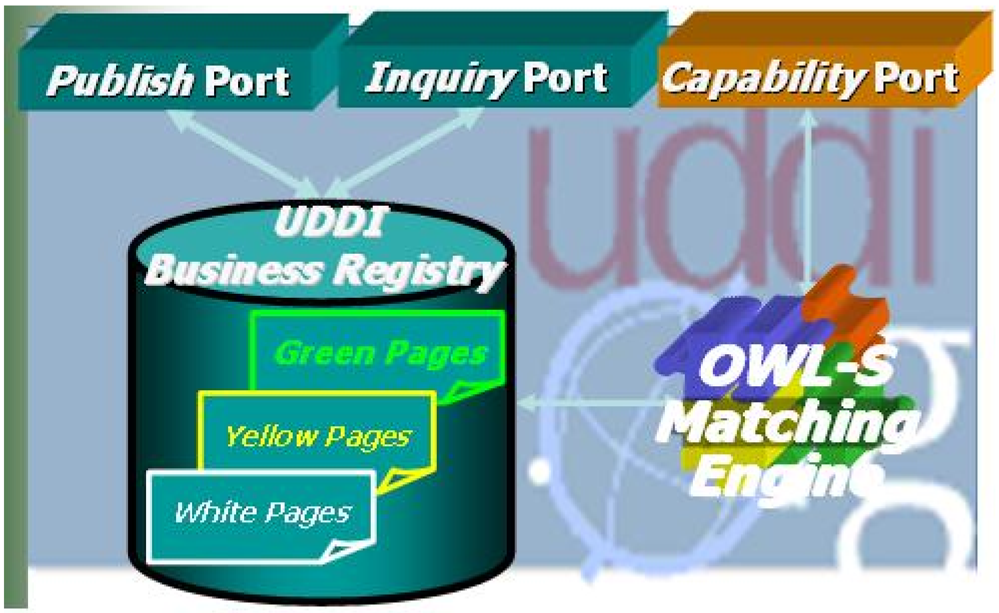


Figure 4. The UDDI/OWL-S Matchmaker

The architecture of the OWL-S Matchmaker is shown in Figure 4. It includes an Advertisement DB that stores the advertisements and a Matching Engine that exploits OWL ontologies to match a request against the advertisements in the Advertisement DB. The API of the OWL-S Matchmaker exposes query and advertise capabilities. Advertisements are stored in the Advertisement DB, while queries are directed to the Matching Engine. The OWL-S matchmaking component is embedded into jUDDI, an open source UDDI registry. The matchmaker and jUDDI can access each other functionalities. We use Racer DL engine, description logic reasoning system, to process semantic data.

The UDDI registry, on receiving an UDDI advertisement containing OWL-S Profile for publishing, processes the advertisements and forwards the advertisement to the matchmaking component. The matchmaker first validates the correctness of the advertisement using the Racer DL engine. After validation, the advertisement is classified based on its semantic information using Racer and stored in the repository. On receiving a query, the matchmaker validates the query, similar to the validation performed on an advertisement, and then matches across all the published advertisements using the matching algorithms we have developed (see matchmaking papers in the Appendix).

The naïve way to implement a matching algorithm, the inputs and the outputs of the request are matched against the inputs and the outputs of all the advertisements that are present in the repository. As the number of advertisements in the repository increases the time to process each query will also increase. To overcome this limitation in our implementation, we perform most of its matching reasoning during the publishing the advertisements itself and store those results to be used during request time. The rationale underlying our approach is that since the publishing of an advertisement is a one-time event, it makes sense to spend time to process it and store partial results and speed up the query processing time, which may occur many times and furthermore it is time critical.

In order to maintain information about the published advertisements, the matchmaker maintains a hierarchical tree structure that represents the subsumption relationships between all the concepts that form the inputs and outputs of all the advertisements published in the repository. Each node in the hierarchical structure represents a concept present in the matchmaker, it also maintains information about degree of match between each output of all the advertisements and the concept it represents. If degree of match between the node's concept and the output of an advertisement is "fail", then the information is not stored. Similarly, each node maintains the degree of match between all the inputs of the advertisements and the concept it represents.

When an advertisement is submitted, the matchmaker loads the ontologies that are used by the advertisement's inputs and outputs into the Racer system and updates its hierarchical tree. The Racer system provides a network API to access its information, which is used to construct and update the hierarchical tree structure.

For each output of the advertisement, the matchmaker extracts the concept which represents the output and locates the corresponding node in the hierarchical data structure. The degree of match between this node's concept and the output of the advertisement is exact, so the matchmaker updates the node with this information. For example, let us assume that the matchmaker maintains a hierarchical tree as shown in Fig. 5 and let an output of an advertisement be Car. The matchmaker updates the information of the Car node that it matches the advertisement exactly. According to the algorithm, the degree of match between output and the concepts immediate subclass are also exact, so the matchmaker updates the node information of all the nodes that are an immediate child of the current node. In our example the matchmaker updates the node information of Coupe and Sedan that it matches the advertisement exactly.

We can also observe that the degree of match between the output and the concepts of all the parent nodes of the current node is subsume. In our example the nodes Thing and Vehicle subsumes an advertisement whose output is Car. The matchmaker updates the node information of all the parents of the current node that the degree of match between the node and the advertisement is subsume. Similarly we can observe that the degree of match between the output and the concepts of all the child nodes, except the immediate child nodes, is plug in. Following our example the degree of match between concept Luxury and an advertisement, whose output is Car, is plug in. The matchmaker performs similar updates to the hierarchical tree for all the outputs and inputs of the advertisement.

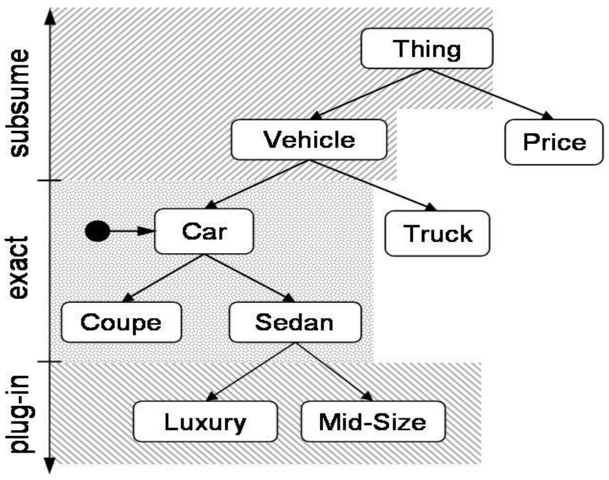


Figure 5. Subsumption Reasoning in Matching

During the publishing phase we are performing most of the work required by the matching algorithm, hence we may spend a considerable amount of time in this phase. But we can show that time spent during this phase does not depend linearly on the number of concepts present in the data structure, but in the order of log of concepts in the data structure, and hence showing that our implementation is scalable. Since we use hierarchical data structure, the time required to insert a node will be in the order of $\log_d$ N, where d is the degree of tree. Similarly, the time required to traverse between any two nodes in a particular branch will also be in the order of $\log_d$ N.

When the matchmaker receives a query, it retrieves all the sets of advertisements that match each output of the request. For example, if the outputs of the request are Car and Price, the matchmaker fetches the information, that mentions the degree of match of between the nodes, in this case car and price, and the outputs of the advertisements. The matchmaker then finds the advertisements that are common between the sets of advertisements retrieved. If no intersection is found then the query fails. If common advertisements are found, they are selected for further processing.

The matchmaker performs a lookup operation and fetches the sets of advertisements for the inputs of the request, similar to the one it performed for the outputs. However, instead of finding the intersection between the sets of advertisements, the matchmaker only keeps the information about advertisements that were selected during the output-processing phase. The resulting advertisements that matched the request's inputs are used to score the advertisements that were selected during the output-processing phase.

OWL-S Broker: Discovery and Mediation

UDDI has been adopted as the main discovery mechanism for Web services, and through our work, for Semantic Web services. But a centralized registry such as UDDI is just one of the types of middle agents that can populate a Web services infrastructure. Brokers (sometime also called mediators or facilitators) provide an alternative view of discovery in which the broker locates a provider on behalf of the requester, and then mediates the interaction between the provider and the requester.

Brokers have been widely used by the Multi-agent community because they facilitate the tight coordination between agents in the Multi Agent System (MAS). In addition, brokers have been proven to be able to do better load balancing than matchmakers.

To mediate between the requester and the provider using OWL-S, the broker needs to adopt a Process Model similar to the Process Model of the provider. This way it can *mediate* and simulate the interaction between the provider and the requester and ask the requester all the information that the provider needs in order to process the request. The problem of the broker is that it does not know the requester's needs until it receives a query from the requester. Therefore, the broker is in the awkward situation in which it cannot expose the appropriate Process Model to the requester because it does not know what the requester needs; by the same token the broker cannot know what the requester needs until it exposes a Process Model. To break this paradoxical situation, at CMU we have been experimenting with *an extension of OWL-S that allows Web services to load a Process Model dynamically.* Using this extension, the broker first learns what are the needs of the requester, then it gives a directive to load a new process model that simulates the interaction with the provider. This extension can also be used to extend OWL-S to multi-party interactions, because the provider and the requester may dynamically load the Process Model of the new party and interact with it.

The other contribution of the study of the Broker is that it highlights very difficult and profound questions about interaction of Semantic Web services on the way toward automatic composition. For example, the Broker may receive information from the requester and needs to transform it in a way that is understandable to the provider of the same service. Furthermore, this translation should be driven by the semantics of the content of the information transmitted. Essentially, this problem hits at the core of the reasons for using Ontologies to represent the content of the information exchanged, and to the problem of composition of web services. Whereas all research to date has concentrated on how to use planners to combine Web services, virtually no effort has been devoted on what information is to be transferred and how.

We have a prototype of the Broker running in a very limited domain. We also have implemented an extension of OWL-S that we are currently investigating and we will propose it to the OWL-S coalition to address the problem of multi-party interactions.

.

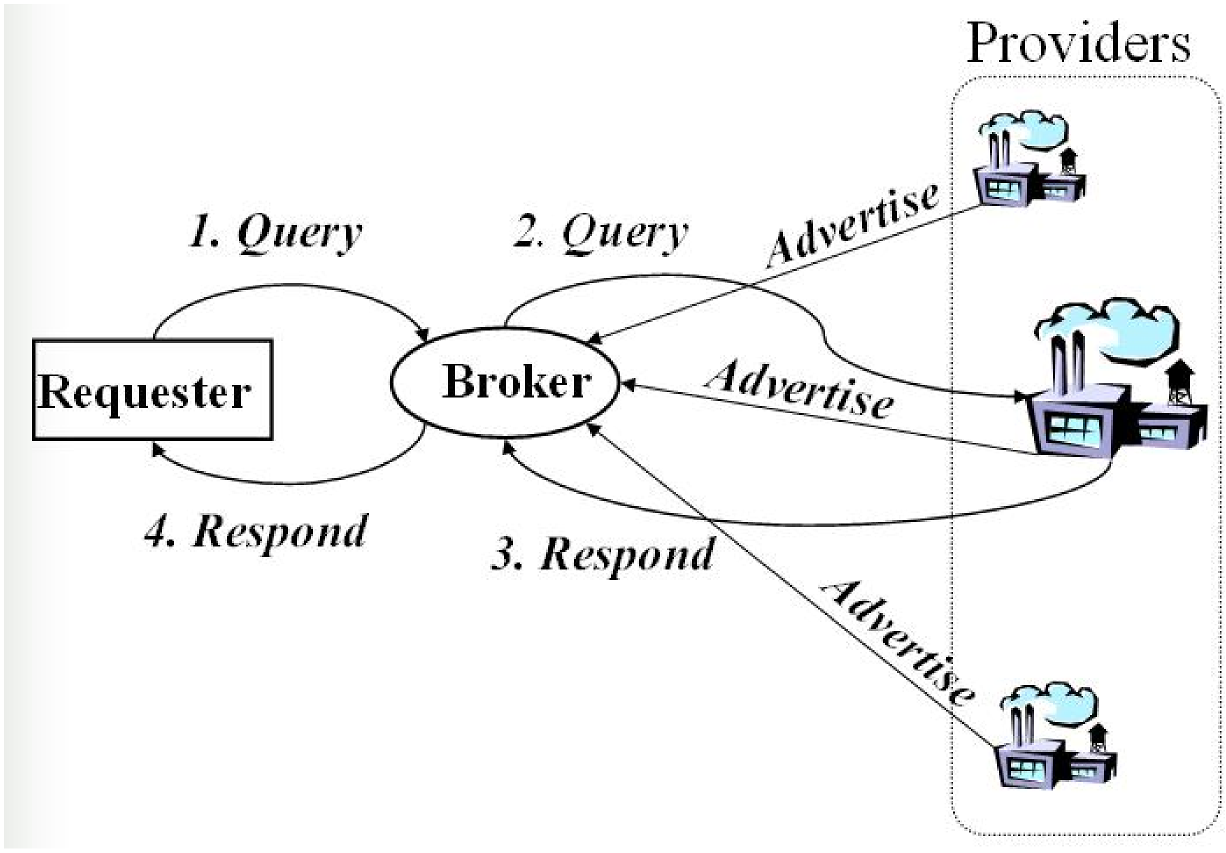


Figure 6. The Broker Protocol

Peer-to-Peer Discovery

Our architecture for distributed discovery infrastructure can be divided into two layers, see Figure 7, namely *Peer-to-Peer layer* to maintain the underlying dynamic network and *Discovery layer* to handle the service discovery process.

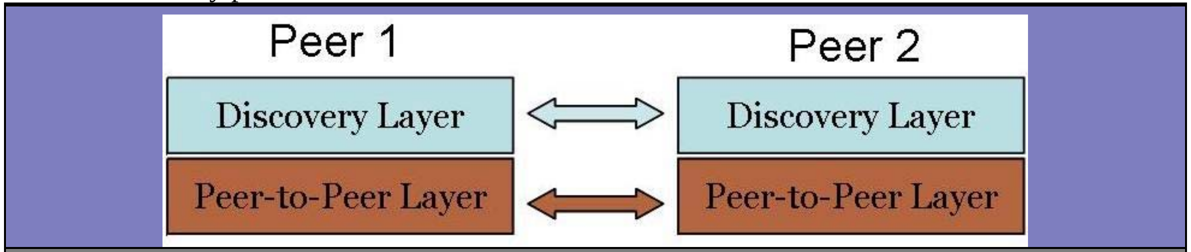


Figure 7 Distributed Discovery Architecture

Peer-to-Peer layer

The Peer-to-Peer layer provides the core peer-to-peer service. The function of this layer is to make sure the nodes are reachable through the peer-to-peer network and to handle the changes in the network topology. This is achieved by sending messages among the nodes in the network similar to the ping/pong messages in the Gnutella network. The peer-to-peer layer is also responsible to support point-to-point conversation between two nodes in the peer-to-peer networking using the existing nodes in the network.

Discovery Layer

The discovery layer is layered on top of the peer-to-peer layer. The discovery layer is responsible for providing the enhanced service discovery mechanism in the peer-to-peer network. The nodes in the discovery layer can possess any or all the three of the following functionalities: server, matchmaker and client. A server provides service in the network that is accessible by other nodes in the peer-to-peer network. A matchmaker acts as a registry and provides the matchmaking functionalities in the peer-to-peer network. The matchmaker uses the matching algorithm of the OWL-S Matchmaker to match between requests and advertisements. A client consumes the services provided in the peer-to-peer network. A node in the network may select to enable all or any of the above functionalities based on its resource availability.

Discovery Protocol

When the matchmaker component is initiated, it joins the peer-to-peer network using the functionality provided by the peer-to-peer layer, next it announces its matchmaker service to the peer-to-peer network and then waits for service advertisements and queries other nodes in the network.

A server provides services in the network that are consumed by clients. The server has to register its services with a service registry so that the clients could find the services they provide. When the server component starts, first it joins the network using the peer-to-peer layer; secondly it searches for matchmaker services in the network, and finally registers the service description to the matchmakers.

When a client requires a service it first joins the network using the peer-to-peer layer, secondly it searches for the matchmakers in the network, and finally it queries the matchmaker for the required service and receives the results from the matchmaker. The client then selects a service from the results and invokes it.

Implementation

The implementation of our architecture is based on JXTA. JXTA or Juxtapose is a networking framework that provides a simple and flexible mechanism to support P2P computing on any platform, anywhere, and at any time. JXTA focuses on providing the bare necessities for building generic P2P applications and leaves the application choices to developers.

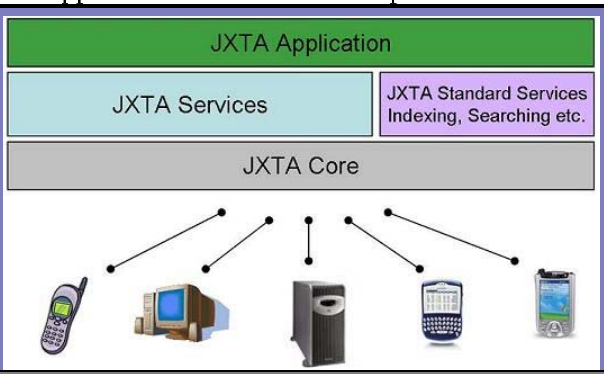


Figure 8 JXTA Protocol Stack

JXTA is abstracted into the following layers - core, service and application. The core layer is responsible for implementing the bare P2P protocols like maintaining the connection to the P2P network, providing point-to-point communication, creating and maintaining the virtual groups, etc. The service layer uses the core layer to provide basic services like searching, indexing etc. Developers can also develop their own services using the service and the core layer. The application layer provides functionality to develop applications using JXTA.

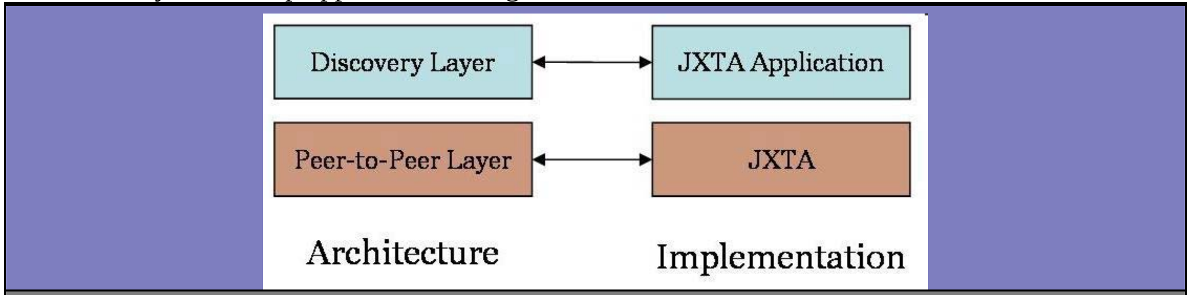


Figure 9 Implementation

In our architecture, the Peer-to-Peer layer is realized using the JXTA core and service layers. The discovery layer of our architecture is implemented as a JXTA application. The functionalities such as matchmaker, server and client are configurable and can be configured while the application starts. We have developed a user interface to interact with the client component. Users can create and issue queries using the interface. The results of queries displayed in the user interface are used to execute the service.

1.2.2. OWL-S Virtual Machine (VM) for Service Interaction

The OWL-S Virtual Machine (OWL-S VM) is the CMU implemented software that allows a requester to interact with a Semantic Web Service. After a requester has found a suitable web service, it interacts with it to satisfy its goals. The process of interacting with a Semantic Web service has the following steps:

1. **Invocation:** The client determines how to make a request to the selected service by unifying its goals and preferences with the effects specified in the service process model, to determine a semantically valid set of inputs to the server. More specifically, invocation is the process by which a client applies a declarative description of a service to request something of the service and interpret the response. Here, the description being interpreted is the OWL-S process model published by the service along with the WSDL specs to which it is grounded. Invocation begins by reasoning backwards from the inputs required by the selected service to find the information available to the client that is required to successfully invoke the service. These input values are then mapped via the service grounding onto the corresponding elements of a WSDL message pattern, resulting finally in a message being communicated to the service. The output message (if any) is handled by essentially reversing the process. A WSDL output message that is received by the client is transformed (again, via the grounding) into an OWL-S representation of the content of that message which can be interpreted by the client's reasoning engine.
2. **Reply:** The service receives the request, and determines whether it can perform the request. It may acknowledge the request, send an error, request additional information, or (generally, on completion) send a reply stating the service results.
3. **Reply Interpretation:** Upon receiving the reply, the client parses the reply, in accordance with its WSDL specification, and uses the grounding specification to map it into the abstract outputs specified in the process model. If the invocation of the service is successful, the effects of the service hold true in the client's world. This additional knowledge may satisfy preconditions for other services.

4. **Goal satisfaction:** The client uses the instantiated outputs to decide whether its goal has been achieved successfully, or whether there is a need for additional interactions with the Web service or for searching for another Web service.

When the process model of a service is a composite process, steps 1,2,3 may be repeated a number of times, since the interaction between the client and the server may require numerous information exchanges. For example, any interaction with a B2C Web site such as Amazon's requires the buyer to fill out multiple forms specifying such things as which book to buy, selecting the appropriate edition, specifying where to ship the book, describing which credit card to use, etc.

The diagram in Figure 10 shows our design and implementation of the architecture of the OWL-S Virtual Machine. The core of the architecture is represented by three components in the center columns: the Web service Invocation, the OWL-S Processor and the OWL Inference Engine. The Web service Invocation module is responsible for contacting other Web services and receiving messages from other Web services. The transaction with other Web services may be based on SOAP messaging, or on straight HTTP or any other mode of communication as described by the WSDL specification of the Web service provider. Upon receiving a message, the Web service invocation extracts the payload, or in other words the content of the message, and either sends it to the OWL Inference Engine or passes it directly to the OWL-S Processor.

The OWL Inference Engine is responsible for reading fragments of OWL ontologies and transforming them into predicates that can be used. The OWL Inference Engine is also responsible for downloading OWL ontologies available on the Web, as well as OWL-S descriptions of other Web services to interact with.

The OWL-S Processor is the center of our implementation: it uses the ontologies gathered from the Web and the OWL-S specifications of the Web services to make sense of the messages it received, and to decide what kind of information to send next. To make these decisions the OWL-S Processor uses a set of rules that implement the semantics of the OWL-S Process Model and Grounding. The OWL-S Processor is also responsible for the generation of the response messages; to accomplish the latter task, the OWL-S Processor uses the Grounding to transform the abstract information exchanges described by the Process Model into concrete message contents that are passed to the Web service Invocation Module to be transformed into actual messages and sent off to their receivers. The current implementation is based on the OWL-Jess-KB, an implementation of the OWL axiomatic semantics for the Jess theorem prover and the Jena parser. The Jena parser parses ontologies and asserts them as new facts in the Jess KB.

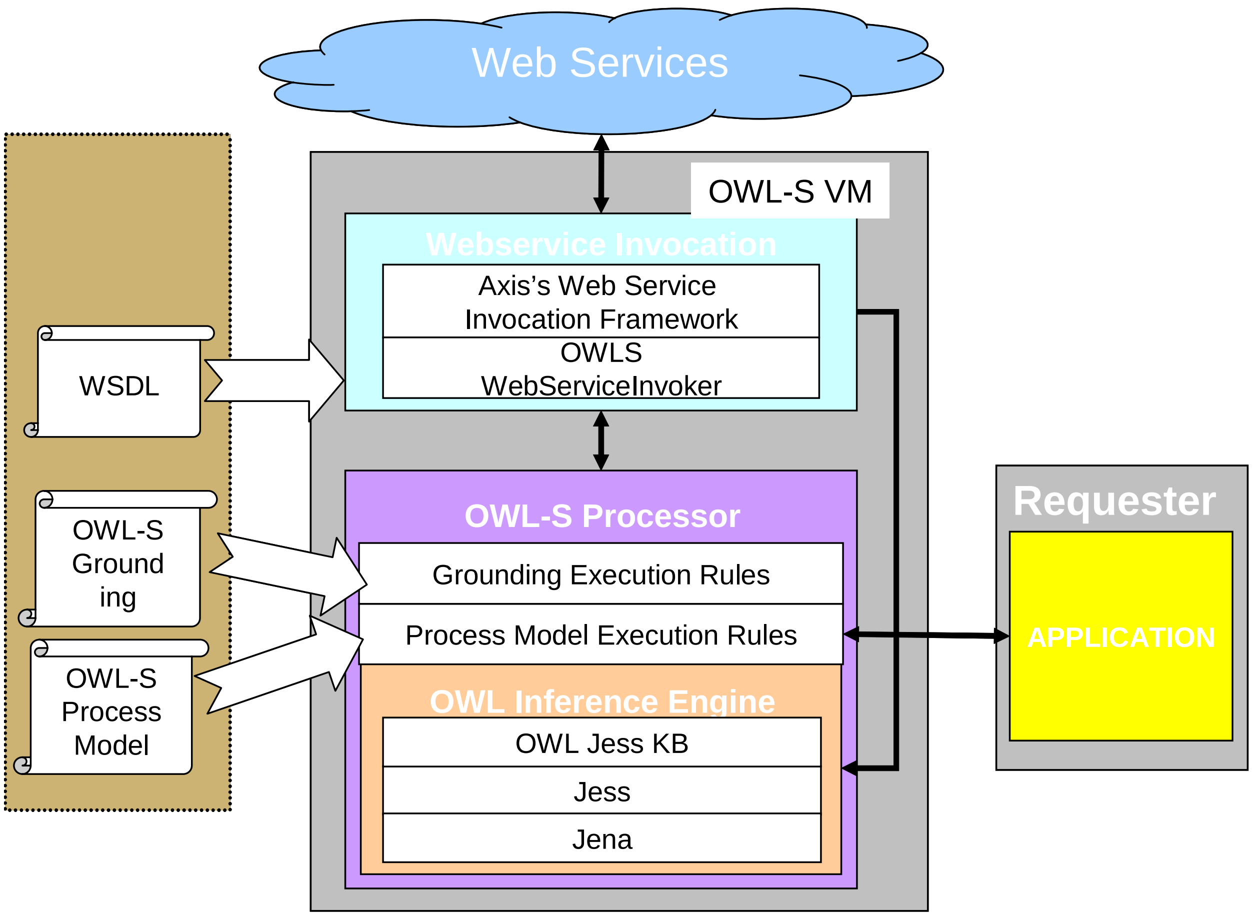


Figure 10: The CMU OWL-S Virtual Machine

The other two columns of the diagram in Figure 10 are also very important. The column on the left shows the information that is downloaded from the Web and how it is used by OWL-S Web services. Specifically the WSDL is used for Web service invocation, while ontologies and OWL-S specifications of other Web services are first parsed and then used by the OWL-S Virtual Machine to make decisions on how to proceed. The column on the right shows the Requester, which is displayed essentially as a black box.

OWL-S does not make any explicit assumption on the Requester's reasoning since its goal is to facilitate autonomous interaction between Web services and their requesters. Nevertheless, the Requester module is responsible for many of the decisions that have to be made while using OWL-S. The Requester is responsible for the interpretation of the content of the messages exchanged and for its integration with the general problem solving of the interaction. The Requester is also responsible for Web services composition during the solution of a problem. Specifically, the Requester module is responsible for the decision of what goals to subcontract to other Web services, or what capability descriptions of potential providers to submit to a OWL-S/UDDI Matchmaker; furthermore, it is responsible for the selection of the most appropriate provider among the providers located by the Matchmaker.

5. Development of Algorithms for Service Discovery, Service Invocation, Service Verification and Composition

In the course of the performance of the funded research, we have developed a variety of algorithms that cover many aspects of developing, verifying, discovering, interacting and composing Semantic Web Services. In particular, we give below a list of the different algorithms and the publications where the interested reader can find technical details. **The papers are downloadable** from www.cs.cmu.edu/~softagents.

In addition, a small number of technical papers is included in the appendix.

List of algorithms and techniques with corresponding citations in the Publications List:

Semantic Web Services (Discovery, Invocation, Composition) (9, 12, 16, 19)

DAML-S/OWL-S General (7, 28)

Brokering (2, 5)

Choreography (11)

Matchmaking (1, 6, 15, 21, 29)

P2P Discovery (18)

Formal Execution Semantics of DAML-S/OWL-S (22, 26)

Ontologies (8)

Security (4, 10)

Verification (3)

Applications (13, 14, 17, 20, 23, 24, 25, 27, 30)

Publications

1. T. Kawamura, T. Kasegawa, A. Ohsuga, M. Paolucci, and K. Sycara, “Web Services Lookup: A Matchmaker Experiment”, in IEEE IT Professional, Vol 7., No. 2., March/April 2005.

2. M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara, “A Broker for OWL-S Web Services”, in Cavedon, Maamar, Martin, Benatallah, (eds) Extending Web Services Technologies: the use of Multi-Agent Approaches, Kluwer, 2005.

3. A. Ankolekar, M. Paolucci, and K. Sycara, “Towards the Formal Verification of the OWL-S Process Models”, in Proceedings of the Fourth International Semantic Web Conference (ISWC-05), Galway, Ireland, November 6-10, 2005.

4. L. Kagal, G. Denker, T. Finin, M. Paolucci, N. Srinivasan and K. Sycara, "An Approach to Confidentiality and Integrity for OWL-S", in Proceedings of AAAI 2004 Spring Symposium.

5. M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara, "Untangling the Broker Paradox in OWL-S", in Proceedings of AAAI 2004 Spring Symposium.

6. N. Srinivasan, M. Paolucci, and K. Sycara, "Adding OWL-S to UDDI, implementation and throughput," in First International Workshop on Semantic Web Services and Web Process Composition **(SWSWPC 2004) 6-9, 2004, San Diego, California, USA.

7. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. R. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara (SRI, CMU, Univ. Toronto) "Bringing Semantics to Web Services: The OWL-S Approach." First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) 6-9, 2004, San Diego, California, USA.

8. K. Sycara and M. Paolucci, "Ontologies in Agent Architectures," in Handbook on Ontologies in Information Systems, 2004.

9. K. Sycara, M. Paolucci, A. Ankolekar and N. Srinivasan, "Automated Discovery, Interaction and Composition of Semantic Web services," in Journal of Web Semantics, Volume 1, Issue 1, September 2003, pp. 27-46.

10. G. Denker, L. Kagal, T. Finin, M. Paolucci, N. Srinivasan, and K. Sycara, "Security For DAML Web Services: Annotation and Matchmaking," in Proceedings of the Second International Semantic Web Conference (ISWC 2003), Sandial Island, Fl, USA, October 2003, pp 335-350.

11. M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura, "Toward a Semantic Choreography of Web Services: From WSDL to DAML-S," in Proceedings of the First International Conference on Web Services (ICWS'03), Las Vegas, Nevada, USA, June 2003, pp 22-26.

a. in ..ps.gz

12. M. Paolucci, A. Ankolekar, N. Srinivasan and K. Sycara, "The DAML-S Virtual Machine," in Proceedings of the Second International Semantic Web Conference (ISWC), 2003, Sandial Island, Fl, USA, October 2003, pp 290-305.

13. A. Ankolekar, J. D. Herbsleb, and K. Sycara, "Addressing Challenges to Open Source Collaboration With the Semantic Web," in Proceedings of Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, the 25th International Conference on Software Engineering (ICSE), 2003, Portland OR, USA, May 3-10 2003, pp 9-13.

14. A. Ankolekar, Y. W. Seo, and K. Sycara, "Investigating Semantic Knowledge for Text Learning," in ACM SIGIR-2003 Workshop on Semantic Web, Toronto, Canada, August 1, 2003.

a. in .ps.gz

15. T. Kawamura, J. A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara, "A Preliminary Report of a Public Experiment of a Semantic Service Matchmaker combined with a UDDI Business Registry," in 1st International Conference on Service Oriented Computing (ICSOC 2003), Trento, Italy, December 2003.

16. M. Paolucci, and K. Sycara, "Autonomous Semantic Web Services," in IEEE Internet Computing, vol. 7, #5, September/October 2003, pp 34-41.

17. M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura, "Towards a Semantic Web Ecommerce," in Proceedings of 6th Conference on Business Information Systems (BIS2003), Colorado Springs, Co, USA, June 2003, pp 153-161 .

18. M. Paolucci, K. Sycara, T. Nishimura, and N. Srinivasan, "Using DAML-S for P2P Discovery," in Proceedings of the First International Conference on Web Services (ICWS'03), Las Vegas, Nevada, USA, June 2003, pp 203- 207.

19. M. Paolucci, K. Sycara, and T. Kawamura, "Delivering Semantic Web Services," in Proceedings of the Twelfth World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003, pp 111- 118.

20. R. Singh, K. Sycara, T. R. Payne, "Distributed AI, Schedules and the Semantic Web," in the XML Journal, Vol. 03, Number 11.

21. Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara; "Importing the Semantic Web in UDDI". In Proceedings of Web Services, E-business and Semantic Web Workshop

22. Anupriya Ankolekar, Frank Huch and Katia Sycara. "Concurrent Semantics for the Web Services Specification Language DAML-S." In Proceedings of the Fifth International Conference on Coordination Models and Languages, York, UK, April 8-11, 2002.

23. Terry R. Payne, Rahul Singh, and Katia Sycara. "Calendar Agents on the Semantic Web." IEEE Intelligent Systems, Vol. 17(3), pp. 84-86, May/June 2002. Copyright 2002, IEEE Computer Society. Also appears in IEEE Distributed Systems Online, Vol. 3(5), 2002.

24. Terry R. Payne, Rahul Singh, and Katia Sycara. "RCal: A Case Study on Semantic Web Agents." In The First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, 2002.

25. Terry R. Payne, Massimo Paolucci, Rahul Singh, and Katia Sycara. "Facilitating Message Exchange though Middle Agents." In The First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, 2002.

26. Anupriya Ankolekar, Frank Huch, Katia Sycara. "Concurrent Execution Semantics for DAML-S with Subtypes." In The First International Semantic Web Conference (ISWC02), pp318-332, Sardegna, Italy, 2002.

27. Terry R. Payne, Rahul Singh, and Katia Sycara. "Browsing Schedules - An Agent-based approach to navigating the Semantic Web." In The First International Semantic Web Conference (ISWC02), pp 469-473, Sardegna, Italy, 2002.

28. The DAML Services Coalition: Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne and Katia Sycara. "DAML-S: Web Service Description for the Semantic Web." In The First International Semantic Web Conference (ISWC02), pp 348-363, Sardegna, Italy, 2002.

29. Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara; "Semantic Matching of Web Services Capabilities." In Proceedings of the 1st International Semantic Web Conference (ISWC02), pp 333-347, Sardegna, Italy, 2002

30. Terry R. Payne, Massimo Paolucci, Rahul Singh, and Katia Sycara. "Communicating Agents in Open Multi Agent Systems." In First GSFC/JPL Workshop on Radical Agent Concepts (WRAC), Arlington VA, 2002.

Appendix A

The following pages include an indicative selection of relevant publications. All publications can be downloaded from www.cs.cmu.edu/~softagents.

Bringing Semantics to Web Services with OWL-S

DAVID MARTIN
Artificial Intelligence Center, SRI International

MARK BURSTEIN
Intelligent Distributed Computing Department, BBN Technologies

DREW McDERMOTT
Computer Science Department, Yale University

SHEILA McILRAITH
Department of Computer Science, University of Toronto

MASSIMO PAOLUCCI[*]
DoCoMo Communications Laboratories Europe

KATIA SYCARA
Robotics Institute, Carnegie Mellon University

DEBORAH McGUINNESS
Knowledge Systems Laboratory, Stanford University

EVREN SIRIN
University of Maryland, College Park

NAVEEN SRINIVASAN
Robotics Institute, Carnegie Mellon University

Abstract

There is a growing recognition of the need for richer semantic specifications of Web services, so as to enable fuller, more flexible automation of service provision and use; to support the construction of more powerful tools and methodologies; and to promote the use of semantically well-founded reasoning about services. We describe OWL for Services (OWL-S), an ontology of concepts providing a rich vocabulary for describing Web services in this way. We show how OWL-S can be used to automate a variety of service-related activities, focusing particularly on service discovery, interoperation, and composition. We describe a developer-friendly “surface

[*] Work performed while Paolucci was a member of the Robotics Institute, Carnegie Mellon University.

syntax” for OWL-S process representations. In addition, we survey some of the proposed extensions and other varied work that has been based on OWL-S.

Keywords: Web services, Semantic Web, Semantic Web services, OWL, OWL-S, service discovery, service composition

1.Introduction

Work on Semantic Web services [56] lies at the intersection of two important trends in the evolution of the World Wide Web (WWW). The first trend is the rapid development of Web service technologies, which promises to make the Web a place where the role of computation (including processes, transactions, workflows, etc.) is on a par with that of information. The second trend is towards the widespread publication of computer-interpretable structured information, enabling the use of software agents to achieve fuller automation of Web usage. This trend is embodied in emerging Semantic Web technologies (e.g., [11]). We briefly discuss each of these two fields before characterizing the objectives of Semantic Web services in general, and OWL for Services (OWL-S) in particular.

Web services. The promise of broad-based interoperability through widely accepted standards for interactions with Web services is now well recognized. Vigorous efforts are underway to define and evolve such standards. For example, the Web Services Description Language (WSDL) [17], being developed and standardized in the W3C's Web Services Description Working Group, is already well established as an essential building block in the evolving stack of Web service technologies. WSDL provides for the specification of structured input and output messages for services, and other details needed for the invocation of the service. WSDL does not, however, support the specification of workflows composed of basic services. In this area, the Business Process Execution Language for Web Services (BPEL4WS) [4], under development through OASIS, has received the most attention. The W3C's Web Services Choreography Working Group [87] is chartered to explore the related area of specifying valid interaction patterns between Web services. To support finding useful Web services, advertising and discovery mechanisms and standards are being proposed, of which, Universal Description, Discovery and Integration (UDDI) [85] has received the most attention to date. Proposals relating to a number of other aspects of Web services, such as resource management and security, are also being developed at W3C and OASIS.

The driving motivation behind these various Web services standardization efforts is the vision of the enormous benefit to be had by achieving reliable, ubiquitous software interoperability, across platforms, across networks, across organizations. Accordingly, the primary technical focus is on standardizing and validating the syntax and mechanisms of message exchange, so as to support reliable communications and interoperable tools.

The Semantic Web. Where *interoperability* is the motivation for Web services, *automation* of information use is the objective of the Semantic Web. The Semantic Web is a set of technologies that makes it possible for software systems to reason about and integrate Web-accessible information for a wide variety of purposes. At its core, the Semantic Web uses a set of formal languages for representing information content in terms of *concepts and relationships*, defined in *ontologies* that live on the Web in an XML-based format. These concepts are used in representing the content of Web documents, their definition in Web-published ontologies ensures that systems reading those documents can find and interpret the concept definitions referenced. Similarly, ontologies can be used to describe meta-content published on Web sites, such as for describing the kinds of information available in Web-accessible database resources or document repositories.

Semantic Web technologies provide the basis for a next-generation Web that is unambiguously interpretable by computers [11]. A key step in the realization of the Semantic Web has been the development of a suitably rich language (or set of languages) for encoding and describing Web content and meta-content. These languages have a well-defined semantics, and must be sufficiently expressive to describe the complex inter-relationships and constraints found between objects described on the Web. They must also be suitable for automated manipulation and reasoning. A number of languages have been developed, all building on XML. These include RDF, RDF(S), and OWL [53], all of which have been standardized at the World Wide Web Consortium (W3C). OWL, building on the other two, is a Web ontology language based on description logics[1]. It provides a natural way of describing class-subclass relationships, constraints on relationships between classes, and defining many different kinds of relationships between class instances that denote real-world entities. OWL supports the creation of ontologies for any domain, and the use of those ontologies as a vocabulary of types and relations for describing the real-world entities mentioned in specific Web resources. OWL is designed to be compatible with decidable automated reasoning procedures, making it possible to build applications that can automatically determine the logical consequences of Semantic Web-accessible statements.

Semantic Web services. Just as the Semantic Web has focused on automating interactions with information resources, people have recognized the possibility of using this technology to enable semantic specifications of Web services. Adding semantic annotations to Web services will enable the construction of more powerful tools and methodologies for describing services and lead to more flexible automation of service provision and use. A more comprehensive and declarative specification of different aspects of services makes possible the automation of a broad range of activities throughout a Web service's lifecycle. For example, semantic service descriptions can support greater automation of service selection and invocation, can make it possible to automate the translation of message content between heterogeneous interoperating services and clients, can be used to develop automated or semi-automated approaches to service composition, and can result in more comprehensive approaches to service monitoring and automatic recovery from failure. Further down the road, richer semantics can help to automate activities such as verification, simulation, configuration, supply chain management, contracting, and negotiation of service contracts.

The combination of Web service and Semantic Web technologies gives rise to a compelling vision of knowledge-enabled use of computational resources across networks. The Web services stack, as we know it today, is beginning to support the construction of a vast global network of interoperable procedures, transactions, devices, sensors, and queryable information sources -- but their use and management is destined to be labor-intensive and brittle if we continue with current technological approaches. Semantic Web technologies, in a complementary fashion, allow us to add machine-interpretable knowledge, enabling a higher degree of automation of service

[1] OWL includes three sublanguages, of which OWL Lite and OWL DL correspond to description logics; OWL Full does not. Apart from condition, precondition, and effect expressions, OWL-S is expressed using OWL DL.

interactions. This automation will involve automated reasoning, about how to compose services and their products flexibly, forming adaptive workflow processes, rather than brittle ones.

To move towards the realization of this vision, researchers have been developing languages, ontologies, algorithms, and architectures under the heading of *Semantic Web Services* [56]. The authors of this paper, members of the OWL-S Coalition, are developing an OWL ontology for services (OWL-S) [61], which seeks to provide the building blocks for encoding rich semantic service descriptions, by building on the capabilities of the Web Ontology Language, OWL. In the long-term, OWL-S and related efforts aim to lay the foundations for the most effective evolution of Web service-related capabilities that is possible with current and maturing technologies, and to do so in a way that provides a single, comprehensive, integrated representational framework. In the short-term, our goal is to promote the rapid adoption of semantically expressive technologies that are already well understood. As such, we have taken pains to construct mechanisms by which OWL-S can be used along with existing and emerging Web services standards, such as WSDL.

This paper gives an overview of OWL-S, a report on its status, and a summary of a wide range of work based on OWL-S. It reflects the authors' design consensus as of OWL-S version 1.2, which is available on the OWL-S Web site [61]. In addition to the OWL ontology files, the release site includes examples and additional forms of documentation, including, in particular, a comprehensive technical overview, a tutorial "walk-through" of a simple example, additional explanatory material regarding the grounding and the use of profile-based class hierarchies, and information about the status of this work, including unresolved issues and future directions. A subset of this material has been presented to the W3C consortium as a member submission [45].

In what follows, we first discuss the high-level objectives of OWL-S (Section 2), and then motivate OWL-S in terms of some simple usage scenarios (Section 3). In Section 4, we describe the structure and content of the service ontology, including its sub-ontologies for profiles, processes, and groundings. (We treat the process modeling sub-ontology at greater length than the other two, because this article introduces the "surface syntax" for process modeling.) Section 5 presents some architectural elements that may be used with OWL-S, and Section 6 focuses on OWL-S-based approaches to service discovery and service composition. In Section 7, we survey the broad scope of tools as well as research and prototype systems that have been developed based on OWL-S. In Section 8, we discuss related efforts to model aspects of Web services. The paper ends with a brief summary and discussion of future directions.

2.Objectives of OWL-S

The high-level objectives of OWL-S include the following:

- *Provide a* ***general-purpose representational framework*** *in which to describe Web services.*
- *Support* ***automation*** *of service management and use by software agents.*
- *Build on existing* ***Web services standards****.*
- *Build on existing* ***Semantic Web standards****.*
- *Provide a* ***comprehensive approach*** *supporting the entire "lifecycle" of service tasks.*

We emphasize that these are *objectives*; that is, while each of them has been addressed to some degree, we do not claim that they have all been fully achieved. This and subsequent sections discuss some of the ways in which they have been addressed, and the Summary section briefly describes future directions that can lead to a more complete realization of them. Note that these objectives are shared by most other Semantic Web service research efforts. In the remainder of this section, we elaborate on the objectives as they relate to OWL-S.

General-purpose representational framework. OWL-S can be described as both a language and an ontology for Web service descriptions. It is a language in the sense that it defines terms (e.g., `Process`, `parameterType`, `If-Then-Else`) that provide a way to describe critical aspects of Web services. These terms reside in a small set of ontologies defined in OWL. Descriptions formed using these ontologies in conjunction with other domain ontologies are normally expressed in the XML-based syntax for OWL/RDF for purposes of supporting their publication and automated use. OWL-S also provides an alternative syntax for process specifications, which is presented in Section 0. This *surface syntax* provides greater readability and maintainability for developers [50] but can be directly mapped into the XML syntax for OWL-S and OWL descriptions.

OWL-S is designed for extensibility. In particular, OWL class, subclass and property definitions can be used to extend the OWL-S ontologies such as the Profile (presented in Section 0) For example, new service categories and characteristics can easily be defined to refine the OWL-S profile-based descriptions used for advertising services in a particular domain. Similarly, the grounding sub-ontology (presented in Section 0) is modular and can be replaced with alternative grounding specifications for translating OWL-S processes into message transport layers not based on WSDL.

As an ontology, OWL-S does not make many specific architectural or implementation commitments. While OWL-S is designed primarily for use in the context of service-oriented architectures, enabling software systems to discover Web services and interact with them directly, OWL-S can also be used in other contexts. For example, the language can be used as the basis for communications in centralized broker architectures, where agents specify objectives to a broker, which then dispatches the requests to the most appropriate services (e.g., [68]). Similarly, it has been shown that OWL-S can be used in architectures where mediators translate between different agent interaction languages or mediate interaction protocols between different types of service applications (e.g., [69]).

Automation. OWL-S is designed to enable the automation of a wide variety of activities related to services. A primary motivation is to enable software agents to use services without any 'built-in' pre-existing knowledge of their properties, capabilities, and protocols. Full automation of service activities in its complete generality is an enormous challenge and may be neither achievable nor desirable. There are a variety of reasons why humans may need or wish to remain "in the loop" in enacting processing on the Web. There are numerous compelling use cases that point to a spectrum of needs for automation or semi-automation of Web service integration activities. The most active investigations of automation have to do with the tasks of service discovery, selection, interaction, and composition, but a compelling case can also be made for automation support to avoid reprogramming when service capabilities and protocols change over time. In later sections we discuss work related to many of these tasks.

Build on existing Web services standards. OWL-S is not intended to replace existing Web services standards, but to augment them with an explicit mechanism for specifying in detail the interpretation of service processes and capabilities, and the semantics of the terms that are used to describe services at that level of detail. OWL-S leverages the Semantic Web and OWL to do this, providing an ontology of concepts that define service structure, behavior, and characteristics. It does not define domain-specific concepts (such as books, or book titles for a book vending service), but leaves it to service providers and communities of interest to define the OWL domain concepts that are needed to specify the domain-specific aspects of particular services.

As discussed in Section 0, the OWL-S grounding explicitly relates the different parts of WSDL messages to OWL concepts, as they are used in connection with inputs and outputs in OWL-S service representations. This mapping to WSDL allows the details of OWL-S service invocations to be specified in WSDL, and allows the use of OWL-S to abstractly represent Web services that do not use OWL directly in their implementation.

As discussed in Section 0, OWL-S has also been mapped to the UDDI framework [65]. This mapping enables the use of UDDI as a registry for the discovery of OWL-S descriptions of Web services. In this way, OWL-S complements UDDI, providing a language for representing Web service capabilities, and algorithms for the matching and discovery of Web services, while relying on the registry infrastructure provided by UDDI.

Build on existing Semantic Web standards. OWL-S builds on existing Semantic Web standards, including not only OWL and RDF, but other emerging standards such as the Semantic Web query language, SPARQL [71], the rule language, SWRL [39], and SWRL-FOL [70][2].

OWL concepts are used to represent the types of service inputs and outputs and may be mentioned in preconditions and effects[3] of services. Such concepts are typically domain-specific, and can draw on Semantic Web ontologies developed for by communities of interest for a diversity of purposes. OWL is also the language in which OWL-S concepts themselves are specified.

Languages such as SWRL and SPARQL may be used to represent the preconditions and effects of processes. These languages are used to overcome limitations of OWL for describing complex constraints on temporal activities, such as those that occur in rules and queries. Whereas OWL is effective for describing taxonomic categories and properties of things, the structures found in SWRL and SPARQL are better suited to describing conditions requiring the use of variables. OWL-S' use of these languages enables more complete characterizations of services. It should be noted, however, that there is not as yet a consensus standard for unifying these semantic frameworks together as OWL-S uses them; thus, a hybrid reasoning approach is needed.

Comprehensive approach. Research on Semantic Web services to date has focused primarily on automation of Web service advertising, discovery, selection, interaction, and composition, and that is also true regarding work based on OWL-S. Nevertheless, there is an even larger lifecycle of tasks related to service development, deployment, management, and use. For example, service development activities may include *simulation* and *verification* (e.g., [5], [59]) and various forms of tool support. Service use may include contracting, negotiation, monitoring and recovery [15]. While OWL-S does not at present directly address these tasks, it provides extensibility mechanisms and basic building blocks by which additional concepts can be added to support them. The extensions supporting security and privacy issues, as described in [22,28,86,35], offer good examples of how this may be done.

[2] Other expressive logical languages may also be used, such as KIF and DRS. We focus more on SWRL, SWRL-FOL, and SPARQL because of their close relationship with RDF and OWL and their visibility as possible Semantic Web language standardization candidates.

[3] As explained in Section 4.2, the specification of a service's *effects* is included in OWL-S' *results* construct.

3.Example scenarios

Here, we provide several simple scenarios to illustrate some of the functionality provided by OWL-S.

Semantic discovery. Service discovery is the task of finding an appropriate service, given a description of the properties of the service that the requester is seeking. Most existing practice relies on human perusal, of service descriptions provided as keywords and text strings, supported by text string matching algorithms that can filter the candidates if given the right keywords to use. This process is labor-intensive, requiring too much user involvement at runtime, and is hard to accommodate at development time without knowing the precise set of keywords to utilize. When service descriptions are based on bare-bones taxonomies, keywords, and/or English descriptions, the forms of requests and the precision of matching techniques are severely limited. When service providers and requesters use different terminologies to describe services, string-based search will frequently fail to discover appropriate services. On the other hand, if service descriptions are based on shared ontologies on the Semantic Web, the hierarchical class structuring and associated reasoning and mediation techniques available with OWL can be applied to produce more flexible and effective service discovery with less need for direct user involvement at runtime.

As a simple illustrating example, consider a requester who would like to buy ‘cutlery’. Let us assume that there are service providers that categorize themselves as sellers of kitchenware (including ‘knives’), but none that advertise ‘cutlery’. With string-based discovery, the requester will not locate any service providers that match the specified objective. (Of course some trial-and-error by a human would likely lead to a match, but here we are aiming to support software agents and tools as requesters.) OWL-S relies on OWL ontologies to describe services, including domain-specific terms. As a result it can overcome the limitations of string-based discovery by providing semantic matching based on widely shared domain ontologies. In this example, even if there were no providers advertising the domain class of cutlery, a requester using OWL-S could use ontological knowledge to request providers of kitchenware, as cutlery would be known to be a subclass of kitchenware.

Ontology-based approaches allow the request to be a description at a different level of abstraction than the descriptions of the service providers (as the matcher can reason about the relationship between the classes), while still allowing the requester to specify important details that might eliminate some subclasses that would otherwise match. For example, the request might include the detail that the provider needed to be located in a particular state, or sell the item at or below a particular price. The application of OWL-S to semantic matching algorithms is discussed in Section 6.1.

Composition. Web service composition is the task of combining independently developed and supported Web services to achieve some user objective. Composition can be achieved manually by specifying a workflow and associated dataflow between the services; it can be done automatically by a computer program; or it can be done by adapting existing workflows interactively for a particular user and task. In any of these cases, the agent performing the composition needs to know the conditions under which the chosen Web services can be executed and what kinds of effects their execution will have on the agent’s state of knowledge and on the state of the world. Key to any automated composition is the availability of software-interpretable descriptions of service capabilities and interaction requirements, and a computer program capable of interpreting and reasoning about them. In addition to the semantic characterization of inputs and outputs, effective composition tools need to consider the preconditions required for successful use of each service, and the effects they will have in the world. "Nonfunctional" attributes of service behavior, such as resource requirements and quality-of-service guarantees, may also be needed.

As an example, consider the scenario described in [11], which describes two people trying to take their mother to a physician for a series of treatments and follow-up meetings. The problem is to come up with a sequence of appointments that will fit everyone's schedules, preferences and constraints. Creating a composition involves discovering new services based on various requirements, e.g. finding all hospitals which provide the treatment and are approved by the health insurance company, coordinating data between heterogeneous services, e.g. personal schedule management software and hospital appointment services, and reasoning about the effects of services without executing those services, e.g. can I schedule all my hospital appointments if I go to the doctor at this date. Finding compatible services and automating the sequencing and data flow between these services can be done most effectively when the input and output parameter types are described as terms in a way that lets reasoning software determine that the transfer is semantically appropriate. This requirement can be met by using an ontology system such as OWL, and a means of describing service structure (preconditions and effects) and function (activity type), such as that provided by OWL-S. Automated composition using OWL-S is discussed in Section 6.2.

Semantic data integration and service interoperation. Semantic data integration is the task of integrating data according to its meaning. It is a pervasive challenge in the Web. In the context of Web services, semantic data integration is critical to service interoperation. In particular, composing Web services requires matching the output(s) of one service to the input(s) of another, and/or matching the effect(s) of one service to the precondition(s) of another. Matching based on syntactic descriptions or natural language text descriptions of these properties is not adequate. The same string term (or document type) may be used to describe different things, and different terms (document types) may be used to refer to the same thing. Data may need to be manipulated or transformed to enable it to serve as input to a subsequent Web service. These transformations must be meaning preserving, and services may need to be provided to effect these transformations. Existing mechanisms for describing Web services do not provide rich enough descriptive mechanisms to automate semantic discovery, integration, or the kinds of semantics-preserving translations needed for dynamic interoperation.

For example, consider a loan scenario in which online lender services require electronic transmission of credit report scores. Assume a particular case where the applicant has moved from the UK to the US and only has a credit report from the UK, which uses a different scoring system, while the lending services require US credit scores. An automated composition system might supply the UK score when asked for a credit score, because both are numbers labeled as credit scores, but the applicant might be denied a loan because the credit score was too low or was invalid because it was too high. If however, a credit translator service is capable of taking a UK credit score and translating it to generate a US credit score, then the pre-requisites for the lender service can be met and the buyer may obtain a loan. To realize this semantic integration requires explicit representations of the requirements and capabilities of services in a machine interpretable form. The Semantic Discovery Service, a prototype system that performs semantic integration and interoperation in the context of dynamic customization of existing Web service compositions, is briefly discussed in Section 6.2.

Pervasive computing. As the devices we use in our everyday life, like televisions and music players in our home or projectors and printers in our office, become more sophisticated and net-

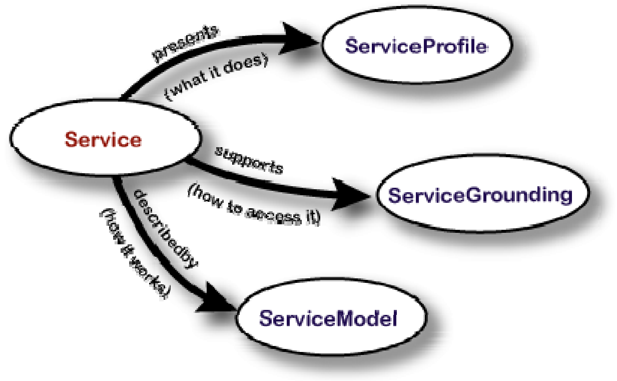


Figure 2. ***Top level of the service ontology.***

accessible, we gain the ability to automate aspects of their usage. The convergence between device and Web service standards (e.g., both WSDL-based Web service descriptions and UPnP-based device descriptions rely heavily on SOAP as the messaging protocol) makes it possible to describe the functionalities of devices as Web services.

As an example, consider a business meeting where one or more people are doing presentations. In such a setting, one might want to do several different tasks that will require interacting with the devices and other people in the environment. For example, such tasks might include finding out the e-mail addresses of everyone in the conference room and e-mailing the presentation document, printing out the presentation handouts, adjusting screen resolution based on the capabilities of the projector, and controlling the lighting in the room when the presentation begins and ends. Realizing such scenarios in the real world not only requires description of the capabilities of the devices, but also requires description of security and privacy policies. Some assurance is needed that the service providing contact information will only disclose such information to appropriate requesters. Similarly, the printer or the projector service should not let anyone use those devices arbitrarily. The discovery and matchmaking functionality provided by OWL-S facilitates composing services in a way that meets these kinds of constraints as they occur in pervasive computing environments. An example of a real-world system using OWL-S in this way is the Fujitsu Task Computing Environment [47]. The extensible nature of the OWL-S description language enables the individual devices in that environment to be described and also addresses critical security and privacy concerns (using OWL-S extensions such as those proposed in [22,28,86,35]).

4.Structure and content of OWL-S

The OWL-S ontology includes three primary sub-ontologies: the service profile, process model, and grounding. Roughly speaking, the profile is used to describe *what the service does*, the process model is used to describe *how the service is used*, and the grounding is used to describe *how to interact with the service*. The profile and process model are thought of as *abstract* characterizations of a service, whereas the grounding makes it possible to interact with a service by providing the necessary *concrete* details related to message format, transport protocol, etc. Figure 1 shows the relationships between the top-level classes of the ontology.

Each service described using OWL-S is represented by an instance of the OWL class *Service*, which has properties that associate it with a process model, groundings and profiles for the service described. The process model provides the complete, canonical description of how the service behaves, and the grounding supplies the details needed to interact with that process model. Each profile may be thought of as a summary of salient aspects of the process model plus additional information that is suitable for purposes of advertising and selection. The definition of the Service class allows for multiple groundings so they can be added dynamically, and alternative groundings can be provided. It allows for multiple profiles so as to allow for tailoring of

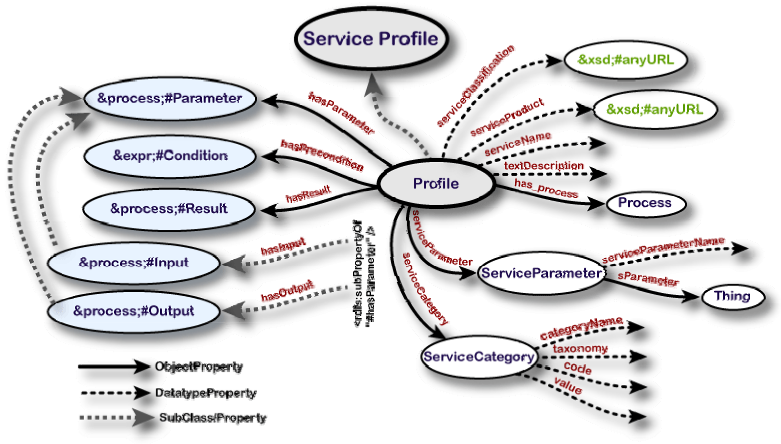


Figure 2. ***The structure of the OWL-S profile.***

advertisements for different contexts.

These notions are explained more thoroughly in the following subsections. Additional details may be found in the Technical Overview at [61].

4.1 The service profile

The OWL-S profile provides a set of concepts to specify capabilities of services, with the goal of supporting capability-based discovery. Specifically, the OWL-S profile allows service providers to advertise what their services do, and service requesters to specify what capabilities they expect from the services they need to use. Crucially, the profile provides an explicit description of those capabilities, so that they do not have to be extracted from incidental properties such as the name of the service, or the company that provides it. By exploiting the structure of OWL-S profiles and their references to OWL concepts, a discovery process can find those services that are most likely to satisfy the needs of a requester.

The overall structure of the OWL-S profile is shown in Figure 2, in which ovals represent OWL classes and arcs represent OWL properties. A profile is an instance of the class `Profile` (or a subclass of `Profile`). The principle elements that occur in a profile include the *profile type*, which is the particular subclass of Profile that's being instantiated, and the *inputs, outputs, preconditions*, and *results* (which are normally either identical to, or a subset of, those same elements in the process model). In addition, a profile may include a *product type*, where appropriate; *service categories* that may be used to refer to existing business taxonomies that may not be codified in OWL, such as NAICS; and a variety of *service parameters* that may be used to specify additional features of the service. Subclasses of `Profile` may be created for particular domains or types of services, and specialized with appropriate properties. For example, for shipping services, there might be a `ShippingServiceProfile` subclass with the additional property `geographicRegionServed`, with `GeographicRegion` (from an appropriate online ontology) as its range.

Broadly speaking, the OWL-S profile describes three aspects of a service. The first one is the *functional* aspect that is represented using input, output, precondition and result properties. The functional aspect represents the information transformation computed by the service from the inputs that the service expects to the outputs it generates, and the transformation in the domain from a set of preconditions that need to be satisfied for the service to execute correctly, to the effects that are produced during the execution of the service. A purchase in an e-store illustrates a typical set of these transformations: the inputs are the name of the desired product and a credit card number, the output is a receipt of purchase, the preconditions specify that that the credit card is valid, and the effects specify that the credit card will the charged, and that the goods will change ownership.

The second aspect, the *classification* aspect, supports the description of the type of service as specified in a taxonomy of businesses (or other suitable domain-specific ontologies). The classification of the service is indicated by the profile type and/or the service categories. Using these elements, the provider can specify explicitly what type of service it provides, and the requester can specify the type of service with which it would like to interact.

The third aspect of the profile is the *non-functional* aspect, which makes distinctions between services that do the same thing but in different ways or with different performance characteristics. Examples of non-functional aspects of a service include the service security and privacy requirements [34], the precision and timeliness (quality of service) of the service provided, and the cost structure. Since it is impossible to provide a complete set of attributes for the representation of service parameters (many of which are domain-specific), the solution adopted by OWL-S is to provide an extensible mechanism that allows the providers and requesters to define the service parameters they need.

A partial example of a profile for a fictitious airline booking service, expressed in the RDF/XML syntax of OWL, is shown in Figure 3[4]. The profile, an instance of

[4] For clarity of presentation and to save space we have simplified the syntax of each input and output in Figure 2. For example, instead of

```
   <hasInput                                   rdf:resource="http://fly.com/Onto#Dep_Airport"/>
```

it should be

```
    <hasInput>

        <process:Input>

          <process:parameterType rdf:datatype="&xsd;#anyURI”>
```

FlightReservationProfile (a subclass of Profile), describes inputs specifying a departure and arrival airport, and a reservation as output. The crucial aspect of this example is that the values used to specify the types of the OWL-S properties are the URIs of concepts defined in some ontology. In this example, the ontology used is the fictitious ontology http://fly.com/Onto. The advantage of using concepts from Web-addressable ontologies, rather than XML schemata, is that the descriptions include clear and unambiguous references to the definitions of the terms used. This, together with the fact that these terms are arranged in a semantic abstraction hierarchy, enables the discovery process to avoid matching failures due to syntactic differences and enable providers and requesters to use different terminology within the space of available ontologies and still find appropriate services as long as the different terms are suitably related by their definitions.

```
<FlightReservationProfile rdf:ID="BravoAir">
   <serviceName>Bravo Air</serviceName>
   <contactInformation rdf:resource="#BAco"/>
   <hasInput rdf:resource="http://fly.com/Onto#Dep_Airport"/>
   <hasInput rdf:resource="http://fly.com/Onto#Arr_Airport"/>
   <hasOutput rdf:resource="http://fly.com/Onto#Reservation"/>
   ……
</FlightReservationProfile>
```

Figure 3. ***A simple, partial OWL-S profile.***

4.2 The process model

Once a Web agent has identified a service as likely to be relevant to its goals, it needs a detailed model of the service to determine whether the service can do the job, and, if so, what constraints must be satisfied and what pattern of interactions will be required to get it to do that job.

An OWL-S process specifies, among other things, the possible patterns of interaction with a Web service. There are two sorts of processes that can be invoked: atomic and composite. An *atomic* process is one that has no internal structure. It corresponds to a single interchange of inputs and outputs between an agent and the service. A *composite* process consists of a set of component processes linked together by control flow and data flow structures. The control flow is described using typical programming language or workflow constructs such as sequences, conditional branches, parallel branches, and loops. Data flow is the description of how information is acquired and used in subsequent steps in this process. A third type of process, the *simple* process, can be used to provide abstracted, non-invocable views of atomic or composite processes. Simple processes are not discussed further in this paper; the reader is referred to the Technical Overview at [61] for information about them.

```
          http://fly.com/Onto#Dep_Airport
        </process:parameterType>

    </process:Input>

</hasInput>
```

The concepts used to describe a process are themselves terms in the OWL-S ontology. The OWL-S definition of Process has a set of associated features (inputs, outputs, preconditions, results) linked to the Process concept by OWL properties (`hasInput`, `hasOutput`, etc.). Composite processes are related to their top-level control structure using the `composedOf` property, and other properties, such as `components`, are used to show the relationships between control structures. Other properties relate the process model to corresponding messages (referenced by the grounding), and to advertised capabilities descriptions in the corresponding OWL-S profile. Figure 4 shows some of the principal elements of the process model.

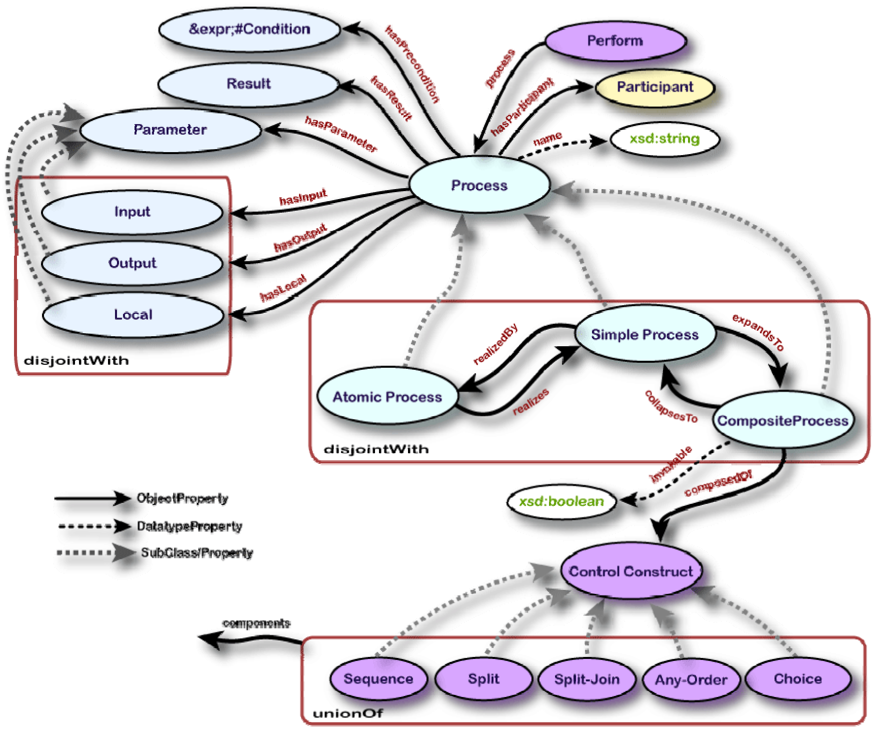


Figure 4. ***The structure of the OWL-S process model.***

OWL-S Process Model Surface Syntax

Machine interpretation of OWL-S process models relies on systems that read the process definitions expressed in OWL's RDF/XML serialization syntax, which is not designed for human consumption. Even simple processes can become swamped in reefs of angle brackets that make it hard for people to read and understand what each process does. For this reason, we have provided a *surface syntax* (or "presentation syntax") that makes OWL-S processes look like programs in a high-level language. This more procedural surface syntax for OWL-S is easily translatable into the canonical RDF/XML syntax. Our exposition style here is to make the "procedural" notation primary, explaining the translations of its terms as we go.

For example, namespaces are a crucial feature of most XML applications, and OWL-S, which directly utilizes OWL, is no exception. In the surface syntax we use the notation

```
with_namespaces
 (uri"http://www.daml.org/services/owl-s/1.2/Amazon.owl",
  service:  uri"http://www.daml.org/services/owl-s/1.2/Service.owl",
  owlproc:  uri"http://www.daml.org/services/owl-s/1.2/Process.owl",
  profile:  uri"http://www.daml.org/services/owl-s/1.2/Profile.owl",
  rdf:      uri"http://www.w3.org/1999/02/22-rdf-syntax-ns")
{
   define atomic process ExpressCongoBuy ...
   define ...
}
```

In the RDF/XML version, these declarations get attached to the outermost RDF element in the usual way:

```
<rdf:RDF
  xmlns="http://www.daml.org/services/owl-s/1.2/Amazon.owl"
  xmlns:service
    ="http://www.daml.org/services/owl-s/1.2/Service.owl"
  xmlns:owlproc
    ="http://www.daml.org/services/owl-s/1.2/Process.owl"
  xmlns:profile
    ="http://www.daml.org/services/owl-s/1.2/Profile.owl"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <process:AtomicProcess rdf:ID="ExpressCongoBuy" ...>
  ...
  </process:AtomicProcess>
  < ...>
  </...>
</rdf:RDF>
```

The with-namespaces expression can be used at any level, and it always gets translated into xmlns declarations, attached to whatever RDF description is most convenient. Once a namespace has been declared, we can then use colons according to the XML convention. However, there are fewer places in the surface syntax where names qualified by namespace prefixes (so-called *qnames*) are needed. For instance, reserved words of the notation are always assumed to be in the OWL-S namespace.

Specifying Process Models

One way to use a process model is as a description of what a client must do in order to cause a service, or a collection of services, to do something. All processes have inputs, outputs, preconditions, and results. The *inputs* are the objects the process works on. The *preconditions* are logical statements that must be true in order for the process to be meaningful and effective, and its terms are specified in `predicate` *`(arg1, .., argn)`* form. For example, a process by which an agent *Y* can buy an object *X* has inputs to identify *X* and a credit card belonging to *Y;* its preconditions might include the requirement that the credit card is valid for purchases. Here is a more specific example, from the "CongoBuy" scenario, an artificial example we developed to represent a book buying service. The complete service is described in [61].

```
with_namespaces
 (uri"http://www.daml.org/services/owl-s/1.2/examples/CongoBuy.owl",
  process:  uri"http://www.daml.org/services/owl-s/1.2/Process.owl",
  books:    uri
    "http://www.daml.org/services/owl-s/1.2/examples/BookConcepts.owl",
  . . .

  rdf:      uri"http://www.w3.org/1999/02/22-rdf-syntax-ns")
{
 (define atomic process ExpressCongoBuy
     (inputs: (ISBN - books:ISBN
                SignInInfo - SignInData
                CC-Number - xsd:decimal
                CC-Type - CreditCardType
                CC-ExpirationDate - xsd:YearMonth),
      locals: (AcctID - CustomerAcctID
                CardUsed - CreditCard),
      precondition:
              (hasAcctID(SignInInfo, AcctID)
                & credit-card-number(CardUsed, CC-Number)
                & validity(CardUsed, valid))
      outputs: ...
      ...);
…}
```

Atomic processes are defined entirely by the values they handle and the effects they have. The former are classified as *inputs*, *locals*, and *outputs.* The latter are classified as *preconditions* and *results.* All these fields are optional. The `inputs` specify the names and types of the values passed to the process. Dashes are used to separate variable names from their conceptual types, which should be OWL concepts. For example, the input parameter ISBN is of type ‘books:ISBN’, which is a concept defined in the namespace ‘books’, an OWL domain ontology. The same convention applies to output declarations. Preconditions are representations of logical formulas with ‘&’ indicating conjunction. Local variables are those bound by occurring in preconditions rather than by being passed to the process explicitly.

The results of a process are specifications of its possible outcomes. Each potential result may be gated by an *output condition*. The distinction between preconditions and output conditions is that preconditions are things that the client can check before deciding to request the service, whereas output conditions are things that are checked by the service in the course of execution that impact the kind of result produced. Results consist of effect and output specifications. Effects are simply propositions which become true, while the information conveyed to output parameters is indicated by the special notation:

```
output( <output-parameter> <= <rdf-description> ).
```

The notation `p |-> r` (read "when *p*, then *r*") means that if *p* is true in the situation just before the process is performed, then it will have result *r*, where *r* consists of a set of outputs and effects. (It doesn’t matter how much time elapses between the situation before and the situation after.) Note that from the perspective of a client reading this process description, and reasoning about it when interacting with the service, what it will observe is the output message that was produced for some particular result condition. At that point, it could infer, based on the process description, that a particular result condition *p* had occurred, and that the listed effects of the process for that result condition should be true.

Our Congo example might have the following outputs and conditional results:

```
define atomic process ExpressCongoBuy
     (…,
      precondition:
              (hasAcctID(SignInInfo, AcctID)
                & credit-card-number(CardUsed, CC-Number)
                & validity(CardUsed, valid))
      outputs (Status - CongoBuyOutputType),
      result:
        (forall (book - books:Book)
         (hasISBN(book, ISBN) & inStockBook(book)
            |->  output(Status <= OrderShippedAcknowledgement)
               & shippedTo(addressOf(AcctID), book))
          &
         (hasISBN(book, ISBN)  & ~inStockBook(book)
            |->  output(Status <= NotifyOutOfStockBook(ISBN)))))
```

Here, two result conditions are described, one where the book is in stock, and one where it is not (~`inStockBook(book)`). In the first case, the output `orderShippedAcknowledgement` is produced, which is our surface-syntax version of an RDF expression representing the meaning of the associated WSDL output message. When the client sees this message, it is licensed to infer that the associated `shippedTo` expression is true, and also that the condition `inStockBook` was true when the service executed. In the other case, a different message would have been sent by the service, whose meaning is captured by the OWL concept `NotifyOutOfStockBook`. The XML representation of this process expresses all of this information in the form of an RDF semantic description, starting with the process itself:

```
<process:AtomicProcess rdf:ID="ExpressCongoBuy">
  <process:hasInput>
```

```
    <process:Input rdf:ID="ExpressCongoBuyBookISBN">
      <process:parameterType rdf:datatype="&xsd;#anyURI">
         &profileHierarchy;#ISBN
      </process:parameterType>
    </process:Input>
  ...
  <process:hasPrecondition>
     ...
  </process:hasPrecondition>
  ...
<process:AtomicProcess>
```

In the XML version, instead of results being produced by a recursive grammar, each result is a separate object, with associated when-conditions indicated using the `inCondition` property. For reasons that will become clear in a moment, we omit the details of these properties, but for the XML equivalent of `output` we can be a bit more complete:

```
<process:AtomicProcess rdf:ID="ExpressCongoBuy">
   ...
  <process:hasResult>
    <process:Result>

      <process:inCondition>
           ...
      </process:inCondition>
      <process:hasEffect>
           ...
      </process:hasEffect>
      <process:withOutput>
        <process:OutputBinding>
          <process:toParam rdf:resource="#ExpressCongoBuyStatus"/>
          <process:valueData
            rdf:resource="#OrderShippedAcknowledgement"/>
        </process:OutputBinding>
      </process:withOutput>
    </process:Result>
  </process:hasResult>
</process:AtomicProcess>
```

This is considerably more verbose than `Status <= OrderShippedAcknowledgement`, but the reduction of the process notation to OWL descriptions allows us to link processes to other entities, as we see in Section 0. It also allows process descriptions to be serialized and read as standard OWL/RDF/XML. We will use the *param* <= *val* notation again later; we call it a *parameter binding*.

A couple of further observations about the process notation: In the surface syntax, we can rely on standard variable-scope conventions to avoid confusion even if the same name is used in more than one context. When we translate the surface syntax into RDF, we prepend the name of the process to all variables when generating IDs. That's why the variable `Status` becomes `ExpressCongoBuyStatus` in the RDF. (This transformation is performed by a special-purpose OWL-S-to-RDF translation algorithm, so there is no need to indicate the name change in the process notation.)

Second, we have left blanks in the RDF wherever the surface equivalent is a logical formula. At the beginning of the OWL-S project, we left such blanks because there was no widely accepted approach to expressing formulas in a way compatible with RDF. Now we leave them blank because there are many alternatives to choose from, including N3 [10], RuleML [74], SWRL [39], SWRL-FOL [70], and SPARQL [71], and a widening acceptance of the idea that formulas can be embedded as (string or quoted) constants without bothering to express them as RDF-like descriptions of anything. In this paper we will avoid the details of formula translations, because it is pretty clear that no matter how much the formulas can be made to *look* like RDF, the resulting descriptions aren't of much use to OWL-based reasoners. The translation of descriptions in the other direction is meaningful; RDF triples are easily translated into logical formulas.

We should stress that we employ RDF encodings such as SWRL “syntactically”: we view them as devices for encoding first-order logic, even if they originally carried more semantic baggage. For instance, SWRL was originally a syntax for Prolog-like rules, whose free variables had a stipulated interpretation. When it is used to express preconditions and effects in OWL-S, the free variables are understood to be bound by binding mechanisms native to OWL-S, such as input and local declarations.

This brings us to the key question of what sorts of inference we expect the OWL-S process models to support. Some of these questions depend on the interpretation of composite processes, which we will discuss shortly. But there are plenty of tasks that can be supported using purely atomic models of processes. Given a set of goals and a set of process specifications, one can find or check a described sequence of process instances to determine if it can accomplish those goals. Finding one is the classic *AI planning* problem, which in this context is called *dynamic service composition*. A variant of dynamic service composition is what we term *nondeterminism reduction:* Given a partially specified structure of atomic processes and state constraints, find an execution sequence consistent with that structure. The structure can be viewed as advice, or in some instances as underspecified scripts of how a Web service composition is to be realized. Yet another form of inference we expect OWL-S to support is *plan recognition:* given one or more actions by an agent, infer what goals the agent might have.

The possibilities for reasoning about processes broaden considerably when we allow processes to have internal structure. Such processes are called *composite processes*:

```
define composite process P
  (inputs: (…)
   outputs: (…)
   preconditions: (…)
   results: (…)
   )
 {
    ---body---
 }
```

In the simplest case, the *body* of a composite process consists of a *sequence* of actions, separated by semicolons. The simplest sort of action is to execute another process. This kind of act is called a `perform`.

```
     define composite process P(inputs-outputs-preconditions-results)
        {
           perform Q(V <= E, ...);
           perform R(X <= F,...);
                ...
        }
```

This notation indicates that the first step of the composite process *P* is to perform the process *Q* (which may or may not be atomic) with some specific parameters. Input parameter *V* of process *Q* should get value *E*, and so forth. Inputs to these steps may be the same as or derived from an input to the composite process, they may be constant values, or, for steps other than the first, they may be the result of an earlier step in the process. Here is a very simple composite process in our surface syntax. The XML/RDF serialization of this simple process would take nearly a full page to show, but see the OWL-S Website [61] for examples.

```
define composite process BuyBook
    (inputs: (ISBN - books:ISBN, user – access:UserID)
 { perform Login(UID <= user);
   perform requestBuyBook(BookID <= ISBN) }
```

If a planning system is given a set of composite processes describing standard ways for interacting with Web services, it can generate a set of instances of those processes that will accomplish its objectives. This paradigm is called *hierarchical planning;* we discuss related work in this area in section 0. In solving the problem of *nondeterminism reduction*, an inference engine verifies that in a particular circumstance, there is a set of choices of actions and their orderings that will allow a plan with underdetermined structure to be executed successfully. Nondeterminism reduction is often more efficient than planning, because all the possibilities are laid out in advance and the system must merely narrow them to the point that all remaining execution traces result in a situation satisfying the goal. There are a couple OWL-S control constructs for expressing nondeterminism. One is the *choice* construct, which appears in the surface syntax as:

```
P1 ;? P2 ;? … ;? Pk
```

and is executed properly when one of the `Pj` is executed. Nondeterminism reduction is often more efficient than planning, because the search space, i.e. the number of process choices the planner must make, is significantly reduced. Another sort of nondeterminism is embodied in the *any order* construct:

```
P1 ||; P2 ||; … ||; Pk
```

which means that all of the `Pj` are to be executed in *some* order. Other constructs in OWL-S include conditional execution:

```
If E then A else B
```

Split-join:

```
P1 ||> P2 ||>  … ||> Pk
```

And split:

```
P1 ||< P2 ||<  … ||< Pk
```

The last two indicate parallel (interleaved) execution of the `Pj`. The difference between them is that “split-join” is finished when all the `Pj` finish, where “split” terminates immediately, after the subprocess instances are spawned.

Describing Data-flow in OWL-S

Another key issue in designing the process language was to allow for flow of data between steps in composite processes, and between inputs and outputs of a single process. One standard way to do that is to add ordinary variables to the language, and allow them to be set and read. The problem with this approach is that it is too general, and the semantics become unclear when different processes within a composite process are executed by different servers; variables referenced by multiple different subprocesses implies that physical data flow actions are to occur. Combined with the possibility of parallel activities the language could generate complex race and lock conditions. To avoid this while allowing for a variety of simpler cases, we introduced explicit data flow expressions into the language. (This is one place where OWL-S differs from the BPEL process notation.) The simplest case is one in which a process step produces a datum that is fed to a later step. We indicate this using the "<=" notation: *p*<= *q* means that parameter *p* gets its value from *q*, a value designator. We introduce the notion of a *step tag* in the surface syntax to indicate which step that value comes from. Any step of a compound process can have a step label joined to it using a double colon, that is *tag*::*step*. If the step is a process with output parameter *y*, then '*tag*.*y*' refers to the value of *y* produced as the output of that particular occurrence of *step*.

For example, the notation

```
{ a:: perform A() ||>  b:: perform B() } ;
 perform C(u <= a.x, v <= b.x)
```

describes a sequence consisting of the parallel performance of processes A and B (with no inputs), followed by a performance of process C, using the x output of A as the u input of C, and the x output of B as the v input of C. Our notation always describes a data flow from the point of view of the data "consumer" (C, in this case), indicating where its inputs come from. This is called the *demand-pull* convention. We have already used this notation in describing the results of a process, which can contain `output` declarations of the form *p*<=*e* that specify the output of the current process as a function of its inputs and local variables. Our hope is that indicating data flows in a more explicit way than traditional variables allow will makes inferences about processes simpler.

Making data flow transparent for iterative processes is an even more difficult problem. The problem with iterations is that data must flow from one iteration to the next. We plan to address this issue in future OWL-S releases by allowing variable declarations in iterative processes, analogously to the way we declare inputs to a process. For each loop variable, the process definition would specify both its initial value and how it is altered on each iteration. So far all we have done is provided some rudimentary notations for loops. Adding the loop variable declarations is a high priority future activity.

4.3 The grounding

The grounding ontology of OWL-S is used to specify how the abstract information exchanges that are detailed by atomic process descriptions are realized by concrete information exchanges between the agent requesting a service and the deployed Web service it has chosen to use. A grounding maps each OWL-S atomic process (in a collection of processes associated with a service) to a WSDL operation (defining input and output messages), and relates each OWL-S process input and output to elements of the XML serialization of the input and output messages of that operation. The purpose of this mapping is to enable the translation of semantic inputs generated by an agent into the appropriate WSDL messages for transmission to the service, and the translation of service output messages back into appropriate semantic descriptions (i.e., OWL descriptions) for interpretation by the agent.[5] It should be noted that a grounding can be supplied at runtime, so as to allow for dynamic binding.

We emphasize that an OWL-S/WSDL grounding involves a complementary use of the two languages in a way that is in accord with the intentions of the authors of WSDL. Both languages are required for the full specification of a grounding, because the two languages do not cover the same conceptual space. WSDL, by default, specifies message types using XML Schema, whereas OWL-S allows for the definition of abstract types as (description logic-based) OWL classes. WSDL/XSD is unable to express the semantics of an OWL class. Similarly, OWL-S has no means to express the binding information that WSDL captures. Thus, it is natural that an OWL-S/WSDL grounding uses OWL classes as the abstract types of message parts declared in WSDL, and then relies on WSDL binding constructs to specify the formatting of the messages.

Given any Web service that is described using WSDL, it is possible to construct an OWL-S description for the same service, and a grounding that will enable an agent to reason with the OWL-S description and produce the appropriately formatted messages for interactions with that service. Conversely, for any existing OWL-S specification, it is possible to construct a WSDL-enabled Web service that works with that specification. Further details about OWL-S groundings may be found in [44] and in the Technical Overview in [61].

The semantics of OWL-S

OWL-S is an OWL ontology. Since OWL has a well-defined semantics, so too does OWL-S. As a consequence, OWL-S ontologies are unambiguously computer-interpretable. However, this is not the complete story.

OWL is a reasonably expressive language, whose expressive power was limited *by design* in order to balance expressiveness against the computational complexity of the description logic inferences it must support. As a result, more sophisticated inter-relationships between OWL concepts cannot be described within the OWL language. While the OWL-S process model sub-ontology has a well-defined semantics, there are superfluous unintended interpretations of the sub-ontology over and above the developers' intended interpretation. The developers sought out several solutions to this problem. The intended interpretation of the process model was defined in written form in the Technical Overview of OWL-S [61]. The intended interpretation of the process model was also defined by a translation to several other more expressive languages. Three such translations were performed. The intended interpretation of the process model was alternatively defined in first-order logic using the situation calculus, a language for reasoning about action and change (e.g., [59]), using Petri nets [59], and using an execution semantics that incorporates subtype polymorphism [5]. More recently, the semantics of OWL-S was described in the Process Specification Language (PSL) (ISO 18629) a formally axiomatized process ontology [37]. Definition of the semantics by translation to popular more expressive languages has added value in that a number of the automated reasoning systems for reasoning about processes are built around these languages (e.g., first-order logic, Petri nets).

[5] OWL-S allows for groundings to other invocation frameworks besides WSDL; for example, at least one project has utilized a grounding that bottoms out in UPnP. Here, we discuss only the WSDL grounding since that is included with the online releases, and is by far the most widely used.

5.Architecture and deployment

As an ontology, OWL-S does not make any commitment with respect to processing environment or Web services architecture. Indeed, OWL-S has been used in very different conditions, from traditional Web service architectures that adopt the Service Oriented Architecture (SOA) ‘triangle’ set of interactions (among a service registry, producer, and consumer), to P2P systems [24] using a Gnutella-based architecture [67], to the multicasting-based UPnP [47] systems used with wireless devices; to architectures that exploit a centralized mediator [43] [91] to manage the interaction with services and perform different forms of translation [73] between clients and services [68] [69].

The experience accumulated working with OWL-S shows that there are a number of functionalities that the different parties involved in the interaction need to perform, and what changes in the different architectures is the distribution of tasks among the different participants. For example, in an architecture that is centered around a registry the matching of advertisements and requests is concentrated in the registry, while in P2P systems every node should be able to perform such matching.

In this section, we describe how OWL-S can be used in a typical Web service interaction in which the service requester formulates a request based on its goals, queries a registry such as UDDI to find a provider, and finally interacts with the provider. The participants and their roles are shown in Figure 5. The main steps in this process are:

- **Formulation of a request:** The client has a goal that needs to be solved, and it would like to find a Web service that can satisfy the goal. To find such a Web service, the client expresses its goal as an OWL-S profile. The translation of a goal into a profile is an abstraction process from a goal to the capabilities that are required to satisfy that goal. Such capabilities are expressed in terms of the results and information produced as well as the quality of service that the client expects from a provider. An algorithm for generating these abstractions is described in [68].
- **Service Request:** The client uses the service profile that it constructed to query a registry that contains a collection of offered services.
- **Matching:** The registry compares the request to its library of service profiles and returns (URI references for) candidates that achieve that goal. The matching process relies on the relations between the terms used to specify the advertisements and those in the request, both of which are defined using OWL ontologies ([40] [42] [64]).
- **Selection:** The client analyzes the candidates returned by the registry and selects a Web service to interact with. Since it is possible that none of the services found will precisely satisfy all the requirements and objectives of the requester, the selection process may require consideration of a number of different trade-offs [8].
- **Invocation:** The client determines how to make a request to the selected service by unifying its goals and preferences with the effects specified in the service process model, to determine a semantically valid set of inputs to the server. These inputs are then mapped onto a WSDL input message pattern using the OWL-S service grounding, and the resulting message is sent.

- **Reply:** The service receives the request, and determines whether it can perform the request. It may acknowledge the request, send an error, request additional information, or (generally, on completion) send a reply stating the service results.
- **Reply Interpretation:** Upon receiving the reply, the client parses the reply, in accordance with its WSDL specification, and uses the grounding specification to map it into the abstract outputs specified in the process model [62]. If the invocation of the service is successful, the effects of the service hold true in the client's world. This additional knowledge may satisfy preconditions for other services.
- **Goal satisfaction:** The client uses the instantiated outputs to decide whether its goal has been achieved successfully, or whether there is a need for additional interactions with the Web service or for searching for another Web service [93].

When the process model of a service is a composite process, steps 5, 6 and 7 may be repeated a number of times, since the interaction between the client and the server may require numerous information exchanges. For example, any interaction with a B2C Web site such as Amazon's requires the buyer to fill out multiple forms specifying such things as which book to buy, selecting the appropriate edition, specifying where to ship the book, describing which credit card to use, etc. The client may also need to interact with multiple Web services concurrently to find out which one will provide the best result, or with the best quality of service. For more information about the semantics and invocation of composite processes refer to [62].

This interaction process is consistent with the Service Oriented Architecture (SOA) interaction protocol that is used with UDDI, WSDL and SOAP. It reflects the similar roles played by the service profile representation of Web services in OWL-S and the UDDI representations of Web services [65][82].

The difference here is that UDDI interactions require developers in the loop to interpret the registry results and create the calls to the services. Figure 5 shows how the process is implemented using OWL-S for more dynamic Web services interactions. OWL-S does not require a programmer to query UDDI, read the WSDL models found there, and implement those interfaces. The client software agent is responsible for the interaction with the OWL-S registry, for the determination of which candidate services are most appropriate, for the determination of the information required to invoke each service, and for the interpretation and response to messages returned by the service. Variations on this execution model for OWL-S have been implemented by a number of academic and industry researchers (e.g. [62], [79], [59]).

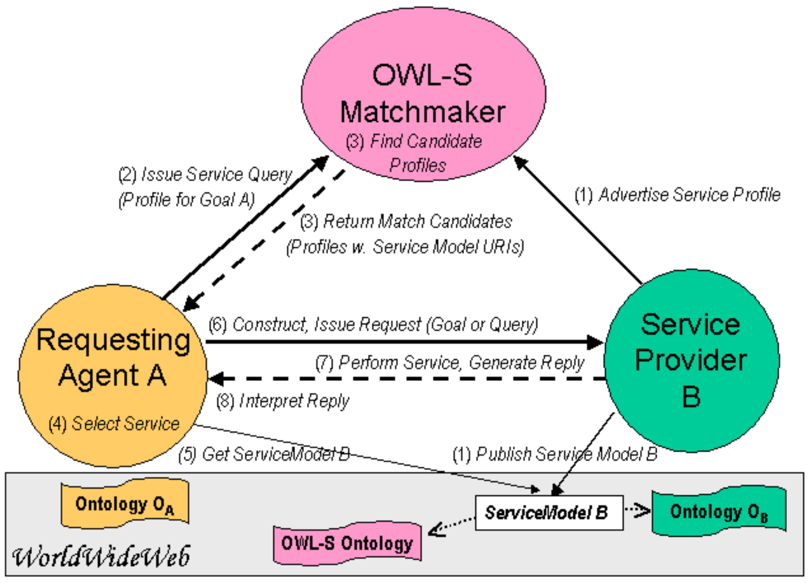

OWL-S Matchmaker
(3) Find Candidate Profiles
(2) Issue Service Query (Profile for Goal A)
(3) Return Match Candidates (Profiles w. Service Model URIs)
(1) Advertise Service Profile
Requesting Agent A
(4) Select Service
Service Provider B
(6) Construct, Issue Request (Goal or Query)
(7) Perform Service, Generate Reply
(8) Interpret Reply
(5) Get ServiceModel B
(1) Publish Service Model B
Ontology O_A
ServiceModel B
Ontology O_B
OWL-S Ontology
WorldWideWeb

Figure 5. Typical OWL-S Semantic Web service interaction model.

6.Application to service discovery and composition

OWL-S aims to support fuller automation and dynamism in service use by providing machine understandable and operational descriptions of services and their capabilities. It has been employed in a wide range of research efforts towards these goals. The most intensive focus (with OWL-S as well as with other Semantic Web service technologies) has been on the enablement of service discovery and composition. In this section, we describe some of the explorations that have taken place, and the additional value that can be realized, around these two critical Web service tasks. While the main focus of our discussion is on work using OWL-S, we also discuss related work based on other approaches.

6.1 Discovery

The task of (Semantic) Web service discovery was introduced in Section 3. Capability-based matching and semantic matching are two examples of where semantic Web service discovery based on the OWL-S profile sub-ontology provides value-added over existing Web service discovery standards. Capability-based matching enables agents to find services that aptly match their requirements for particular results, while semantic matching allows flexibility in identifying the service parameters needed (and available to the client) to specify the criteria of the requested service.

UDDI provides a number of means of searching its registry of services; services can be searched by name, by business, by bindings or by TModels. For example, it is possible to look for services that adhere to Rosetta Net specifications or services that sell new cars. Unfortunately, the search mechanism supported by UDDI is limited to keyword matches and does not support any inference based on the taxonomies referred to by the TModels. For example a car selling service may describe itself as “New Car Dealers” which is an entry in NAICS, but a search for “Automobile Dealers” services will not identify the car selling service despite the fact that “New Car Dealers” may be regarded as a subtype of “Automobile Dealers”. Since OWL-S uses OWL ontologies to describe functional and non-functional properties it overcomes the limitations of syntactic search by providing semantic search based on subsumption relations.

Broadly speaking, there are two ways to represent capabilities of Web services. The first approach provides an extensive class hierarchy with each class representing a set of similar services. Using such a hierarchy, services may be defined as instances of classes that represent their capabilities. Amazon, for example, may advertise itself as a member of an OWL class for bookselling services. (For advertising purposes, such a class would be a subclass of `Profile`.). The second way to represent a service’s capabilities is to provide a generic description of its functionality in terms of the state transformations that it produces. For example, Amazon may specify that it provides a service that takes as input a book title, author, address and a valid credit card number, and produces a state transition such that the book is delivered to address, the credit card is charged the book price, and the book delivered changes its ownership. Despite their differences, both ways to represent capabilities use ontologies to provide the connection between what the Web service does and the general description of the environment in which the Web service operates.

There are trade-offs between these two styles of representation in service discovery. The use of an explicit ontology of capabilities facilitates the discovery process since the matching process is reduced to subsumption tests between the capabilities expressed in service profiles and those expressed in a service discovery query. Both descriptions reference concepts in the ontologies of the domain the service covers. On the other hand, enumerating all possible capabilities even in restricted domains for ontology encoding may be difficult. For example, consider the problem of representing translation services from a source language LS to a target language LT. Assuming n possible languages, there are n^2 possible types of translation services. A service taxonomy might have different classes of service for each pair of languages that could be translated, or it might just represent “translation services” as one general category, with explicit properties that allow particular services to describe the languages that they can translate from and translate to. This latter approach is consistent with describing the capability in terms of a state transformation. It distinguishes the translators by describing how they produce different kinds of results. OWL-S provides support for both styles of service description.

A number of capability matching algorithms using OWL-S service profile descriptions have been proposed. In general, they exploit one of the two views of the capabilities described above. Matching algorithms, such as described in [40], assume the availability of ontologies of functionalities to express capabilities. Matching between a request and the available advertisements is reduced to their subsumption relation. Different degrees of match are detected depending on whether the advertisement and the request describe the same capability or whether one subsumes the other.

Other matching algorithms, such as in [64], [82], and [33], assume that capabilities are described by the state transformation produced by the Web service. Paolucci et al. [64] describe a matchmaking algorithm that compares the state transformations described in the request to the ones described in the advertisements. More specifically, the subsumption relation between input and output types are examined to generate flexible matches, which are beyond the capabilities of existing text-based matching engines. They perform two matches, one comparing outputs and one comparing inputs. If the output required by the requester is of a kind covered (subsumed) by the advertisement, then the inputs are checked. If the inputs specified in the request are subsumed by the input types acceptable to the service, then the service is a candidate to accomplish the requester's requirement. Numerous extensions have been proposed to improve the matchmaking algorithms by exploiting more features of subsumption relations [40]. Instead of looking at only the parameter types, these matching algorithms treat the whole OWL-S profile description as one OWL concept and go beyond equality and subsumption matching. Researchers have also pointed out the need to have a ranking mechanism to evaluate the matches produced. Benatallah et al. [9] describe one such ranking method based on the algorithms derived from hypergraph theory. Cardoso et al. [15] compute the syntactic, operational and semantic similarities expressed in OWL-S profiles to generate a ranking function.

Using OWL-S for Web service discovery has not been limited to matching algorithms. Several different researchers have also investigated how OWL-S can be integrated into existing Web service discovery architectures. The OWL-S/UDDI matchmaker [82] integrates OWL-S capability matching into the UDDI registry. This integration is based on the mapping of OWL-S service profiles into UDDI Web service representations [65]. Akkiraju et al. [1] improved this coupling by providing additional extensions to the UDDI inquiry API and enhancing the service discovery of UDDI by performing semantic matching and automatic service composition using planning algorithms. Other efforts show how to integrate OWL-S with different network protocols and P2P infrastructure. Specifically, Paolucci et al. [67] discussed how to use OWL-S based discovery in Gnutella P2P environment, Elenius and Ingmarsson [24] achieved similar functionality in JXTA P2P infrastructure and Masuoka et al. [47] presented discovery methods in pervasive environments by utilizing UPnP protocol.

6.2 Composition

The task of automated Web service composition was introduced in Section 3. Automated composition is the process of automatically selecting, composing, integrating and executing Web services to achieve a user's objective. "Make the travel arrangements for my WWW2005 conference trip" or "Buy me an Apple iPod at the best available price" are examples of possible user objectives addressed by composition. Industrial standards provide tools for specifying a manually generated composition so that it can be executed automatically; BPEL is an example of such an orchestration system. Human beings perform manual Web service composition by exploiting their cultural knowledge of what a Web service does (e.g., that www.apple.com will debit your credit card and send you an iPod), as well as information provided on the service's Web pages, in order to compose and execute a collection of services to achieve some objective. To automate Web service composition, all this information must be encoded explicitly in an unambiguous computer-interpretable form. None of the existing industrial standards for Web Service description encode this level of detail. Further, the descriptions they provide are not unambiguously computer interpretable and as a consequence not reliably manipulated by an automated reasoning system; hence the need for OWL-S.

Automated Web service Composition is impossible to achieve using existing Web service standards, so the obvious value-added of OWL-S is that it enables automation of a task that is now being performed manually. Even after the composition has been developed, the advantage of the OWL-S approach compared with other approaches, namely BPEL4WS and WS-CDL, is that OWL-S allows the flexibility to change the composition structure on the fly to adapt to changing conditions. For example, if a Web service that is needed within a composition is no longer available, or it does not have the product that the client needs, the overall composition does not fail; rather it can be modified by introducing on the fly other Web services that solve the problem at hand. As a result, OWL-S supports a form of dynamic Web service composition that is not supported by any other Web service standard.

Automated Web service Composition is akin to both an AI planning problem and a software synthesis problem, and draws heavily on both of these areas of research. In order to perform automated Web service composition, a reasoning system must select, compose, integrate and execute Web services that collectively achieve the user's objective. This involves resolving constraints between Web service inputs, outputs, preconditions and results (IOPRs) and the properties of the services that the user desires. Many of these Web service properties, including preconditions, effects, and non-functional properties of the service are not encoded in any existing industrial standard. They are encoded, in unambiguous computer-interpretable form in OWL-S.

There are several different approaches to Web service Composition based on OWL-S descriptions. For example, the OWL-S process model has been used to construct high-level generic procedures that approximately describe how to achieve some objective. Then these high-level plans are expanded and refined using automated reasoning machinery. This is the task of nondeterminism reduction discussed previously. The first such system was the Golog system [55, 56]. This system interleaves the execution of information-providing services with the simulation of world-altering ones to generate a sequence of Web services customized to user's preferences and constraints. In a similar spirit, several other researchers [32, 79] have used the paradigm of Hierarchical Task Network (HTN) planning [60] to perform automated Web service composition. HTN planning has also been used in [66] to compose Web services in the travel domain and in the organization of a B2B supply chain. Other planning techniques that have been applied to the composition of OWL-S services, range from simple classical STRIPS-style planning [78], to extended estimated-regression planning [48], to "Planning as Model Checking" [84] that deals with non-determinism, partial observability, and complex goals. Some of these systems limit themselves to dealing only with atomic processes, since operator-based planning systems are unable to represent composite processes. In [57], the authors propose a way of compiling a large class of composite process specifications into atomic process specifications.

There are many systems that deal with a restricted service composition problem where preconditions and results (effects) are ignored. Constantinescu et al. (e.g., [19]) presents a system that constructs chains of services based on partial matches of input/output types determined at runtime. These have great utility for information integration. Web Service Composer [80] uses an interactive composition method based on parameter types to put services together and filters the potential matches using information from the compositional context. An improved version of this approach is an end-user interactive composition system, STEER [46]. STEER lets fairly unsophisticated users find, select, compose and execute local, pervasive and remote Web services to achieve common tasks. The system combines services that adhere to different ontologies by utilizing semantic mappings and/or transformation functions that are also published as Web services.

Another OWL-S based approach for composition employed in the Semantic Discovery Service (SDS) [42] is to augment the standard BPEL4WS execution engine with an OWL-S ontology to enable run-time discovery and binding of services. The integration of OWL-S with BPEL demonstrates the value-added of OWL-S relative to existing industrial Web service orchestration systems. The system performs semantic integration and interoperation in the context of dynamic customization of existing Web service compositions (BPEL workflows). Customizing constraints provided by the user enable the dynamic binding of user-customized services in the workflow. However, this dynamic binding sometimes requires further semantic integration. OWL-S is used to describe the services, the customizing user constraints, and to perform Web service discovery and semantic integration. This system was extended to add an explanation facility that enables rich explanation of the system's success or failure at creating an integrated composition [52]. The explainable system integrates Inference Web [54] with the existing system. Composition steps are registered using the Inference Web and both successful and failed composition efforts are explained, providing transparency and audit information.

While all of this work is promising, we are still some distance from the goal of automated Web service composition. With adoption of approaches to Web service description such as OWL-S and advances in planning-related and program synthesis technologies, we believe that broad-scale automated Web service composition is well within reach.

7.Extensions, applications, and tools

The release of OWL-S has stimulated a lot of interesting research in the Semantic Web services area. Applications from industry have begun to emerge as the need for extra semantics has become increasingly evident. As a result, a wide variety of OWL-S applications and tools are now available. OWL-S has also been extended by researchers to support various application specific requirements. OWL-S is the most mature and by far the most used Semantic Web Service framework. As discussed in Section 6, many researchers have developed methods for automated discovery, matchmaking, and composition of Web services using OWL-S. In this section, we summarize other applications, tools and extensions that build on or support OWL-S.

The OWL-S Editor [25], a plug-in to the popular ontology development tool, Protégé [31], provides a comprehensive set of capabilities for creating and maintaining OWL-S service descriptions. Among other things, the editor allows the user to build complex process models using a GUI. Another OWL-S editor [75] is provided by University of Malta and provides functionality to create, validate and visualize OWL-S services. DINST [30] is another graphical tool to visually build OWL-S services. The service creation process sets up a layered service ontology where OWL-S is used as an upper service ontology.

CMU OWL-S Development Environment (CODE) [83] supports a semantic Web service developer through the whole process from the Java generation, to the compilation of OWL-S descriptions, to the deployment and registration with UDDI. CODE includes modules such as WSDL2OWL-S translator, an Eclipse-based editor and the OWL-S Virtual Machine [62]. Mindswap OWL-S API [81] provides a Java API for programmatic access to read, execute and write OWL-S service descriptions. The API provides functionality for parsing, serializing, validating, reasoning, and executing services for various different versions of OWL-S.

There are also tools that primarily focus on the generation of OWL-S descriptions from WSDL files. One such tool is Assam (Automated Semantic Service Annotation with Machine learning) [38] that assists the user to semantically annotate (legacy) WSDL services. The tool uses a combination of different machine learning techniques such as iterative relational classification and ensemble learning to semi-automate the annotation process. OntoLink (Ontology-Linker) [46] is another tool that helps users to generate executable OWL-S descriptions from WSDL files. The emphasis of the tool is to semi-automate the generation of *grounding specifications* by letting the user define mappings between ontologies and XML schema definitions through a GUI. An XSLT transformation is automatically generated based on the mappings the user defined and the resulting OWL-S grounding is directly executable. OntoLink's other functionality is to specify semantic and syntactic mappings/transformations between concepts defined in different ontologies (in a similar fashion) so that services defined using disjoint ontologies can interoperate together. Mappings between ontologies are generated using XSLT and automatically wrapped inside a so-called *translator service*. These translator services transform the output of one service to a compatible format that another service can consume so that two previously incompatible services can be composed together.

In addition to developing tools for OWL-S, researchers have proposed several extensions to OWL-S framework. For example, Denker et al. [22] present security annotations for OWL-S services to enable brokering between agents and services. A set of ontologies that describe credentials, security mechanisms and privacy options has been used for this purpose. Kagal et al. [28] extends this framework to express such annotations in Rei [29], a logic-based language for defining rules and constraints over domain-specific ontologies. The OWL-S Matchmaker system [64] is also extended for policy matching and the OWL-S Virtual Machine (VM) [62] has been extended to enforce policies and security mechanisms. Similar extensions to OWL-S have been proposed in [86] to support policy and contract management using KAoS services and tools. A policy management framework is used as an authorization service in grid computing environments, as a distributed policy specification and enforcement capability for a semantic matchmaker, and as a verification tool for service composition and contract management. Other kinds of extensions to OWL-S include augmenting the process ontology to model WS brokers [68], extensions to describe concrete application servers which facilitate reusing and combining Semantic Web software modules (e.g. ontology stores, reasoners, etc.), adaptations of OWL-S for describing Web services in the bioinformatics domain [92] and enriching OWL-S with speech-acts to describe agent based Web services [36].

The OWL-S Web site [61] includes pages listing related publications, tools, and use cases from the community at large. In addition, a number of open-source tools are hosted at www.semwebcentral.org. Discussion of issues related to OWL-S is conducted on the public mailing list of the W3C's Semantic Web Services Interest Group [76].

8.Related work

Here, we comment briefly on the relationship of OWL-S to selected Web service and Semantic Web technologies. (Related work focusing on service discovery and composition is included in Section 6.) Space does not permit a comprehensive description of related work, and we emphasize that there are many relevant technologies that are beyond the scope of this article. The technologies discussed here are: BPEL4WS, CDL, ebXML, Grid services, SWSF, and WSMO, and WSDL-S.

The Business Process Execution Language for Web Services (BPEL4WS) [4] enables the specification of executable business processes (including Web services) and business process protocols in terms of their execution logic or control flow. Executable business processes specify the behavior of individual participants within Web service interactions and can be invoked directly, whereas business process protocols (also called abstract processes) abstract from internal behavior to describe the messages exchanged by the various Web services within an interaction.

Abstract processes only consider protocol-relevant data and ignore process-internal data and computation. The effects of such computation on the business protocol are then described using non-deterministic data values. Executable processes, on the other hand, are described using a process description language that deals with both protocol-relevant and process-internal data. BPEL4WS also defines several mechanisms for recovery from faults, such as catching and handling of faults, and compensation handlers that specify compensatory activities in the event of actions that cannot be explicitly undone.

The area of greatest overlap between BPEL4WS and OWL-S is in the process model, but there are some crucial differences. OWL-S' process notation includes preconditions and results, which enables automated tools to select and compose Web services. The BPEL4WS notation includes complex control structures and exception-recovery protocols. It would be fairly easy to add such features to OWL-S, but we are reluctant to do so without further research on how to automate reasoning about them.

The OWL-S process model and BPEL4WS attempt to cover similar territory, but in complementary ways. The emphasis in OWL-S is on making process descriptions computer-interpretable — described with sufficient information to enable *automation* of a variety of tasks including Web service discovery, invocation, and composition. BPEL4WS provides a language primarily intended for *manually* constructing processes and protocols. The two design goals are not really incompatible. For instance, recent work [42] has shown how BPEL4WS can use OWL-S descriptions of services to augment its functionality to include tasks such as dynamic partner binding and semantic integration.

ebXML (Electronic Business using eXtensible Markup Language) [23] addresses the broad problem of B2B interaction from a workflow perspective. Business interactions are described through two views: a Business Operational View (BOV) and a Functional Service View (FSV). The BOV deals with the semantics of business data transactions, which include operational conventions, agreements, mutual obligations and the like between businesses. The FSV deals with the supporting services: their capabilities, interfaces and protocols. Although ebXML does not restrict the means of B2B interaction to Web services, their focus is essentially the same as that of OWL-S.

ebXML enables Web services to describe the business processes they support and the services they offer using Collaboration Protocol Profiles (CPP). A business process in ebXML is considered to be a set of business document exchanges between a set of Web services. This is akin to the Web services message exchange as commonly described in Web services standards. CPPs contain industry classification, contact information, supported business processes, interface requirements etc. They are registered within an ebXML registry (similar to a UDDI registry), in which other Web services and their business processes can be discovered. However, ebXML's scope does not extend to the format in which the business documents are specified. This is left to the interacting Web services to agree upon a priori by the creation of a Collaboration Protocol Agreement. Since OWL-S provides a language for the description of the behavior of Web services, its scope is complementary to that of ebXML. In fact, OWL-S descriptions could themselves be used within ebXML to describe the business processes of interacting Web services.

The **Web Service Choreography Description Language (CDL)** describes Web service interactions in terms of their externally observable behavior, typically exposed though message exchanges. Each participant in an interaction specifies an interface, describing the temporal ordering and logical dependence of messages it sends and receives. In addition, a global model can be specified that describes the collective message exchange of interacting Web services. Unlike BPEL4WS, CDL does not describe the executable details of individual Web services. It focuses on the problem of collaboration (message exchange between distributed peers) rather than orchestration (creation of executable Web services). Consequently, it does not assume the presence of a central process managing the interactions between Web services. For the same reason, CDL control flow constructs are considerably simpler than those of BPEL4WS. CDL documents are formal specifications intended for explication, (automated) verification, and conformance testing, although they can be used to automatically generate code skeletons.

CDL corresponds most closely to the OWL-S process model. Both share the goals of describing the message exchange between participating Web services. However, unlike CDL, OWL-S markup is also meant to support execution and the generation of executable service compositions (e.g., using planning techniques). Also, CDL specifically eschews any description of the *significance*, e.g., the business significance, of interactions whereas OWL-S is specifically intended to support the description of everything from the "real world" preconditions and results of an invocation to the classification of services by their primary purpose.

Grid services. A Grid is a system designed "to coordinate resources that are not subject to centralized control using standard, open, general purpose protocols and interfaces to deliver nontrivial quality of service" [26]. The resources available on a Grid are modeled as *Grid services* with state. The Open Grid Services Infrastructure (OGSI) uses extended WSDL constructs and XML schemas to introduce the notion of stateful Web services along with standard operations for creating and destroying Web services, and for representing, querying, and updating metadata associated with a Web service. In addition, OGSI provides mechanisms for references to instances of Web services and asynchronous notification of Web service state changes. OGSI also defines an XML Schema for describing faults that take place during WSDL operations.

The OGSI framework has been recently refactored to define a family of related specifications that could be adopted on a piecemeal basis. This refactoring together with some extensions to keep pace with changes in Web services standards forms the WS-Resource Framework (WSRF). Although OGSI and WSRF go beyond WSDL with the specification of several kinds of specialized port types and fault messages, they do not attempt to describe the behavior of Grid or Web services as OWL-S does. In this sense, OWL-S is complementary to both of these specifications.

The **Semantic Web Services Framework (SWSF)** [77], published online recently by the Semantic Web Services Initiative, builds loosely on OWL-S to provide a more comprehensive framework, in the sense of defining a larger set of concepts. It also builds on a mature pre-existing ontology of process modeling concepts, the Process Specification Language (PSL). SWSF specifies a Web-oriented language, SWSL, with a logic programming layer and also a first-order logic layer. It uses SWSL to define an ontology of service concepts (SWSO), and takes advantage of SWSL's greater expressiveness (relative to OWL) to more completely axiomatize the concepts. Since SWSO is completely axiomatized in first-order logic, it avoids the need to use hybrid (DL-based and FOL-based) reasoning as is the case with OWL-S.

The **Web Services Modeling Ontology (WSMO)** [90] shares with OWL-S and SWSF the vision that ontologies are essential to support automatic discovery, interoperation, composition, etc. of Web services. Like SWSF, the WSMO effort defines an expressive Web-oriented language, WSML [21], which provides a uniform syntax for subdialects ranging from description logic to first-order logic. Like OWL-S, WSMO declares inputs, outputs, preconditions, and results (although using somewhat different terminology) associated with services. Unlike OWL-S, WSMO does not provide a notation for building up composite processes in terms of control flow and data flow. Instead, it focuses on specification of internal and external choreography, using an approach based on abstract state machines. Other distinguishing characteristics include WSMO's emphasis on the production of a reference implementation of an execution environment, WSMX, and on the specification of mediators --- mapping programs that solve the interoperation problems between Web services.

In WSMO's approach, mediators perform tasks such as translation between ontologies, or between the messages that one Web service produces and those that another Web service expects. WSMO includes a taxonomy of possible mediators that helps to classify the different tasks that mediators are supposed to solve. The definition of mediators in WSMO calls attention to some important translation tasks associated with Web services. Not surprisingly, these same translation tasks are needed in support of interactions with OWL-S described Web, and some OWL-S based systems also make use of mediator components. However, rather than stipulating the existence of a distinct type of component in the Web services infrastructure, OWL-S provides to Web services and their clients the information that is needed to find existing mediation services that can reconcile their mismatches, or perhaps to create mediators through the process of Web service composition [69].

WSDL-S [2] is a small set of proposed extensions to WSDL by which semantic annotations may be associated with WSDL elements such as operations, input and output type schemas, and interfaces. WSDL-S aims to support the use of "semantic concepts analogous to those in OWL-S while being agnostic to the semantic representation language" [2]. The way in which WSDL-S allows one to specify a correspondence between WSDL elements and semantic concepts is very similar to how the OWL-S grounding works; indeed, something very much like OWL-S declarations could be used as the referents of the WSDL-S attributes. The most notable difference between WSDL-S and the OWL-S grounding is that with WSDL-S the correspondence must be given in the WSDL spec, whereas with OWL-S it is given in a separate OWL document. (The OWL-S grounding also proposes some WSDL extension elements, but they are regarded as optional for most purposes.) Thus, the aims of WSDL-S are compatible with those of OWL-S, but WSDL-S focuses on the practical advantages of a more lightweight, incremental approach.

9.Conclusions and future directions

OWL-S is an ontology designed for use in describing Web services. We have given an overview of OWL-S and discussed some of the ways it can be used to automate a variety of service-related activities, focusing particularly on service discovery, interoperation, and composition. While noting that OWL-S makes no commitments to a particular system architecture, we have discussed some of the architectural choices that work well with it.

OWL-S has been used in many ways, to address a range of challenges, and numerous extensions have been proposed and demonstrated in prototype systems. We believe this indicates both the need for this kind of approach, and its flexibility and generality.

For OWL-S (or a similar approach) to become widely used will require that it become more tightly integrated with commercial Web services standards and that mature tool support makes it easier to use for non-expert developers. This article introduces the OWL-S process model surface syntax, which is a step in that direction. Further work on algorithms that effectively support discovery, selection, and composition of services in large-scale, complex settings is also needed. In addition, it will be important to conduct empirical evaluation of the applicability and benefits of OWL-S in developing and managing service-based systems in businesses and other settings.

There is a need for more complete guidelines and documentation of best practices regarding architectural configurations using OWL-S and the expected behavior of common components, such as execution engines, matchmakers, and service composers. Other future work includes fundamental ontology extensions to support error-handling, additional work on security and quality-of-service specifications and also extensions to allow for the specification of specific instances (executions) of process models, which are needed to support monitoring and recovery. Other fundamental extensions are needed to support negotiation and contracting.

Finally, there is a need to develop specializations of the profile for a variety of domains, and for broad categories of services. For example, for services that sell goods, ontology modules are needed for the specification of cost models and product guarantees. To encourage the participation of communities of interest in developing these extensions, we plan to investigate the use of an open-source style of evolution as we go forward.

We see OWL-S as having made a critical first step in the process of formulating the necessary declarative representational framework for fully specifying the capabilities and behavior of Web services, in a way that supports greater automation of service-related activities, such as discovery and composition. We believe that it should continue to be exercised and developed to meet the needs of the full array of such activities, including negotiation and contracting, monitoring and recovery, and so forth. We expect that the need to better support such activities in the world of Web services will lead to significant further evolution of OWL-S and similar approaches.

10.Acknowledgements

This paper describes OWL-S Version 1.2. This work began in February 2001 with the convening of the DAML Services Coalition. Since then, there have been 4 releases of DAML-S followed by 4 subsequent releases of OWL-S after DAML+OIL evolved into OWL. Numerous people have contributed to this effort, many of whom are not reflected in the authorship of this paper. In particular, we wish to acknowledge the contributions of Anupriya Ankolekar, Grit Denker, Daniel Elenius, Jerry Hobbs, Lalana Kagal, Ora Lassila, Srini Narayanan, Bijan Parsia, Terry Payne, Marta Sabou, Craig Schlenoff, Monika Solanki, Randy Washington and Honglei Zeng. (Many of the above have also been active participants in the OWL-S Coalition at different times.) We also wish to thank Ian Horrocks, Peter Patel-Schneider and Michael Uschold for helpful comments at various stages of the development of this ontology. This work has also benefited from input from members of the Semantic Web Services Initiative (SWSI). We thank the W3C for providing the use of the public-sws-ig mailing list for discussion of OWL-S issues, and Carine Bournez at W3C for helping to make this possible. In addition, we would like to express our gratitude to the many interested researchers and users who have employed OWL-S in their projects and publications, and to those who have taken the trouble to share their suggestions and questions in the discussions of OWL-S on the list.

Our tremendous thanks go to Jim Hendler, Murray Burke and Mark Greaves (DAML program managers) for their tremendous support and championing of this effort. Finally we gratefully acknowledge funding from the DARPA DAML program, without which this effort would not have been possible.

11.References

1. R. Akkiraju, R. Goodwin, P. Doshi, and S. Roeder, "A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI", in *Proceedings of IJCAI Information Integration on the Web Workshop*, Acapulco, Mexico, August 2003.
2. R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma, "Web Service Semantics - WSDL-S", Technical Note, Version 1.0, April, 2005, available at http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html.
3. J. L. Ambite (Ed.), *Proceedings of the ICAPS2003 Workshop on Planning for Web Services*, 2003.
4. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte (Editor), I. Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services", Version 1.1, 2003, at http://www.ibm.com/developerworks/webservices/library/ws-bpel/.
5. Ankolekar, F. Huch and K. Sycara, "Concurrent Execution Semantics of DAML-S with Subtypes", In *Proc. of the First International Semantic Web Conference (ISWC 2002)*, 2002.
6. Ankolekar, M. Paolucci, and K. Sycara, "Towards a Formal Verification of OWL-S Process Models", In *Proc. International Semantic Web Conference (ISWC 2005)*, 2005.
7. Apache Axis, at http://ws.apache.org/axis/.
8. W.-T. Balke and M. Wagner, "Towards personalized selection of Web Services", in *Proceedings of the International World Wide Web Conf.*, Budapest, Hungary, 2003.
9. B. Benatallah, M. Hacid, C. Rey and F. Toumani, "Request Rewriting-Based Web Service Discovery", in *Proceedings of the Second International Semantic Web Conference* (ISWC 2003), pp 335-350, October 2003.
10. T. Berners-Lee, "Primer: Getting into Rdf and the Semantic Web using N3", at http://www.w3.org/2000/10/swap/Primer.html, 2000.
11. T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web", *Scientific American*, May 2001.
12. Bernstein and M. Klein, "High Precision Service Retrieval", in *Proceedings of the First International Semantic Web Conference* (ISWC 2002), Sardegna, 2002.
13. P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, "Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking", in *Proceedings of the 17th Int. Joint Conference on Artificial Intelligence Conference* (IJCAI'01), 2001.
14. P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, H. Stuckenschmidt: "C-OWL: Contextualizing Ontologies", *Proceedings of the International Semantic Web Conference*, pp. 164-179, October 2003.
15. M. Burstein, C. Bussler, T. Finin, M. Huhns, M. Paolucci, A. Sheth, S. Williams and M. Zaremba , "A Semantic Web Services Architecture" *IEEE Internet Computing*, September-October 2005.
16. J. Cardoso and A. Sheth, "Semantic E-workflow Composition", *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.
17. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1", 2001,
at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.
18. S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello, "Logic Based Approach to Web Services Discovery and Matchmaking", in *Proceedings of E-services workshop at 5th International Conference on Electronic Commerce* (ICEC'03), Pittsburgh, USA, 2003.
19. Constantinescu, B. Faltings, and W. Binder, "Large Scale, Type-Compatible Service Composition", in *Proceedings of the 2nd International Conference on Web Services* (ICWS 2004), San Diego, USA, July 2004.
20. M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P.F. Patel-Schneider, and L. A. Stein, "Web Ontology Language (OWL) W3C Reference version 1.0", 18 August 2003, at http://www.w3.org/TR/2002/WD-owl-ref-20021112.

21. J. De Bruijn, H. Lausen, A. Polleres, and D. Fensel, “The Web Service Modeling Language WSML: An Overview,” *DERI Technical Report 2005-06-16*, 2005, at http://www.wsmo.org/wsml/wsml-resources/deri-tr-2005-06-16.pdf.
22. G. Denker, L. Kagal, T. Finin, M. Paolucci, N. Srinivasan and K. Sycara, “Security For DAML Web Services: Annotation and Matchmaking”, in *Proceedings of the Second International Semantic Web Conference* (ISWC 2003), pp. 335-350, October 2003.
23. Electronic Business using eXtensible Markup Language (ebXML) Web site, at http://www.ebxml.org/.
24. D. Elenius, M. Ingmarsson, “Ontology-based Service Discovery in P2P Networks”, in *Proceedings of the MobiQuitous'04 Workshop on Peer-to-Peer Knowledge Management (P2PKM 2004),* Boston, USA, August 2004.
25. D. Elenius, G. Denker, D. Martin, F. Gilham, J. Khouri, S. Sadaati, R. Senanayake, “The OWL-S Editor -- A Development Tool for Semantic Web Services”, in *Proceedings of the 2nd European Semantic Web Conference*, May 2005.
26. Foster, “What is the Grid? A Three Point Checklist”, July 20, 2002, at http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf.
27. E. Friedman-Hill, “Jess, the Rule Engine for the Java Platform”, at http://herzberg.ca.sandia.gov/jess/.
28. L. Kagal, M. Paoucci, N. Srinivasan, G. Denker, T. Finin, and K. Sycara, “Authorization and Privacy for Semantic Web Services*”, IEEE Intelligent Systems* 19(4): 50-56, July 2004.
29. L. Kagal, T. Finin, and A. Joshi, “A Policy Based Approach to Security on the Semantic Web”, in *Proceedings of the 2nd International Semantic Web Conference (ISWC2003),* Sanibel Island, FL, October 2003.
30. M. Klein and B. König-Ries, “A Process and a Tool for Creating Service Descriptions based on DAML-S”, in *Proceedings of 4th VLDB Workshop on Technologies for E-Services* (TES'03), Berlin, Germany, September 2003.
31. H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen, “The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications”, in *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan (2004).
32. U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler, “Information Gathering During Planning for Web Service Composition”, in *Proceedings of the Third International Semantic Web Conference* (ISWC2004), Hiroshima, Japan, November 2004.
33. T. Di Noia, E. Di Sciacio, F. M. Donini and M. Mongiello, “Semantic Matchmaking in a P-2-P Electronic Marketplace”, SAC 2003, pp. 582-586, 2003.
34. R. Fikes and N. J. Nilsson, “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”, *Artificial Intelligence 2*, pp. 189-208, 1971.
35. F. Gandon and N. Sadeh, “Semantic Web Technologies to Reconcile Privacy and Context Awareness”, *Web Semantics Journal*, 1(3), 2004.
36. N. Gibbins, S. Harris, and N. Shadbolt, “Agent-based Semantic Web Services”, in *Proc. of the 12th International WWW Conference* (WWW2003), 2003.
37. M. Gruninger, “Applications of PSL to Semantic Web Services”, In Proc. *Workshop on Semantic Web and Databases*. Very Large Databases Conference, Berlin 2003.
38. Heß, E. Johnston and N. Kushmerick, “ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services”, in *Proceedings of the 3rd International Semantic Web Conference* (ISWC2004), Hiroshima, Japan, November 2004.
39. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, “SWRL: A Semantic Web Rule Language Combining OWL and RuleML”, W3C Member Submission, May 2004, at http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.
40. L. Li and I. Horrocks, “A Software Framework for Matchmaking Based on Semantic Web Technology”, in *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331-339, ACM, 2003.
41. T. W. Malone, K. Crowston, B. P. Jintae Lee, C. Dellarocas, G. Wyner, J. Quimby, C. S. Osborn, A. Bernstein, G. Herman, M. Klein, and E. O'Donnell, “Tools for Inventing Organizations: Toward a Handbook of Organizational Processes”, *Management Science*, 45(3): 425--443, March, 1997.
42. D. Mandell and S. McIlraith, “Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation”, in *Proceedings of the Second International Semantic Web Conference (ISWC2003),* pp. 227--241, 2003. Code for the working system is available at: http://projects.semwebcentral.org/
43. D. Martin, A. Cheyer, and D. Moran, “The Open Agent Architecture: A Framework for Building Distributed Software Systems,” Applied Artificial Intelligence, vol. 13, nos. 1 and 2, 1999, pp. 91–128.
44. D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, S. McIlraith, “Describing Web Services using OWL-S and WSDL”, in Release 1.2 of OWL-S, at http://www.daml.org/services/owl-s/1.2/owl-s-wsdl.html.
45. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne,.E. Sirin, N. Srinivasan, K. Sycara, “OWL-S: Semantic Markup for Web Services” – W3C Member Submission 22 November 2004, at http://www.w3.org/Submission/2004/07/.
46. R. Masuoka, Y. Labrou, B. Parsia, and E. Sirin, “Ontology-Enabled Pervasive Computing Applications”, in *IEEE Intelligent Systems*, 18(10): 68-72, 2003.
47. R. Masuoka, B. Parsia and Y. Labrou, “Task Computing - The Semantic Web meets Pervasive Computing”, in *Proceedings of the Second International Semantic Web Conference* (ISWC 2003), Sanibel Island, FL, USA, October 2003.
48. D. McDermott, “Estimated-Regression Planning for Interactions with Web Services”, in *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems* (AIPS2002), Toulouse, France, April 2002.

49. D McDermott, “The Planning Domain Definition Language Manual”, *Yale Computer Science Report 1165* (CVC Report 980003), 1998.
50. D. McDermott, “Surface Syntax for OWL-S,” in Release 1.2 of OWL-S, at http://www.daml.org/services/owl-s/1.2/owl-s-gram/owl-s-gram-htm.html.
51. D. McDermott and D. Dou, “Representing Disjunction and Quantifiers in RDF”, *Proceedings of the First International Semantic Web Conference* (ISWC2002), 2002.
52. D. L. McGuinness, D. Mandell, S. Mcllraith, P. Pinheiro da Silva, “Explainable Semantic Discovery Services”, *Stanford Networking Research Center Project Review*, Stanford, CA, February 2005.
53. D. L. McGuinness and F. van Harmelen, “OWL Web Ontology Language Overview”, World Wide Web Consortium (W3C) Recommendation. February 10, 2004, at http://www.w3.org/TR/owl-features/
54. D. L. McGuinness and P. P. da Silva, “Explaining Answers from the Semantic Web: The Inference Web Approach”, *Journal of Web Semantics*. (1) 4:397-413, October 2004.
55. S. McIlraith and T. Son, “Adapting Golog for Composition of Semantic Web Services”, in *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning* (KR2002), pp. 482-493, 2002.
56. S. McIlraith., T.C. Son and H. Zeng, “Semantic Web Services”, *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46--53, March/April, 2001.
57. S. McIlraith and R. Fadel, “Planning with Complex Actions”, in *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning (NMR2002)*, pages 356-364, April, 2002.
58. D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, “SHOP: Simple Hierarchical Ordered Planner”, in *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI-99), pp.968—973, 1999.
59. S. Narayanan and S. McIlraith, “Simulation, Verification and Automated Composition of Web Services”, in *Proceedings of the Eleventh International World Wide Web Conference* (WWW2002), Honolulu, Hawaii, May 2002.
60. D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman, “SHOP2: An HTN Planning System”, *Journal of Artificial Intelligence Research,* 2003*.*
61. D. Martin, M. Burstein, D. McDermott, D. McGuinness, S. McIlraith, M. Paolucci, E. Sirin, N. Srinivasan, K. Sycara, “OWL-S 1.2 Release”, At http://www.daml.org/services/owl-s/1.2/.
62. M. Paolucci, A. Ankolekar, N. Srinivasan, and K. Sycara, “The DAML-S Virtual Machine”, in *Proceedings of the Second International Semantic Web Conference* (ISWC 2003), pp 335-350, October 2003.
63. M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura, “Toward a Semantic Choreography of Web services: from WSDL to DAML-S”, in *Proceedings of ICWS03*, 2003.
64. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, “Semantic Matching of Web Services Capabilities”, in *Proceedings of the First International Semantic Web Conference* (ISWC2002), 2002.
65. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, “Importing the Semantic Web in UDDI”, in *Proceedings of E-Services and the Semantic Web* (ESSW02), 2002.
66. M. Paolucci, K. Sycara, and T. Kawamura, “Delivering Semantic Web Services”, in *Proceedings of the Twelfth World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 2003, pp 111- 118.
67. M. Paolucci, K. Sycara, T. Nishimura, and N. Srinivasan, “Using DAML-S for P2P Discovery”, in *Proceedings of the First International Conference on Web Services* (ICWS'03), Las Vegas, USA, June 2003, pp 203- 207.
68. M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara, “Untangling the Broker Paradox in OWL-S”, in *Proceedings of AAAI Spring Symposium on Semantic Web Services*, 2004.
69. M. Paolucci, N. Srinivasan and K. Sycara, “Expressing WSMO Mediators in OWL-S”, in *Proceedings of the Semantic Web Services Workshop (SWS2004)* at the Third International Semantic Web Conference (ISWC 2004).
70. P. F. Patel-Schneider, “A Proposal for a SWRL Extension Towards First-Order Logic”, W3C Member Submission, April 2005, at http://www.w3.org/Submission/2005/SUBM-SWRL-FOL-20050411/.
71. E. Prud'hommeaux and A. Seaborne, “SPARQL Query Language for RDF”, at http://www.w3.org/TR/rdf-sparql-query/2005/.
72. J. Rao, P. Kungas, and M. Matskin, “Composition of Semantic Web services using Linear Logic theorem proving”, *Information Systems Journal - Special Issue on the Semantic Web and Web Services*, 2004.
73. D. Roman, H. Lausen, U. Keller, “Web Service Modeling Ontology”, at http://www.wsmo.org/TR/d2/v1.1/#mediators.
74. The Rule Markup Initiative, at http://www.dfki.uni-kl.de/ruleml/.
75. J. Scicluna, C. Abela, and M. Montebello, “Visual Modelling of OWL-S Services”, in *Proceedings of the IADIS International Conference WWW/Internet* (ICWI2004), Madrid, Spain, October 2004.
76. Semantic Web Services Interest Group (a W3C activity) home page, at http://www.w3.org/2002/ws/swsig/.
77. Semantic Web Services Framework, version 1.0, at http://www.daml.org/services/swsf/1.0/.
78. M. Sheshagiri, M. desJardins, and T. Finin, “A Planner for Composing Services Described in DAML-S”, in the *Proceedings of AAMAS’03 Workshop on Web Services and Agent-based Engineering*, 2003.
79. 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.
80. 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.
81. E. Sirin and B. Parsia, “The OWL-S Java API”, Poster, in *Proceedings of the 3rd International Semantic Web Conference* (ISWC2004), Hiroshima, Japan, November 2004.

82. N. Srinivasan, M. Paolucci, K. Sycara, "An Efficient Algorithm for OWL-S Based Semantic Search in UDDI", SWSWPC 2004: 96-110.
83. N. Srinivasan, M. Paolucci, and K. Sycara, "CODE: A Development Environment for OWL-S Web services", at http://projects.semwebcentral.org/projects/owl-s-ide/
84. P. Traverso and M. Pistore, "Automated Composition of Semantic Web Services into Executable Processes", in *Proceedings of the 3rd International Semantic Web Conference* (ISWC2004), Hiroshima, Japan, Nov. 2004.
85. "The Universal Description, Discovery and Integration (UDDI) Protocol", Version 3, 2003, at http://www.uddi.org/.
86. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, S. Aitken, "KAoS Policy Management for Semantic Web Services", *IEEE Intelligent Systems* 19(4): 32-41, July 2004.
87. Web Services Choreography Working Group home page, at http://www.w3.org/2002/ws/chor/.
88. Web Services Description Working Group home page, at http://www.w3.org/2002/ws/desc/.
89. "Web Services Architecture" (Working Group Note), February 2004, at http://www.w3.org/TR/ws-arch/.
90. Web Service Modeling Ontology (WSMO) Web site, at http://www.wsmo.org/.
91. H. Wong and K. Sycara, "A Taxonomy of Middle-Agents for the Internet," *Proc. 5th Int'l Conf. Multi-Agent Systems (ICMAS 2000)*, AAAI Press, 2000, pp. 465–466.
92. C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood, "A Suite of DAML+OIL ontologies to describe Bioinformatics Web Services and Data", in *International Journal of Cooperative Information Systems*, 12(2): 197–224, 2003.
93. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, "Automating DAML-S Web Services Composition Using SHOP2", in *Proceedings of the Second International Semantic Web Conference* (ISWC2003), 2003.

Appendix B - Adding OWL-S to UDDI: Implementation and Throughput

Naveen Srinivasan, Massimo Paolucci and Katia Sycara

Robotics Institute, Carnegie Mellon University, USA

***Abstract* – The increasing availability of web services demands a discovery mechanism to find the service that satisfies our requirement. The classification-based and syntax based search provided UDDI produces results that are coarse in nature. We need a discovery mechanism that allows us to search for our requirements with finer granularity. OWL-S allows us to semantically describe the web services in terms of capability they offer. We propose OWL-S/UDDI matchmaker that combines the better of two technologies. We have also implemented and analyzed its performance.**

Index Terms - UDDI, OWL-S, Capability Matching

INTRODUCTION

Web Services have changed the Web from a database of static document to e-business marketplace. Web Service technology are being adapted by Business-to-Business interaction and even in some Business-to-Consumer interactions. The widespread adoption of web services is mainly due to its simplicity and the data interoperability provided by its infrastructural components namely XML [7], SOAP [10] and WSDL [11].

With the proliferation of Web Services, it is becoming increasingly difficult to find the appropriate web service that will satisfy our requirements. Universal Description, Discovery and Integration [8] (here after UDDI) is the industry standard developed to solve the web service discovery problem. UDDI is a registry that allows businesses to describe and register the web services that they provide. It also allows businesses to discover the services that fit their requirement and to integrate them with their business component.

While UDDI has many features that make it an appealing registry for Web services, its discovery mechanism has two crucial limitations. First limitation of the UDDI is its search mechanism. In UDDI a web service can describe its functionality using a classification schemes like NAISC, UNSPSC etc. For example, a Domestic Air Cargo Transport Service can use the UNSPSC code 78.10.15.01.00 to describe it functionality. Although we can discover web services using the classification mechanism, the search would yield results that are coarse in nature with high precision and recall errors. The second shortcoming of UDDI is the usage of XML to describe its data model. UDDI guarantees syntactic interoperability, but it fails to provide a semantic description of its content. Therefore, two identical XML descriptions may have very different meaning, and conversely, two different descriptions describe the same features of the same object. Hence, the XML data is not machine understandable. XML's lack of explicit semantics proves to be an additional obstacle to UDDI's discovery mechanism.

The semantic Web initiative [5] addresses the problem of XML lack of semantics by generating a set of XML based languages, such as RDF and OWL, which rely on ontologies that explicitly specify the content of the tags used in the documents. In this paper, we adopt a Web service description based on OWL-S [3], an ontology in OWL [15] that can be used to describe the capability of web services. Like UDDI, OWL-S allows to associate classification schemes to describe the functionality of web services. In addition, OWL-S also provides a capability-based description mechanism [6] to describe web services. Using capability base description we can express the functionality of web services in terms of inputs and precondition they require and outputs and effects they produce. Capability-based search will overcome the limitations of the UDDI discovery mechanism and would yield results that are less coarse in nature.

In this paper we provide a detailed discussion on an implementation of the OWL-S/UDDI Matchmaker [2] that takes advantage of UDDI's proliferation in web service technology infrastructure and OWL-S's explicit capability representation. Specifically, we provided a mapping between OWL-S profile and UDDI data model based on [1], so that we can embed OWL-S profile descriptions inside UDDI descriptions and publish them in any UDDI registry. We have also enhanced the UDDI registry with a matchmaker module that can process the embedded OWL-S description.

In rest of the paper, we first briefly explain the OWL-S in particularly OWL-S profile. In section 3 we describe UDDI and its TModel mechanism in details. In the following section we describe the mapping between OWL-S profile and UDDI. In section 5 we describe the capability-matching algorithm followed by architecture and implementation details of OWL-S Matchmaker. In Section 6 we present experimental results comparing the performances of our OWL-S/UDDI Matchmaker implementation with a standard UDDI registry and finally we conclude.

OWL-S

OWL-S is ontology, based on OWL, to semantically describe web services. OWL-S is characterized by three modules: a *Service Profile* which describes capability of web service, a *Process Model* that describes the details of working of services and finally a *Grounding* that maps the data exchanged in the Process Model to WSDL operations and SOAP messages.

Since the Service Profile provides a description of the capabilities of Web services, it provides to be crucial for Web service discovery. OWL-S Profile describes various aspects of a web service, namely *service provider information, functional attributes* and *functionality description*. The provider information in the OWL-S Profile are defined through the Actor ontology that represents the service provider's contact information. OWL-S Profile's Functional Attributes can be used to augment description of the web services. Some functional attributes defined in OWL-S Profile are Quality Rating, to assign a rating like Dun & Bradstreet rating to a service, Geographic Radius, to describe the geographical constraints of the service, and Service Category, to categorize the service using classification schemes like NAISC, UNSPSC etc. For example, the Geographic Radius Functional Attribute can be used by the provider of ABC book buying service, whose is not willing to ship book outside US, to constrain the delivery area of her service to US. In so doing she can prevent a requester from India from ordering a book from ABC book buying service.

Functional Description is used to describe the capabilities of the web services in terms of input, output, pre-condition and effects. The input and output describe the information transformation produced by the Web service. For example in ABC book-buying service, the inputs required by the service would be ISBN of the book and credit card information. The output of the book buying service may be a confirmation that the book order has been received and processed successfully. A Pre-Condition represents the conditions in the World that should be true in order for the service to be invoked successfully. In our book-buying example, a pre-condition of the service would be a valid credit card. An effect represents the actions that result as a consequence of executing the service. In our example, effects of the service would be charging of the credit card and changing of the ownership book.

UDDI

UDDI [8] is an industrial initiative aimed to create an Internet-wide network of registries of web services for enabling businesses to quickly, easily, and dynamically discover web services and interact with one another. UDDI infrastructure allows businesses to advertise their web services and also to search for web services. A business can use the UDDI data model to register their presence on the web by specifying the point of contact who can provide more information about the service, the way they conduct electronic business and other technical information like the location of the service in terms of IP address and the port number, thus enabling its business partners not only to discovery but also to invoke their web services automatically.

UDDI Representation

The representation of web services in UDDI is shown in Fig [1]. A business is represented as a *Business Entity* object that records information like name of the business, points of contact such as the physical address, telephone number, e-mail and fax number, the URL of the company web site, and so on. A Business Entity is associated with one or more *Business Services* that are descriptions of the specific services that a business provides. In turn a Business Service is associated with one or more *Binding Templates* that specify the service access end point. UDDI supports the specification of a wide range of binding specification that range from HTTP, to mail, to fax and others.

The description of businesses and services provided above supports the description of the basic information of services: how they are named, who to contact to gain information, how to invoke them. But it does not provide any indication of what type of service have been defined or additional technical details about the service.

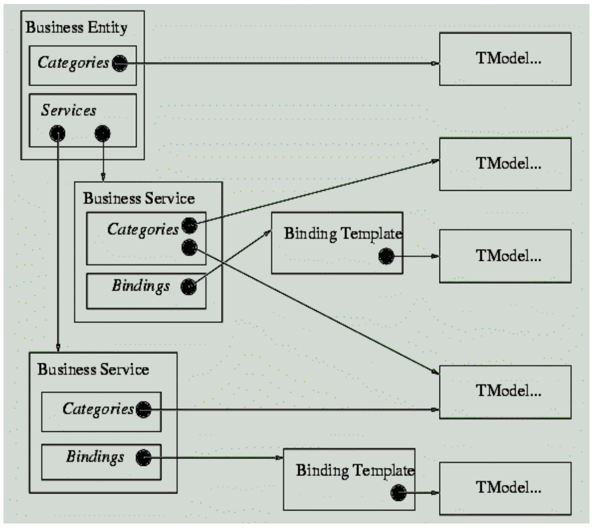


Fig 1 UDDI Service Representation

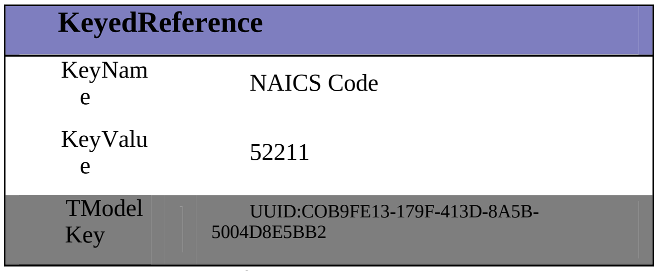
In addition to the description of businesses, services and binding templates, UDDI provides a data structure called *TModel* that allows the specification of additional attributes of the entities described in the UDDI repository. An example of a TModel is provided in Fig [2], the example shows the different fields of a TModel: *Name* is an identifier of the TModel, a unique *Key* assigned to the TModel in form a UUID [12], an text *Description* of a TModel, *an Overview URL* where a more detailed text description of the TModel can be found; finally, the *Technical Model Locators* specify information that is used within UDDI to control the use of the TModel.

Technical Model Information	
Name	ntis-gov:naics:1997
Key	UUID:COB9FE13-179F-413D-8A5B-5004D8E5BB2
Technical Model Description(s) :	
Business Taxonomy: NAICS (1997 Release)	
This tModel defines the NAICS industry taxonomy	
Overview URL	http://www.uddi.org/taxonomies/Core_Taxonomy_OvervierDoc.html#NAICS
Technical Model Locator(s) :	
Code	categorization
Description	types
Type	UDDITYPE

Fig 2 TModel for NAISC Category

TModels define attributes that can be used to specify information about the services. Fig [3] shows how the NAICS TModel may be used to specify the classification of a Commercial Banking service. The *KeyName* in the figure is a name of the attribute, the *KeyValue* is the value is the code of Commercial Banking services within the NAICS classification, and *TModelKey* is the key of the TModel used, in this case the TModel defined in Fig [3].

UDDI supports two types of TModels: the first type are TModels that express technical specifications of the services such as the protocols that they adhere to or interchange formats such as the Rosetta Net Partner Interface Processes specification [9]. The second type of TModels express abstract specifications about the service within predefined classification and taxonomic schemes, as the example of the NAICS TModel above. In this case, a service can specify its position within the general classification scheme.



KeyedReference	
KeyName	NAICS Code
KeyValue	52211
TModel Key	UUID:COB9FE13-179F-413D-8A5B-5004D8E5BB2

Fig 3 TModel for a Commercial Banking Service

UDDI allows a wide range of searches of the registry: services can be searched by name, by location, by business, by bindings or by TModels. For example it is possible to look for all the services that have a WSDL representation, or for all the services that adhere to the specification of the Rosetta Net. Unfortunately, the search mechanism supported by UDDI is limited to keyword matches and UDDI does not support any inference based on the taxonomies referred to by the TModels. For example a banking service may describe itself as “Commercial Banking” which is a valid entry in NAICS, but any search based for “Depository Credit Intermediation” services will not identify the banking service despite the fact that “Commercial Banking” is a subtype of “Depository Credit Intermediation”.

MAPPING OWL-S INTO UDDI

The goal of this work is to make use of OWL-S in UDDI, in doing this, it is important to avoid breaking UDDI. Instead we try to exploit the UDDI own mechanism, in particular TModels, to map OWL-S concepts in the UDDI data structure. Here we propose a mapping between OWL-S Profile elements and UDDI Data model that is provided to embed OWL-S Profile into UDDI. Fig [4] shows the complete mapping between OWL-S 1.0 Profile and UDDI Data Model. As we can see in Fig [4], some of the OWL-S Profile elements can be directly mapped to UDDI elements. For example, we can map the contact information in the OWL-S Profile and Business Service’s contact information in the UDDI data model directly. However, for many OWL-S Profile elements, like input, output, service parameters and so on, there are no corresponding UDDI descriptions. We use a TModel mechanism for representing the unmapped

Profile attributes in UDDI descriptions.

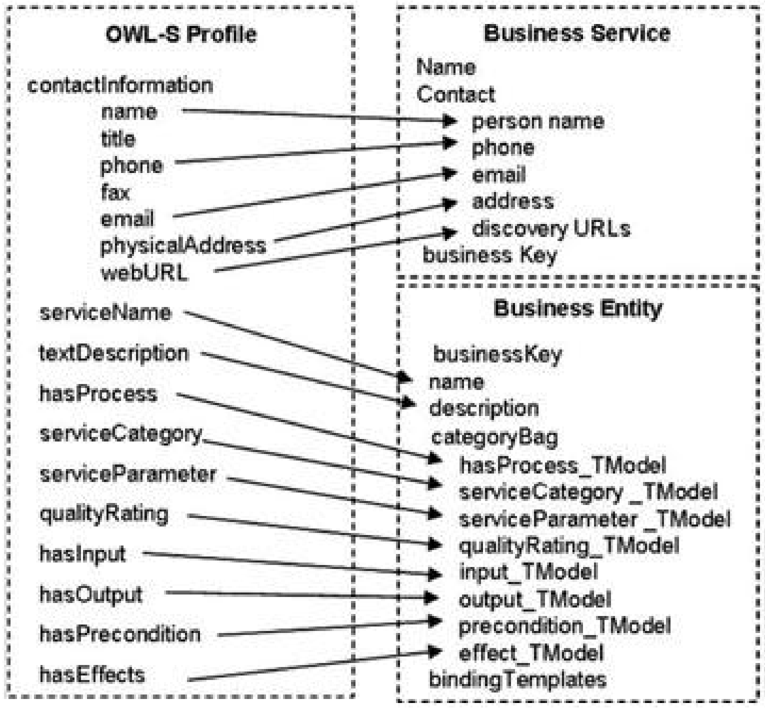


Figure 4. Mapping OWL-S to UDDI

The mechanism we use is loosely based on the WSDL-to-UDDI mapping proposed by the OASIS committee [13]. We define specialized UDDI TModels for each unmapped elements in OWL-S Profile. For OWL-S Profile we defined 8 specialized TModels for OWL-S Input, Output, Service Parameter and so on. We will use these specialized TModel, similar to the way we used NAISC TModel, for mapping the OWL-S profile elements to UDDI. For example, let us consider a Stock Quote reporting service which takes a ticker symbol of a company as an input and returns the latest quote. Fig [5] shows a mapping of the stock quote service's input and output to its equivalent UDDI Data Models.

CategoryBag	
KeyedReference	
KeyName	Ticker Input
KeyValue	financialOntology:Ticker
TModelKey	UUID of OWL-S Input TModel
KeyedReference	
KeyName	Quote Output
KeyValue	financialOntology:quote
TModelKey	UUID of OWL-S Output TModel

Fig 4 TModel for Stock Quote Service

The UDDI generated from the mapping can be published and queried in a UDDI registry. While processing a query, the UDDI registry can treat these specialized OWL-S TModel keys, like other TModels and perform a syntactic search. However, a syntactic search does not take advantage of the semantic data that is used to describe the capability of the service. We need to enhance the UDDI registry to process the semantic information.

OWL-S / UDDI MATCHMAKER ARCHITECTURE

We have augmented the UDDI registry with an OWL-S Matchmaking component to perform semantic based capability matching on advertisements containing OWL-S profile information. Fig [6] shows the architecture of the OWL-S/UDDI Matchmaker.

MATCHING ALGORITHM

The matching algorithm we used in our matchmaker is based on the algorithm presented in [2]. This algorithm attempts to satisfy two requirements; the first one is that the matching process should exploit the ontologies available on the semantic Web. The second requirement is that in the real world, it is would be difficult to find advertisements that exactly match the request. Hence, the algorithm defines a more flexible matching mechanism based on the OWL's subsumption mechanism. When a user submits a request, the algorithm finds the appropriate service as follows, for each advertisement in the repository; first it matches the outputs of the request against the outputs of the advertisement. If the advertisements are found after the output match, the inputs of the request are matched against inputs of the matched advertisements.

In the above algorithm, the degree of match between two outputs or two inputs depends of the match between the concepts that represented by them. The matching between the concepts representing the input and output is not syntactic, but a semantic in nature. For example consider an advertisement, of a vehicle selling service, whose output is specified as Vehicle and a request, by a person seeking for a car selling service, whose output is specified as Car. Although there is no exact match between the outputs of the request and the advertisement, given an ontology fragment as show in Fig [7], the matching algorithm recognizes a match between the request and advertisement because the concept Vehicle subsumes the concept Car. This matching is valid under the assumption that a vehicle selling service may sell a car because car is a type of vehicle.

The matching algorithm recognizes four degrees of match between two concepts. Let us assume OutR represents the concepts of an output of a request, and OutA that of an advertisement. The different degree of match between OutR and OutA is calculated in the following way.

exact: If OutR and OutA are equivalent or if OutR is a immediate subclass of OutA. For example given the ontology fragment Fig [7], the degree of match between a request whose output is Sedan and an advertisement whose output is Car is exact.

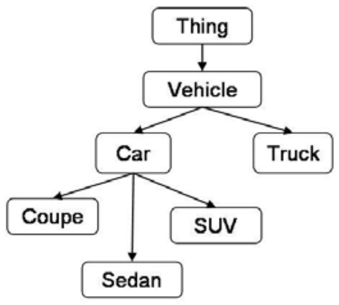


Fig 5 A fragment of the Vehicle Ontology

plug in: If OutA subsumes OutR, then OutA is assumed to set that encompass OutR or in other words OutA can be plugged instead of OutR. For example we can assume a service-selling Vehicle would also sell Car. However this match is considered to be inferior to exact match.

subsume: If OutR subsumes OutA, then the provider may or may not completely satisfy the requester. Hence this match is inferior than plug in match.

fail: A match is recognized as a failure if there is no subsumption relation between OutA and OutR.

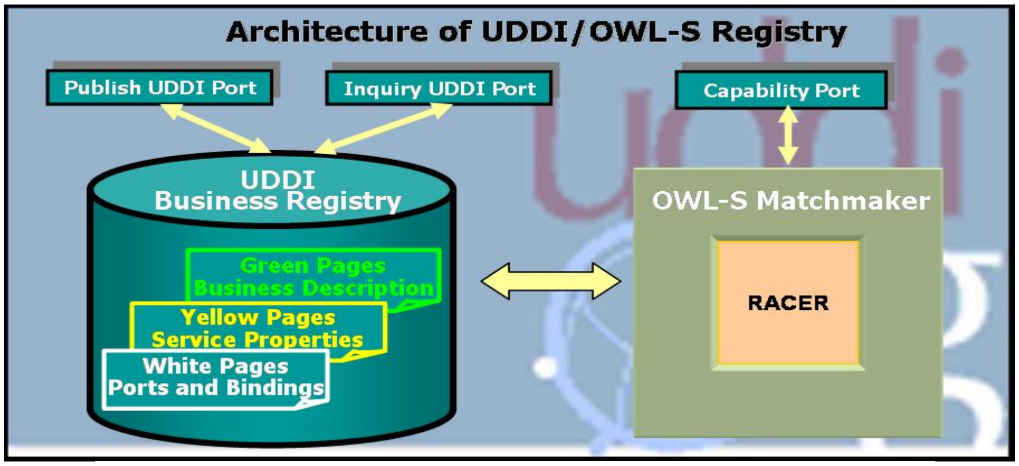

Architecture of UDDI/OWL-S Registry
Publish UDDI Port
Inquiry UDDI Port
Capability Port
UDDI Business Registry
Green Pages Business Description
Yellow Pages Service Properties
White Pages Ports and Bindings
OWL-S Matchmaker
RACER

Fig 4 Architecture of OWL-S / UDDI Matchmaker

IMPLEMENTATION

The architecture of the OWL-S / UDDI Matchmaker is shown Fig [6]. The OWL-S matchmaking component is embedded into jUDDI, an open source UDDI registry [14]. The matchmaker and jUDDI can access each other functionalities. We use Racer DL engine [4], description logic reasoning system, to process semantic data.

The UDDI registry, on receiving an UDDI advertisement containing OWL-S Profile for publishing, processes the advertisements and forwards the advertisement to the matchmaking component. The matchmaker first validates the correctness of the advertisement using the Racer DL engine. After validation, the advertisement is classified based on its semantic information using Racer and stored in the repository. On receiving a query, the matchmaker validates the query, similar to the validation performed on an advertisement, and then matched across all the published advertisements using the algorithm explained in Section 5.1.

The naïve way to implement a matching algorithm, the inputs and the outputs of the request are matched against the inputs and the outputs of all the advertisements that are present in the repository. As the number of advertisements in the repository increases the time to process each query will also increase. To overcome this limitation in our implementation, we perform most of its matching reasoning during the publishing the advertisements itself and store those results to be used during request time. The rational underlying our approach is that since the publishing of an advertisement is a one-time event, it makes sense to spend time to process it and store partial results and speed up the query processing time, which may occur many times and furthermore it is time critical.

PUBLISHING

In order to maintain information about the published advertisements, the matchmaker maintains a hierarchical tree structure that represents the subsumption relationships between all the concepts that form the inputs and outputs of all the advertisements published in the repository. Each node in the hierarchical structure represents a concept present in the matchmaker; it also maintains information about degree of match between each output of all the advertisements and the concept it represents. If degree of match between the node's concept and the output of an advertisement is fail, then the information is not stored. Similarly, each node maintains the degree of match between all the inputs of the advertisements and the concept it represents.

When an advertisement submitted, the matchmaker loads the ontologies that are used by the advertisement's inputs and outputs into the Racer system and updates its hierarchical tree. The Racer system provides a network API to access its information, which is used to construct and update the hierarchical tree structure.

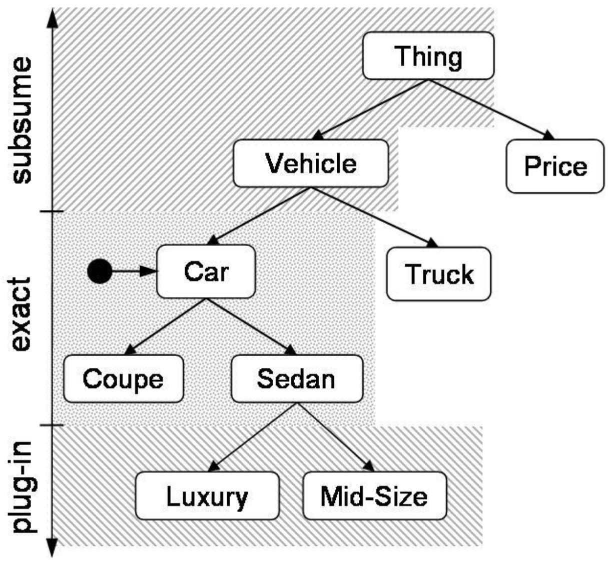


Fig 6 Advertisement Propagation

For each output of the advertisement, the matchmaker extracts the concept that represents the output and locates the corresponding node in the hierarchical data structure. The degree of match between this node's concept and the output of the advertisement is exact, so the matchmaker updates the node with this information. For example, let us assume that the matchmaker maintains a hierarchical tree as shown in Fig [8] and let an output of an advertisement be Car. The matchmaker updates the information of the Car node that it matches the advertisement exactly. According to the algorithm, the degree of match between output and the concepts immediate subclass are also exact, so the matchmaker updates the node information of all the nodes that are immediate child of the current node. In our example the matchmaker updates the node information of Coupe and Sedan that it matches the advertisement exactly.

We can also observe that the degree of match between the output and the concepts of all the parent nodes of current node is subsume. In our example the nodes Thing and Vehicle subsumes an advertisement whose output is Car. The matchmaker updates the node information of all the parents of the current node that the degree of match between the node and the advertisement is subsume. Similarly we can observe that the degree of match between the output and the concepts of all the child nodes, except the immediate child nodes, is plug in. Following our example the degree of match between concept Luxury and an advertisement, whose output is Car, is plug in. The matchmaker performs similar updates to the hierarchical tree for all the outputs and inputs of the advertisement.

During the publishing phase we are performing most of the work required by the matching algorithm, hence we may spend a considerable amount of time in this phase. But we can show that time spend during this phase, does not depend linearly on the number of concepts present in the data structure but in the order of log of concepts in the data structure, and hence showing that our implementation is scalable. Since we use hierarchical data structure, the time required to insert a node will be in the order of $\log_d N$, where d is the degree of tree. Similarly time required to traverse between any two nodes in a particular branch will also be in the order of $\log_d N$.

The time required for publishing an advertisement will be equal to the time required by Racer for classification of the ontologies used by inputs and outputs of the advertisement, plus the time required to update the hierarchical structure with the newly added concepts, plus the time required to propagate information about the newly added advertisement to the hierarchical structure. And in a best case scenario, when no ontology needs to loaded, it will be equal to the time required for publishing will be time for updating plus the time for propagating.

$$\text{Time}_{\text{publish}} = \text{Time}_{\text{Classification}} + \text{Time}_{\text{Update}} + \text{Time}_{\text{propagate}}$$

The time required by Racer for classification is neither depended on the number of concepts nor the number of advertisements present in the matchmaker. The time required by the other two operations, update and propagate, will in the Order ($\log_d N$). Hence the publishing time is not increased linearly by the number of concepts present in the matchmaker. We can also notice that the publishing time does not depend on the number of advertisements in the matchmaker.

QUERYING

Since most of the matching information is pre-computed at advertisement time, the matchmaker's query phase is reduced to simple lookups in the hierarchical data structure for all the outputs and inputs of the request. We also save time by not allowing the query to load ontologies. Although loading ontologies required by the query appears to be a good idea, we do not allow it for the following reasons: Firstly, the ontologies loaded by the query may be used only one time, and over time we may result is storing information about lot of unused concepts. Secondly if we load a new ontology in the matchmaker it is likely that new ontology may not be related to ontologies already present in the matchmaker. Finally, we have to deal with problem of trust; an ontology required by the query may change the subsumption relationships between existing concepts. For example the loaded ontology may insert relationships that would result in knowledge base, with a statement: the concept apple is equivalent to concept orange. Hence loading a malicious ontology will result in producing erroneous results for subsequent queries.

When the matchmaker receives a query, it retrieves all the sets of advertisements that match each output of the request. For example, if the outputs of the request are Car and Price, the matchmaker fetches the information, that mentions the degree of match of between the nodes, in this case car and price, and the outputs of the advertisements. The matchmaker then finds the advertisements that are common between the sets of advertisements retrieved. If no intersection is found then the query fails. If common advertisements are found, they are selected for further processing.

The matchmaker performs a lookup operation and fetches the sets of advertisements for the inputs of the request, similar to the one it performed for the outputs. However, instead of find intersection between the sets of advertisements, the matchmaker only keeps the information about advertisements that were selected during the output-processing phase. The resulting advertisements that matched the request's inputs are used to score the advertisements that were selected during the output-processing phase.

We can show that time required processing a query does not depends on the number of advertisements published in the matchmaker. As we can see the querying phase involves lookups and intersections between the selected advertisements. In our implementation, lookups can perform in constant time. Hence time to process a query depends on the time to perform the intersection operations between the selected advertisements.

$$\text{Time}_{\text{query}} = \text{Time}_{\text{Lookup}} + \text{Time}_{\text{Intersection\_seladv}}$$

If we implement the original algorithm as suggested in [2], then the request is matched against all the published advertisements. The time required by for processing a query will be equal time required to match inputs and outputs of the request with all the advertisement, plus time required for the intersection operation.

$$\text{Time}_{\text{query}} = N * \text{Time}_{\text{Match}} + \text{Time}_{\text{Intersection\_seladv}}$$

where Time $_{\text{Match}}$ is time required to find degree of match between two concepts. From the above equation we can observe that the query processing time depends linearly on the number of advertisements in the repository.

EXPERIMENTAL RESULTS

We conducted some preliminary experiments comparing the performance of our OWL-S/UDDI matchmaker and a UDDI registry. We will provide a more elaborate experimental result in the final version of the paper. We have used apache’s jUDDI, an open source UDDI registry for our matchmaker’s implement.

In our experiments, we calculate the processing time of an advertisement by calculating difference time difference between the time the registry received the advertisement and the time the result is delivered. We use this approach over measuring the time in the client code because it eliminates the network latency time due to the client server communication.

PERFORMANCE - TIME

In our first experiment we compared the time take to publish an advertisement in an OWL-S/UDDI matchmaker and in an UDDI registry. In this experiment we assumed that ontologies required by the inputs and outputs of an advertisements are already present in the matchmaker. Table [1] shows the average time taken to publish 50 advertisements.

	Time in ms	Standard Deviation
UDDI	163.98	86.17
OWL-S/UDDI Matchmaker	1050.77	167.96

Table 1 Publishing Time without loading ontologies

The time taken by OWL-S/UDDI registry is time taken to publish the advertisement in the UDDI registry plus the time taken by matchmaker component to process the result. Fig 9 shows the time distribution of various activates during publishing of an advertisement in OWL-S/UDDI matchmaker. As you can see 70% of the time is spend in loading and validating the advertisement. We can also see that 15% of the time is taken to update the hierarchical tree structure maintained by the matchmaker. We could drastically reduce the processing time of publishing an advertisement if we had direct access to Racer and it components.

PERFORMANCE – ONTOLOGY LOADING

In the second experiment, we analyzed the performance of our matchmaker when we publish advertisements that require the matchmaker to load new ontology. In our experiment, we published 50 advertisements in our OWL-S/UDDI matchmaker and measure the time taken to publish each advertisement. All these 50 advertisements uses different ontologies to describe the inputs and outputs of the advertisement, hence it require that matchmaker to load complete different ontologies for every advertisement. Each of these advertisements has three inputs and one output and requires loading an ontology containing 30 concepts.

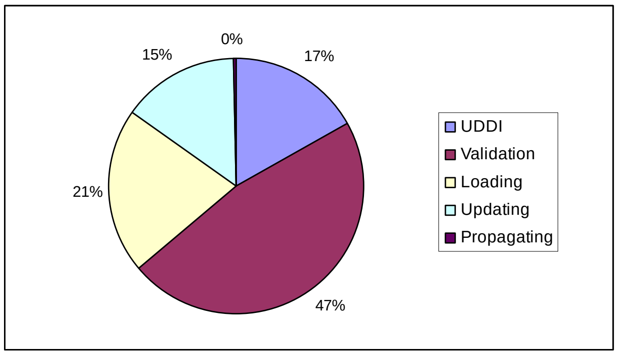


Fig 7 Time distribution during publishing an advertisement

Fig 10 shows the time taken to publish each of the 50 advertisements. Although the time take to publish an advertisement is increase linearly with the number of advertisements, the linear increase is mainly due to a limitation of the Racer system. Whenever we load a new ontology into the Racer we have determine if we need to update the hierarchical structure maintained by the matchmaker, if so, what concepts should be updated. The Racer system does not provide any direct means to find out this information. Hence we need to find out this information through a series of interactions. The new-concept line in the Fig [10] represents the time required to perform this operation, which contributes for linear increase of the publishing time. We can eliminate the time required for this process if either the Racer can provide the information directly or if we could have direct access to the Racer System.

PERFORMANCE – QUERYING

In our final experiment, we calculated time taken to process a query. The queries we used do not load new ontology into the matchmaker, they use the ontologies that are already present in the matchmaker. We used 50 queries each with three inputs and one output. Table 2 shows the

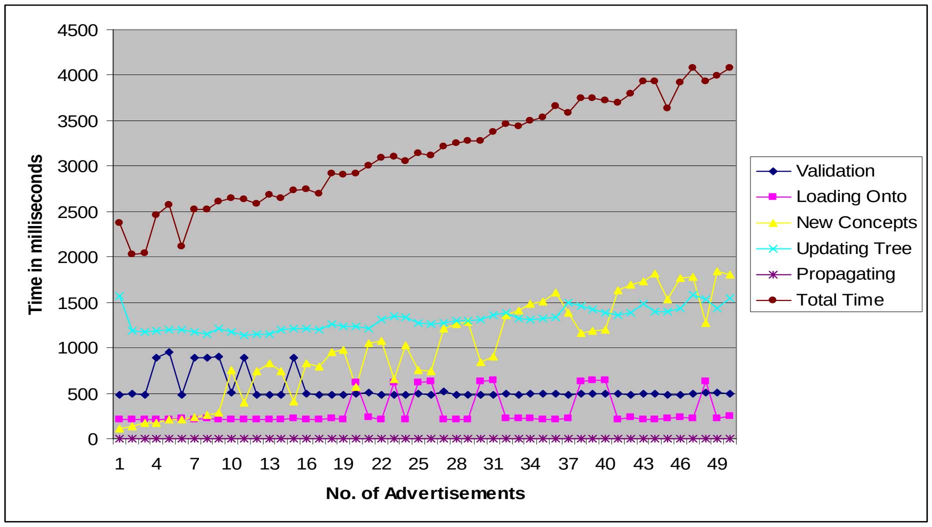


Fig 7 Publishing time for advertises that required loading ontology

average time required to process 50 queries. We can see that the time required to process the query is almost constant supporting our analyzes in section 5.2

	Time in ms	Standard Deviation
OWL-S/UDDI Matchmaker	1.306	.54

Table 2 Query processing time

CONCLUSION

In this paper we have described the importance of web service discovery and the shortcoming of the UDDI, the current web service discovery mechanism. We proposed a solution to use OWL-S, a more expressive capability-based web service description language, in combination with UDDI and OWL-S to take advantage of both these technologies. We also proposed a mapping to embed OWL-S Profile descriptions inside UDDI Data model. Then we discussed about our matching algorithm and architecture of our OWL-S/UDDI matchmaker. We also analyzed the scalability of our implementation against a more traditional approach proposed. Finally we presented experimental results supporting our analysis and comparing the performance with a UDDI registry.

BIBLIOGRAPHY

[1] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara; "Importing the Semantic Web in UDDI". In Proceedings of Web Services, E-business and Semantic Web Workshop, 2002.

[2] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara; "Semantic Matching of Web Services Capabilities." In Proceedings of the 1st International Semantic Web Conference (ISWC2002).

[3] Anupriya et al; "DAML-S: Web Service Description for the Semantic Web" In Proceedings of The First International Semantic Web Conference (ISWC), 2002.

[4] Volker Haarslev, Ralf Möller; "RACER System Description" In Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.

[5] Tim Berners-Lee and James Hendler and Ora Lassila, "The Semantic Web", Scientific American, volume 284, Number 5, pages 34-43, 2001.

[6] Katia Sycara et al, "Larks, Dynamic matchmaking among Heterogeneous Software Agents in Cyberspace", AAMAS, 5, 173-203, 2002.

[7] W3C, "Extensible Markup Language (XML) 1.0 (Second Edition)", http://www.w3.org/TR/2000/REC-xml-20001006, 2000.

[8] UDDI, "The UDDI Technical White Paper", http://www.uddi.org, 2000.

[9] RosettaNet,http://www.rosettanet.org, 2000.

[10] W3C, "SOAP Version 1.2, W3C Working Draft 17 December 2000", http://www.w3.org/TR/2001/WD-soap12-part0-20011217/, 2001.

[11] Erik Christensen and Francisco Curbera and Greg Meredith and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1", http://www.w3.org/ TR/2001/NOTE-wsdl-20010315, 2001.

[12] ISO/IEC 11578:1996, "Information technology -- Open Systems Interconnection -- Remote Procedure Call", http://www.iso.ch/, 2001.

[13] John Colgrave and Karsten Januszewski, "Using WSDL in a UDDI Registry, Version 2.0", http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v200-20031104.htm, OASIS UDDI Technical Note, 2003.

[14] jUDDI, http://ws.apache.org/juddi/.

[15] W3C, "Web Ontology Language", http://www.w3. org/2001/sw/WebOnt/.

Appendix C-CODE: A Development Environment for OWL-S Web services

Naveen Srinivasan[1], Massimo Paolucci[1], and Katia Sycara[1]

[1] The Robotics Institute, Carnegie Mellon University,

Pittsburgh, PA 15213, USA

{naveen, paolucci, Katia}@cs.cmu.edu

Abstract. The generation of Semantic Web services is a complex and error prone process. CODE, the system that we present here, is an Integrated Development Environment that supports the developer through the whole process from the Java generation, to the compilation of OWL-S descriptions to the deployment and registration with UDDI.

1 Introduction

Web services are becoming an effective paradigm of distributed computation over the Intranets and the all Internet. Furthermore, the interest in Semantic Web services is also growing with industry standards such as EBXML and UDDI interested in using OWL in their standards. Within the Semantic Web activity, many important aspects of using semantics within a Web services framework received wide attention. For example, OWL-S [8] stresses the goal directed approach to discovery and composition of Web services, whereas the WSMO framework stresses the importance of mediation. One aspect that has not been analyzed yet is the difficulty of using Semantic Web services languages to describe Web services.

In this paper we analyze this problem from OWL-S viewpoint. The compilation of an OWL-S description of a Web service requires many different types of information and many different activities such as the actual implementation of the Web service; the compilation of the WSDL description; the compilation of the OWL-S Profile, Process Model, and Grounding; the specification of the semantics of all inputs and outputs and their mappings to XML schemata representing the data that flow over the wire. More importantly, the results of all these activities are strictly related: the Profile should represent the capabilities of the Web service, the Process Model should be faithful to the implementation of the Web service, the Grounding should provide a consistent mapping between OWL-S and WSDL, and finally the Web service implementation should be bug free. As a consequence, the compilation of an OWL-S description is very time consuming and error prone, and the few tools that are available to support the developer do not form a consistent suite, therefore, they are very difficult to use on a consistent basis.

CODE (CMU's OWL-S Development Environment) addresses the problems of the developer by providing a uniform integrated development environment. CODE supports the developer through the whole generation process, from the Java development to the generation of the OWL-S descriptions, to the deployment and registration of the Web service with UDDI. Furthermore, through its editing facilities, CODE guarantees the syntactic correctness of the service description, and it allows the developer to use the SPIN model checker [11] to verify correctness claims about the control flow of the OWL-S Process Model. As a result CODE helps the developer to detect problems at development and compilation time, reducing the likelihood of execution time errors.

The guiding principle of the design of CODE is to integrate the tools that the developer needs during the implementation, description and deployment of Semantic Web services, in a single consistent and extensible environment. The consistency of the development tools allows the developer to move seamlessly between the different aspects of Semantic Web services development, while the extensibility of the environment allows other parties to provide additional contributions.

To realize this vision, we implemented CODE as an Eclipse [12] plug-in. Eclipse is known primarily a Java IDE that supports the programmer in during code development. But Eclipse goes beyond Java. Eclipse is an open platform for the integration of different tools that the community is developing. Using Eclipse and its many plug-ins, it is possible to deign a system using UML, implement details of the system that have not been modeled in UML, and deploy the system, all in one coherent uniform environment. CODE contributes to this vision by providing an extension of the development process that supports the developer to compile OWL-S descriptions of the Web service. Ultimately, the objective of CODE is to provide a uniform environment that supports the developer from the design of the system, to its implementation, to its semantic description, to its deployment.

The rest of the paper is organized as follows: Section 2 describes the overview of OWL-S. In section 3, we describe in details the life cycle of the development of semantic web services and how CODE supports through out the development process. In Section 4 we show how CODE also be used by a client to consume the semantic web services generated using the above process. Section 5 illustrates an use case followed by conclusion.

2 OWL-S Overview

OWL-S is a Web Services description language that enriches Web Services descriptions based on WSDL with semantic information from OWL [13] ontologies. OWL-S is organized in three modules: a Profile that describes capabilities of Web Services as well as additional features that help to describe the service; a Process Model that provides a description of the activity of the Web Service provider from which the Web Service requester can derive the interaction; a Grounding that is a description of how abstract information exchanges described in the Process Model is mapped onto actual messages that the provider and the requester exchange.

The OWL-S Profile describes the Web Service capabilities, as well as additional features of Web Services such as provenance and quality of cost specifications of the Web service. The role of the OWL-S Profile is to support different modalities of discovery and to support the requester decision of whether to use a given Web service. OWL-S describes capabilities of Web Services by the transformation that they produce. In addition to capabilities, OWL-S Profiles provides provenance information that describes the entity (person or company) that deployed the service; and non-functional parameters that describe features of the services such as quality rating for the service.

The second module of OWL-S is the Process Model; the Process Model specifies the interaction protocol in the sense that it allows the requester to know what information to send to the provider and what information will be sent by the provider at a given time during the transaction. A Process Model is defined as an ordered collection of processes, where each process produces a state transformation or a data exchange with the Web service clients.

The OWL-S Process Model distinguishes between two types of processes: composite processes and atomic processes. Atomic processes correspond to operations that the provider can perform directly. Composite processes are used to describe collections of processes (either atomic or composite) organized on the basis of some control flow structure. For example, a sequence of processes is defined as a composite process of type sequence. Similarly, a conditional statement (or choice as defined in OWL-S) is also a composite process. The OWL-S process model allows any type of control flow structure including loops, sequences, conditionals,

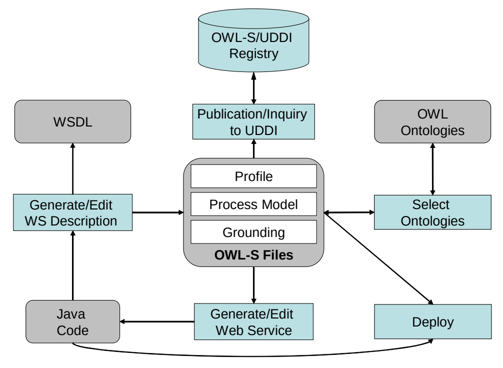


Figure 3: Activities workflow for the generation of OWL-S Web services

nondeterministic choice and concurrency. Apart from expressing the control flow between the processes, the process model can also be used to express the data being exchanged between the processes. For example, in a process model of an e-commerce website the user's identification information that may flow from the login process to other processes may be represented using the dataflow constructs.

The last module of OWL-S is the Grounding that describes how atomic processes; which provide abstract descriptions of the information exchanges with the requesters, are transformed into concrete messages that can be exchanged on the net, or through procedure call. Specifically, the OWL-S Grounding is defined as a one to one mapping from atomic processes to WSDL specifications of messages. From WSDL it inherits the definition of abstract message and binding, while the information that is used to compose the messages is extracted by the execution of the process model.

3 CODE

CODE aims at supporting the Web service developer through the whole lifecycle of development of the Web service. In this section we analyze in some detail the lifecycle of an OWL-S specification and how CODE supports the developer. Then we will analyze in greater details the implementation of the different modules of CODE.

3.1 Lifecycle of OWL-S Generation

The implementation of a Semantic Web service requires a number of very different activities. Figure 2 shows the relation between those activities and what they generate. Crucially there is no clear starting point to the diagram, but two obvious starting points are the Java code and the OWL-S specification. These two starting points define two different ways in which a developer may address the problem of describing a Web service. We call the first approach *code-driven*, and the second approach *model-driven.*

In the code-driven approach the Web service is implemented using Java or any other programming language, and the OWL-S description is derived from the code. Following this approach, the developer takes the following steps:

1. Develop the Java code of its implementation;

2. Generate the WSDL and OWL-S description of the Web service;
3. Use ontologies to enrich the Web service description;
4. Use the OWL-S description to publish the Web service with UDDI.

One natural use of this approach is to expose legacy systems through Web services. In such a case the most of the functionality will be already implemented and the developer needs only to write the code needed to expose those functionalities as Web services. Crucially, many of these four steps can be partially automated to ease the developer work and remove then danger of erroneous descriptions. For example, in step 2, the generation of the WSDL description is done automatically from the source code of the Web service with tools such as Axis [1] and .Net [6]; furthermore, the OWL-S description can be partially generated from WSDL [2]. Similarly, there exists a mapping from the OWL-S Profile to UDDI [10] that can be used to automate step 4.

The model-driven approach follows the opposite direction, rather than starting from the Web service code and ending in with the generation of the OWL-S description, the model-driven approach starts with the OWL-S specification of the functionalities that the developer expects from the Web service, and the specification of the Web service interaction process, and ends with a partial generation of the (Java) code of the Web service. Specifically, the generation process in this case is the following:

1. The OWL-S ontology is developed using OWL ontologies;
2. The OWL-S process model is used to implement the Web service;
3. The complete Web service implementation is used to generate WSDL;
4. OWL-S is used to publish a Web service description with UDDI.

The model-driven approach achieves the same level of automation of the code-driven approach, using essentially the same tools. The only exception is step 2, which requires the generation of Java code from OWL-S Process Model descriptions. This generation is done by generating a code-stub for each atomic process specified in the Atomic Process, and by imposing that the processes are executed in the sequence specified by the control flow of the OWL-S Process Model. The result of this process is the generation of a set of abstract classes that need to be instantiated by the Web service developer into concrete classes that perform the functionalities promised by the Web service. The advantage of this approach is that the Web service developer does not have to deal with the implementation of the interaction protocol of the Web service since that interaction protocol implemented automatically by the Web service generation module.

3.2 Using CODE to support OWL-S Generation

CODE supports both ways to generate an OWL-S Web service description to leave to the developer the freedom to select the most appropriate way to describe the Web service. Specifically, CODE supports the Web service developer to perform the following six operations

1. Edit the Web Service code through a close integration with the Eclipse plug-in for Java development;
2. Generate WSDL from the complete the Web service implementation;
3. Generate OWL-S description of the Web service from WSDL;
4. Develop the OWL-S ontology using OWL ontologies;
5. Automatically derive Web service code through he OWL-S Process Model;
6. Publish a Web service description with UDDI;
7. Deploy the Web service publishing the Web service code, WSDL and OWL-S descriptions on a Web server.

Figure 4 shows the architecture of CODE. The main data structure of CODE is, of course, the OWL-S Process Model, Profile and Grounding. They are represented internally using the *OWL-S API* module. The OWL-S API provides Java classes and methods to extract information from an OWL-S description or to generate an OWL-S description. In turn the OWL-S API is based on the Jena [5] OWL models and API. In turn, the OWL-S API is the bases for all the other modules that process OWL-S.

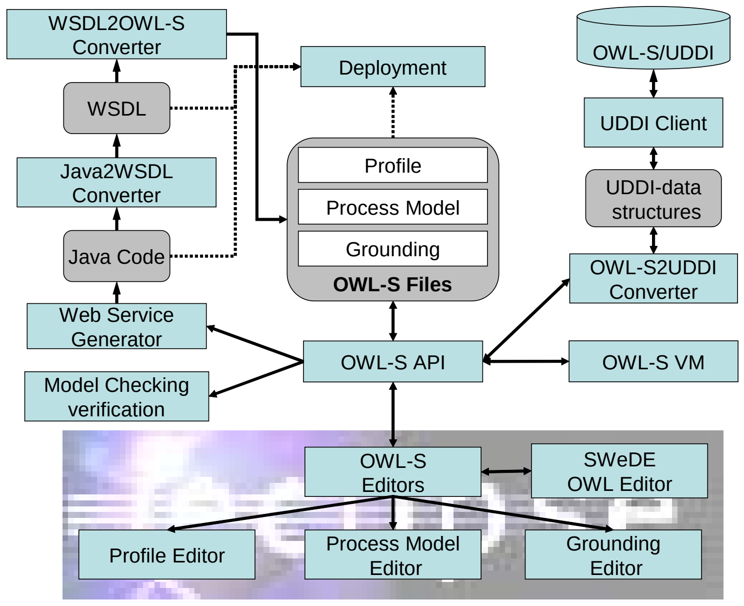


Figure 4: The architecture of CODE

Using the architecture is easy to show how these 7 functionalities are achieved. The first functionality, which is the editing of the Java code, is not really part of CODE, but it is provided in the Java plug-in of Eclipse and it is integrated with CODE so that the developer does not realize that she is moving between different plug-ins. The second and third functionalities are achieved through the Axis Java2WSDL converter and the WSDL2OWL-S converter. The results of these two steps are a complete WSDL description, and schematic Profile, Process Model and Grounding. Those schematic descriptions contain placeholders for atomic processes, mappings between atomic processes and WSDL operations and placeholders for inputs and outputs in the Profile. But, these descriptions are missing semantic descriptions of the inputs and outputs since WSDL does not provide any semantic description, and they are missing the control flow specification since WSDL does not impose any order on the invocation of operations. The implementation of the code-driven approach results from using these three functionalities.

The fourth functionality is to generate the OWL-S Web service description. This description can be generated either from scratch or by editing an existing OWL-S description. In the latter case, the editing process may complete the OWL-S descriptions that were generated using WSDL. The three editors at the bottom of Figure 4 are used to provide this functionality, with the optional assistance of the SWeDE [9] OWL plug-in for Eclipse. The result of this process is a complete OWL-S description of the Web service that specifies the semantics of all inputs and outputs, the preconditions and effects, and the complete control flow and data flow of the Web service. The fifth functionality is realized by the Web service Generator module. The combination of these last two functionalities supports the model-driven Web service generation.

The last two functionalities: publication with UDDI and deployment are also supported. The publication with UDDI results from the translation of OWL-S Profiles into UDDI Web service descriptions, which are then published into a UDDI registry using a UDDI client. The deployment is supported by packaging the Web service code appropriately and pushing it on a Web Server.

CODE supports two additional functionalities, in addition to the six functionalities reported above. The first one is the Model Checking verification that is implemented as a mapping between the OWL-S Process Model and the Promela language used by the Spin Verifier. Such a mapping allows the developer to verify claims about the Web service. The second functionality is the OWL-S Virtual Machine that is used by CODE to generate automatically the client code.

3.3 General Design of CODE

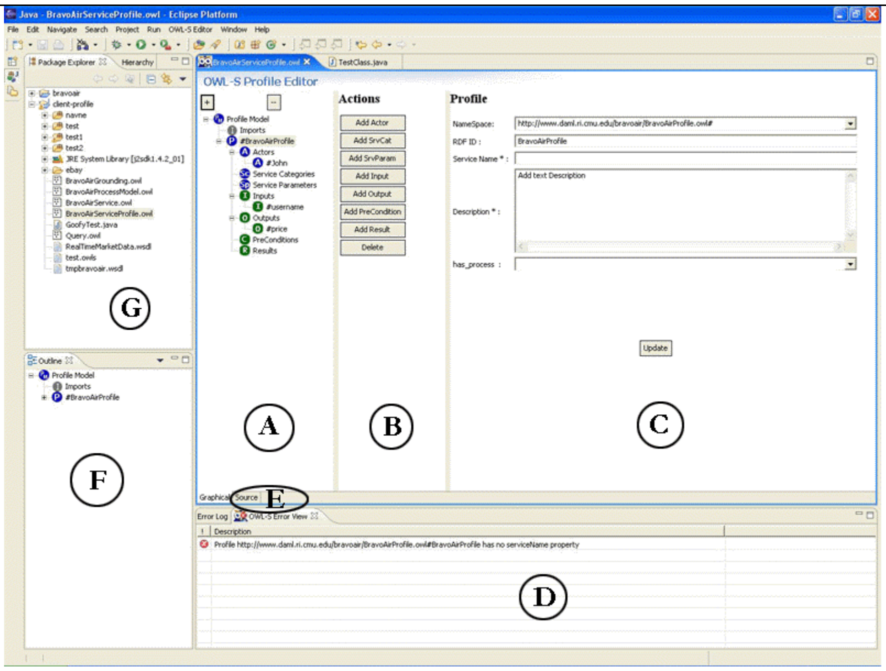
In this section we will discuss the general design principles that are observed across all the OWL-S editors in CODE. There are four editors to support different fragments of the OWL-S descriptions namely profile, process model, grounding and service. A uniform user interface is maintained across all four editors to reduce to maintain cognitive consistency and make the interfaces easier to use.

Figure 5 shows the general layout of the editors. The editor is mainly divided into four parts: file navigation pane, outline pane, main editor pane and error pane. The navigator pane marked as G is a file navigation window and is contributed by the eclipse framework. It is used to browse and manipulate the file system and is also used to manage the projects inside the eclipse workspace. The outline pane is marked as F displays a tree-based synopsis of the file that is being edited. The main editor pane that is composed of areas labeled A, B, C and E provides means to edit the OWL-S files. Finally error pane labeled D displays information of about the errors in the OWL-S file that is being edited.

The main editor pane provides two modes editing form based editing and text based editing. The form editor and the text editor are arranged in a tabular layout stacked one on top of the other, the Figure 5 shows the form editor and the text editor can be accessed using the tab (shown as label E) located at the bottom of the form editor. The form editor provides guidance to the developer on what information should be added at each stage of the compilation of the OWL-S description. For instance, in the compilation of a process, it requires the developer to enter the inputs, outputs, preconditions and effects. In turn, each one of them is a form that requires the developer to enter the appropriate information. If the information entered is not correct the developer is flagged an error that she can immediately fix.

The form editor is further divided into three sections namely tree pane, action pane and form pane. The tree pane labeled A in Figure 5 presents the elements of the OWL-S description in a hierarchical fashion. The user can browse through the hierarchical structure and may select an element to edit or add an attribute to an element. The action pane and the form pane are marked as B and C respectively are responsive to the selection in the tree pane. The action pane displays the controls such as adding and deleting of attributes that are pertinent to the element selected in the tree pane. Similarly the form pane displays the attributes of the element in form like manner that may be modified.

The text-based editor is an extension of SWeDE, an eclipse-based OWL editor. Although this mode of editing is relatively less intuitive than the form editing, it can be used by more experienced developers to generate their OWL-S code more expeditiously as well as to generate ontologies that are used to describe concepts that are specific of the Web service.

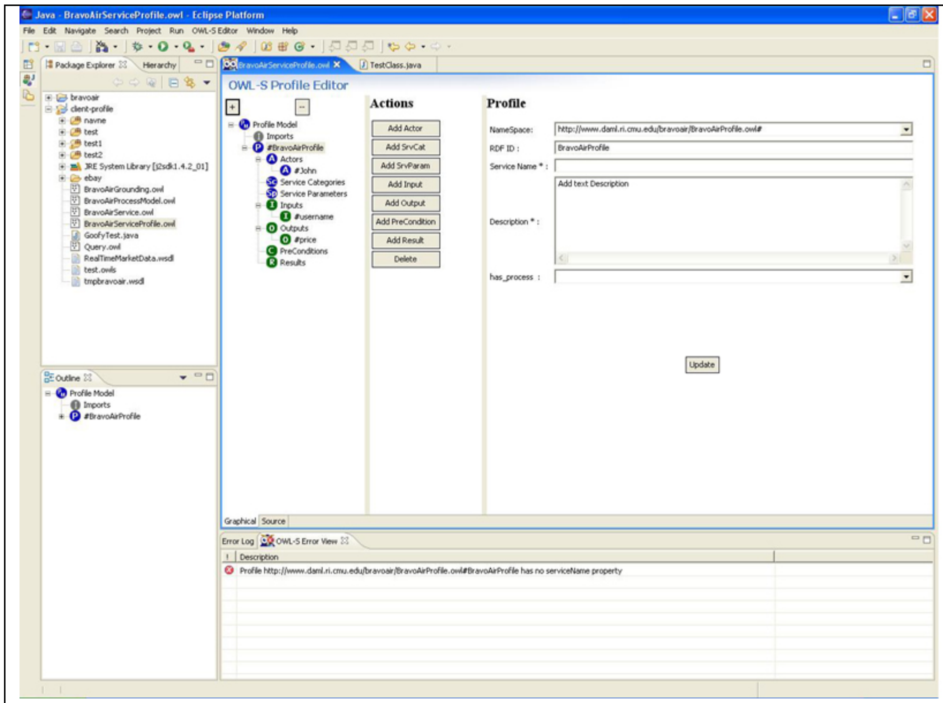


NameSpace: http://www.daml.ri.cmu.edu/bravoair/BravoAirProfile.owl#
RDF ID : BravoAirProfile
Service Name * :
Description * : Add text Description
has_process :

Profile http://www.daml.ri.cmu.edu/bravoair/BravoAirProfile.owl#BravoAirProfile has no serviceName property

Figure 5: General layout of OWL-S Editors

Apart from the editing functionalities provided by the OWL-S editors, each editor also offers functionalities specific to the type of OWL-S file it handles. For example the profile editor provides functionalities like publishing, querying etc and likewise the process editor provides functionalities like verification, execution etc. These functionalities that are specific to each editor can be accessed using the drop down menu of the main eclipse window.



NameSpace: http://www.daml.ri.cmu.edu/bravoair/BravoAirProfile.owl#
RDF ID : BravoAirProfile
Service Name * :
Description * : Add text Description
has_process :

```
! | Description
  | Profile http://www.daml.ri.cmu.edu/bravoair/BravoAirProfile.owl#BravoAirProfile has no serviceName property
```

Figure 6: Profile Editor

3.4 Profile Editor

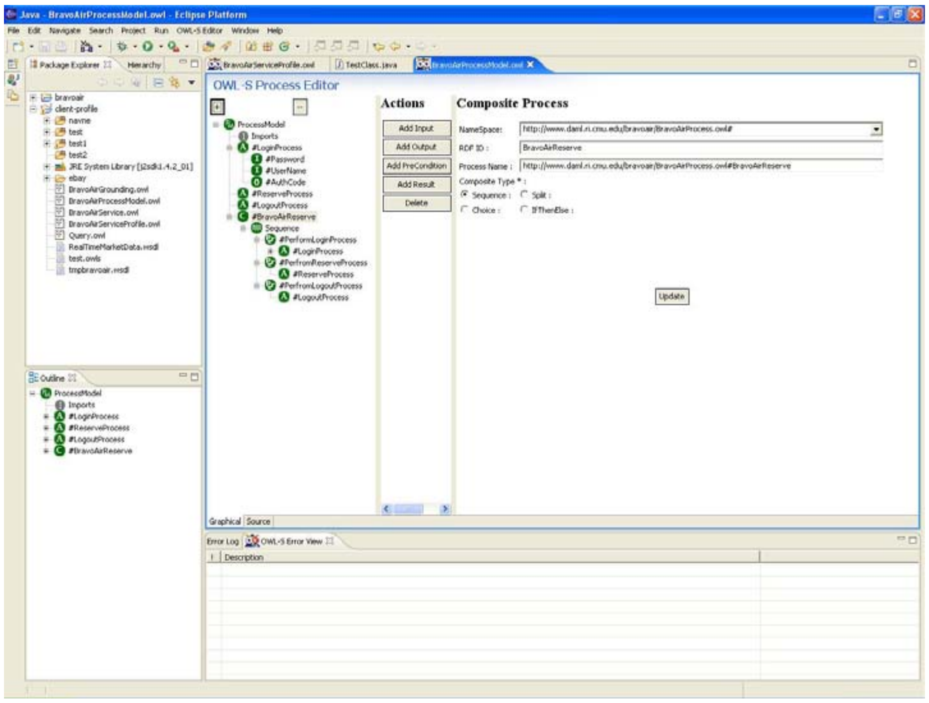
The Profile editor supports the developer in the following two tasks: the first one is the editing of the Service Profile of the Web service; the second one is the registration and querying with an UDDI server.

Figure 6 displays the form-based editor that is used to compile the OWL-S Profile. The leftmost selection of the OWL-S Profile editor shows the hierarchal structure of the profile file that is being edited. The action pane shows the list of actions that can be performed on the node that is selected; in this case a profile is selected in the tree structure. The form pane displays the attributes of the node that is selected.

The profile editor also supports the discovery of semantic web services for both web service developers and web service consumers. On completing the profile a web service developer can register the profile to an UDDI registry. Similarly a consumer look for a web service could compile a profile and use it to query the UDDI registries to look for web services that satisfy her profile. The results returned by the UDDI registry are displayed to the user that may load into the editors for further processing.

3.5 Process Model Editor

The Process Editor supports the developer in the generation of the Process Model using the same approach of the Profile editor. Similar to the profile editor, it provides a form-based editor to define processes and their control and data flow. The tree structure in the form editor supports drag and drop operation that expedites the construction of control flow and dataflow in composite processes.



Composite Process

NameSpace: http://www.daml.ri.cmu.edu/bravoair/BravoAirProcess.owl#

RDF ID : BravoAirReserve

Process Name : http://www.daml.ri.cmu.edu/bravoair/BravoAirProcess.owl#BravoAirReserve

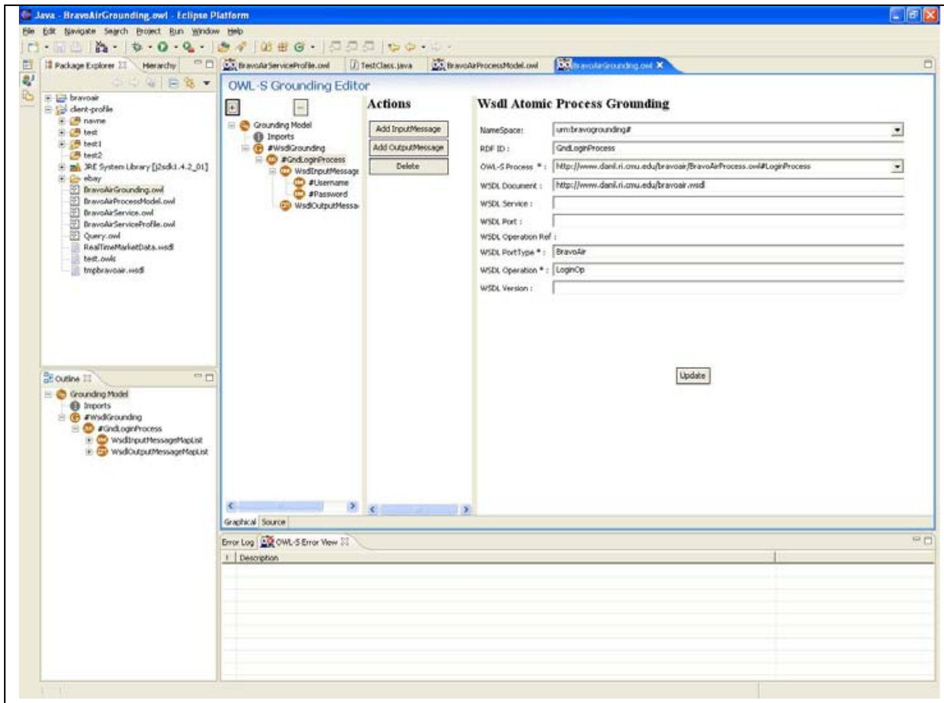
Composite Type * : [✓ Sequence :] Split : Choice : IfThenElse :

Figure 7: Process Editor

When adding a new composite process we add the components of the process that constitutes the control flow of a process. While display a composite process in process tree its components are displayed in a nested manner and hence one can visualize the control flow of the process model using the structure of the process tree.

A dataflow link to a component of the composite process is added by selecting it and selecting the appropriate action from the action pane. In order to add a dataflow link between two components we need three information, first the name of the input or the output of the component that needs the data, second the name of the other component that generates the data and finally the name of the input or output in the other component that actually has the data.

In addition, the editor provides verification and execution functionalities. The developed Process Model can be verified using the Spin model checker, to eliminate any inconsistencies in the workflow. Likewise, the developer can execute the Process Model using OWL-S VM to eliminate any error during actual invocation of the Web service. The OWL-S VM execution can also be used by client to execute the process model of the service that she discovered using the profile editor.



Wsdl Atomic Process Grounding

NameSpace: urn:bravogrounding#
RDF ID : GndLoginProcess
OWL-S Process * : http://www.daml.ri.cmu.edu/bravoair/BravoAirProcess.owl#LoginProcess
WSDL Document : http://www.daml.ri.cmu.edu/bravoair.wsdl
WSDL Service :
WSDL Port :
WSDL Operation Ref :
WSDL PortType * : BravoAir
WSDL Operation * : LoginOp
WSDL Version :

Figure 8:Grounding Editor

3.6 Grounding Editor

The grounding editor supports the compilation of grounding descriptions. In order to compile the grounding description the following two files are required: a process model description file containing information about atomic processes and a WSDL file that contain information about the operations and messages exchanged. Theses two files are loaded into the editor before commencing the compilation process. We generate grounding descriptions by adding information like which process in the process model description maps to which operation in the WSDL file, likewise which input/output of a process maps to which message in the WSDL file. If the user generated the OWL-S description using the WSDL2OWL-S converter, then most of the mapping information will be already present in the grounding file and would be missing very little information. In this case the generated grounding file is loaded into editor to complete the missing information.

3.7 Service Editor

The service editor assists the developer in the compiling the OWL-S service description. The function of the service description is to bind the profile, the process, and the grounding descriptions of a web service since there are no explicit links exist between them. Using the service editor we can load the profile, the process and the grounding descriptions and build the service description by choosing the profile, the process and the grounding pair that represents the service.

4 Supporting clients

Most of the discussion in the previous sections concentrated on using CODE to support Web service development and description. While this has been the major focus of our development, CODE can also be used to support the implementation of Web services clients. Client development requires two functionalities, the first one is the discovery of Web services that have the capability to solve crucial problems of the client; the second problem is the development of the interaction code that allows the client to interact with the Web service correctly.

CODE supports the development of clients on both accounts. The discovery process is support through the interaction with the OWL-S/UDDI. Essentially, to discover a Web service the developer can use the Profile Editor to specify what it expects from the Web service. Essentially, the developer compiles the profile of the "perfect" Web service she would like to work with. This profile is translated into a UDDI representation and used to query the OWL-S/UDDI. The result of the query is a set of services that match the query. In some lucky case the match may be perfect, but in most cases, the discovery process will select Web services that are "similar enough" to the Web service that was originally requested. The client developers use the descriptions of these services to decide which one to use and to decide whether she needs to gather additional information by invoking additional Web services.

The second aspect of the implementation of the client code is the generation of the interaction code with the Web service. The problem here is that the Process Model specifies the order in which the Web service expects information and what type of information the Web service needs. Any violation of such order, or a violation of the type of information expected by the Web service would lead to a failure of the interaction. To control the interaction process, CODE supports the automatic generation of Web service specific interfaces to the OWL-S Virtual Machine. The OWL-S VM takes the OWL-S specification and executes it, but it has to ask to the client what information to send, and which non-deterministic choices to make. By implementing the interface the developer is obliged to provide the functionalities that support the OWL-S VM in its interaction with the Web service.

5 Conclusions

CODE is an integrated development environment that aims at supporting Web service developers in both the implementation of Web services and in the generation of an OWL-S description of their Web services. The goal of CODE is to support both the server side and the client side developers. The server side developers are supported by providing two modalities of generating Web service descriptions. The first modality is code-driven where the developer generates the OWL-S description directly from the Java code that implements the Web service. The second modality is model-driven where the developer first generates the OWL-S description of the Web services, and then uses it as a model to generate the Java implementation. The client side developers are supported by supporting the discovery of Web services that the client may want to interact with, and by generating automatically the client code that interacts with the Web service.

The functionalities provided by CODE are very similar to the functionalities provided by the OWL-S Editor under development at SRI [3]. The OWL-S Editor is defined as a Protégé [7] plug-in and it builds on top of the OWL plug-in already available in Protégé. Overall it provides a good support for OWL and a good graphical interface that displays the workflow and data flow of Process Models. At the time of writing is quite difficult to compare the two systems since they are in very active development, therefore new functionalities are added all the time. Furthermore, any distinction between the two systems depend more on the underlying support, namely Protégé vs Eclipse, rather than profound philosophical differences. A research challenge for the near future will be to analyze the functionalities provided by the two systems to distill a core set of functionalities that are needed to implement Semantic Web services.

Future work on CODE will involve a greater integration with the Eclipse Modeling Framework (EMF) model-building tool [4]. EMF allows the developer to move seamlessly between the modeling and the coding of a system. Using EMF, developers may decide how much they want to model the system that they are building and how much they want to develop directly. This approach is very similar to our vision to support both model-driven and code-driven development of Web services. We are currently analyzing how to integrate the development of OWL-S description within the EMF framework. If our attempt will be successful, developers will be able to move seamlessly between UML modeling, Java programming and OWL-S description of Web services.

References

1. Axis Web site: http://ws.apache.org/axis/.
2. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.:Web Service Description Language (WSDL 1.1): http://www.w3.org/TR/wsdl.
3. Denker, G., Elenius, D., and Martin, D.: OWL-S Editor: http://owlseditor.semwebcentral.org/.
4. Eclipse Modeling Framework (EMF) homepage: http://download.eclipse.org/tools/emf/scripts/home.php.
5. Jena Homepage: http://jena.sourceforge.net/.
6. Microsoft .Net Homepage: www.microsoft.com/net/.
7. N. F. Noy, R. W. Fergerson, & M. A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. 2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France, 2000.
8. OWL-S Homepage: www.daml.org/services/OWL-S_1.1.
9. SWeDE Homepage: http://owl-eclipse.projects.semwebcentral.org/.
10. UDDI Homepage: www.uddi.org.
11. Gerard J. Holzmann: The SPIN Model Checker: Primer and Reference Manual: Addison-Wesley Professional (September 4, 2003).
12. Eclipse http://www.eclipse.org.
13. Deborah L. McGuinness, Frank van Harmelen: OWL Web Ontology Language: http://www.w3.org/TR/owl-features/.

Appendix D - **A Broker for OWL-S Web Services**

Massimo Paolucci, Julien Soudry, Naveen Srinivasan, Katia Sycara

The Robotics Institute, Carnegie Mellon University

Abstract:
Brokers are widely used in distributed information systems such as Multi-agent systems and distributed databases. Yet, there has not been a detailed analysis of Brokers' architecture and no general solution has been proposed on how the Brokers' tasks have to be accomplished. In this paper, we provide a detailed analysis of these tasks, and an implementation based on OWL-S. We show that while OWL-S is adequate to provide all the information that is needed by the Broker, the straightforward implementation of the Broker using OWL-S results in a paradoxical situation. We solve this paradox by extending the Process Modeling language of OWL-S. Finally, we propose a solution to a number of issues that arise in the brokered management of the interaction between Web services such as the abstraction from queries to capabilities required to solve that query, and management of the knowledge required by the Broker to control the multi-party interaction.

Key words:OWL-S, OWL, Brokers, Semantic Web services

Introduction

Brokering is a natural coordination and mediation mechanism that we often encounter in our daily life. The most striking example of Brokers is stock Brokers that mediate between the stock market and its investors. Brokers play a role when there is a need to facilitate the interaction between two or more parties. For example, if two parties want to communicate, but they do not share a common language, Brokers may provide translation services, or if the two parties do not trust each other, a Broker may provide a trusted intermediary (e.g. an escrow service for e-commerce transactions). Furthermore, Brokers may provide anonymization for one (or both) of the parties, by mediating the transaction.

Not surprisingly, Brokers are one of the main discovery and synchronization mechanisms among autonomous agents [9][28]. Examples include the OAA Facilitator [19] which Brokers between OAA agents that collaborate toward the solution of a problem. Furthermore, Brokers have been widely used in many agents' applications such as integration of heterogeneous information sources and Data Bases [16], e-commerce [14] [11], pervasive computing [6] and more recently in coordinating between Web services in the IRS-II framework [21]. Finally, theoretical studies [9] [28] show that Brokers can perform a range of coordination activities such as load balance between different agents, and anonymization where the Broker acts as a proxy of an agent effectively hiding the provider or the requester of a given functionality.

Because of their properties and their wide applicability, Brokers are a natural candidate component for the Web Services infrastructure. The SOAP [20] specification and WS

Architecture group (WSA) [4] specifications have provisions for intermediaries, but their role is limited to message routing. Furthermore, WSA assumes the existence of policy enforcing guards and auditing guards that act as Brokers that verify that the transactions performed by the agents are consistent with current policies. However, Brokers with rich functionality of *discovery* and *mediation*, are not part of the Web Services infrastructure.

In the current Web services infrastructure, the only component that is devoted to discovery is UDDI [27]. However, UDDI is a registry that does not perform any mediation between the requester and the discovered service provider. A number of services that satisfy a particular request could be found using UDDI, but then it is up to the requester to decide which web service to use and how to interact with the selected provider. The DAML-S/OWL-S Matchmaker [23] provides automated semantic matching of service requests to capability advertisements, but, after the matchmaker returns a list of candidate providers, the selection of the most suitable service is done by the requester, which can use the DAML-S Virtual Machine [25] to invoke the selected service.

In this paper, we provide an analysis of the requirements of a Broker that performs both discovery and mediation between agents and Web services. We show that such a Broker performs very complex reasoning tasks that include (1) the interpretation of the capability advertisements of service providers; (2) the interpretation of the requesters' queries that must be fulfilled by a service provider; (3) finding the best provider based on the requester's query; (4) invocation of the selected provider on behalf of the requester, interacting with the provider as necessary to fulfill the query, and (5) returning the query results to the requester. The accomplishment of these tasks requires ontologies to describe capabilities of Web services, their interaction patterns and the domain they operate on, and a logic that allows reasoning on those ontologies. Furthermore, we will provide a description of an implementation of a Broker using OWL-S [22], a Web services description language based on OWL [8] and the Semantic Web [3].

The implementation of the Broker also highlights some of the challenges of the automatic interaction between Web services. The first overall challenge concerns the selection of a suitable service provider to fulfill the requester's query. The requester's query is a particular instantiation of an input to an invocation of a service (e.g. "what is the five day forecast in Pittsburgh"?), while advertisements of Web services express the capabilities of the Web service, in other words the class of queries that it can answer (e.g. "I provide a service that gives weather information"). Therefore, the Broker cannot match directly the query against its advertisement store, rather the Broker needs first to transform the query into the capabilities requested to answer it, only then the matching can be effected. How this transformation can be automatically generated is a big challenge. The second major challenge stems from the OWL-S specification of how a requester interacts with a provider. This interaction is specified through the provider's Process Model (see section 2). After a provider has been selected, its Process Model is used by the requester to guide the information that the requester must include in its messages to the provider, (since the Process Model specifies what information the provider expects). In a Brokered system neither a requester that addresses a Broker nor the Broker itself know a priori which is a suitable provider, hence they do not know a suitable Process Model to guide the information that the requester provides to the Broker or the information that the Broker presents to the requester. Hence it is not clear how a requester can formulate its query to the Broker or how the Broker can intermediate interactions of a requester with a provider whose Process Model is known only at a later stage. These are two additional challenges that a Broker design must address.

In this paper, we present an approach to the development and implementation of a Semantic Broker, namely a Broker that performs semantic discovery and mediation among Web Service requesters and providers[6]. The Broker uses OWL-S to perform its functions. Our approach gives solutions to the three challenges presented above. The solution necessitates an extension to OWL-S to address the "chicken and egg" problem of the initial lack of knowledge of the provider's Process Model on the part of the requester and of the Broker.

The rest of the paper is organized as follows. In section 2, we present an overview of OWL-S; in section 3, we provide a detailed analysis of the Broker, exploring its interaction protocol and the reasoning tasks that it has to accomplish. In section 4, we show how OWL-S provides the information that the Broker needs to perform its tasks. In section 5, we explore the problems that emerge from describing the Broker with OWL-S, and we describe the *exec* extension of OWL-S. In section 6, we describe the basic features of our implementation and provide details on how we address the reasoning problems of the Broker. At last, in section 7, we conclude.

OWL-S

OWL-S is a Semantic Web Services description language that enriches Web Services descriptions with semantic information from OWL [8] ontologies and the Semantic Web [3]. OWL-S is organized in three modules: a *Profile* that describes capabilities of Web Services as well as additional features that help to describe the service. A *Process Model* that provides a description of the activity of the Web Service provider from which the Web Service requester can derive information about the service invocation. A *Grounding* that is a description of how abstract information exchanges described in the Process Model is mapped onto actual messages that the provider and the requester exchange.

The role of the OWL-S Profile is to support the requester in (a) discovering suitable providers, and (b) selecting among them. The OWL-S Profile achieves the first goal, i.e. provider discovery, by prescribing ways for representing Web Service capabilities; the Service Profile fulfills the second goal, i.e. service selection, by providing for the representation of additional information about the service, such as information describing provenance and quality or cost specifications of the Web service.

A Web Service capability is the description of the service functionality, i.e. what the service does. For example, the capability of Barnes and Noble, a bookseller, is to sell books. The capability of a Web Service can be viewed in two ways: first as a service category within an ontology of services (e.g. selling books is-a selling products) or as a transformation of a set of inputs to a set of outputs (e.g. selling books transforms the inputs "book title" and "book author" to the output "book invoice"). OWL-S Profile describes capabilities of Web Services by the transformation that they produce. Besides transforming inputs into outputs at an information level, invoking a Web Service can produce effects in the real world and need preconditions to be satisfied. For example, invoking the book selling service and buying a book produces the effect in the real world that the requester's credit card gets charged; a precondition for the invocation of the book selling service is that the requester has a valid credit card.

[6] Naturally, we envision multiple distributed Brokers on the Internet or Intranets. In this paper, we speak of "the" Broker for ease of exposition.

In addition to capabilities, an OWL-S Profile provides *provenance* information that describes the entity (person or company) that deployed the service; and *non-functional parameters* that describe features of the services such as quality rating for the service. These pieces of information help a requester discriminate among different services with similar capabilities. For example, a requester may prefer a bookseller that has a Dunn and Bradstreet quality rating.

In order to make its capabilities known to service requesters, a service provider advertises its capabilities with infrastructure registries, or more precisely *middle agents* [28], that record which agents are present in the system. UDDI is an example of a middle agent, with the limitation that it can make limited use of the information provided by the OWL-S Profile. The OWL-S/UDDI Matchmaker [23][24] is another example, which combines UDDI and OWL-S. Finally, the Broker defined in this paper is another example of a middle agent that performs both discovery and mediation.

The second module of OWL-S is the Process Model. The Process Model has two aims: the first one is to show how the provider achieves its goals, and the second to provide the *requester-provider interaction protocol.* The first goal is achieved by allowing the provider to make public a description of its computation, to the extent that the provider feels comfortable to do so. The requester-provider interaction protocol is derived by the processes that the provider performs by locating when the provider needs information, and what type of information, and where it sends information, and what type of information.

A Process Model defines a set of concurrent threads of execution. Each thread is an ordered collection of processes. OWL-S distinguishes between two types of processes: composite processes and atomic processes. Atomic processes correspond to operations that the provider can perform directly. Composite processes are used to describe collections of processes (either atomic, or composite) organized on the basis of some control flow structure. For example, a *sequence* of processes is defined as a composite process whose processes are executed one after the other. Other control constructs supported by OWL-S are *cond* for conditional expressions, *choice* for non-deterministic choices between alternative control flows, and *spawn* for spawning a new concurrent thread. Finally, OWL-S includes looping constructs like *while* and *repeat-until.*

Seq	$\frac{-}{\Pi,E[\text{return v >>=e}],\varphi)\rightarrow\Pi,(E[(\text{e v})],\varphi)}$
Spawn	$\frac{-}{\Pi,(E[\text{spawn e}],\varphi)\rightarrow\Pi,(E[\text{return()}],\varphi),(e,\varnothing)}$
Cond^{True}	$\frac{-}{\Pi,(E[\text{cond C } e_1\ e_2],\varphi)\rightarrow\Pi,(E[e_1],\varphi)}$
Choice^{Left}	$\frac{\Pi,(E[e_1],\varphi)\rightarrow\Pi',(E[e_1'],\varphi')}{\Pi,(E[\text{choice } e_1\ e_2],\varphi)\rightarrow\Pi',(E[e_1'],\varphi')}$
Atomic[7]	$\frac{-}{\Pi,(E[\text{atomic e}],\varphi)\rightarrow\Pi,(E[\text{return ()}],\varphi')}$

Table -1. **Execution Semantics of OWL-S control structures**

The execution of a process produces a state transition where either some information is exchanged with some partner, or the agent produces a change in the environment. A state is

[7] The execution semantics presented in [1] does not include an explicit notion of atomic process, rather atomic processes are constructed as a combination of operations that receive messages, send messages, and apply functions.

defined as a touple (φ,Π) where Π represents the set of concurrent threads, and φ the state of the thread the process is executed in [1][8]. Processes modify the state by either changing the state of their thread φ, for instance, an atomic process may read a message from a port, or modify the set of concurrent threads Π through the spawning of new threads or the closing of other threads. The formal semantics of the OWL-S composite and atomic processes is shown here in Table 1[9]. Looping constructs are implemented as combinations of sequences and conditions.

Each rule in Table 1 specifies how the execution of a process changes the overall state. Sequences of processes, expressed here by the temporal constraint *return v >>=e*, applies *e* to the results *v* of the previous step. The execution of a spawn operation, results in the beginning of the execution of a new thread (e,∅), while it returns no value in the current thread (return ()). The other rules specify the result of executing other types of control constructs, $Cond^{True}$ specifies the results of the execution of a conditional statement if the condition is true; a similar rule would be used for a false condition. $Choice^{Left}$ specifies the results of the execution of a non-deterministic selection of the first process of a list; a similar rule would be used for other choices. Finally, *Atomic* describes the results of executing an atomic process, which has an effect on the state of the current thread φ but it does not modify the set of concurrent processes ∏.

The last module of OWL-S is the Grounding that describes how atomic processes which provide abstract descriptions of the information exchanges with the requesters, are transformed into concrete messages or remote procedure calls over the net. Specifically, the OWL-S Grounding is defined as a one to one mapping from atomic processes to WSDL [5] input and output message specifications. From WSDL it inherits the definition of abstract message and binding, while the information that is used to compose the messages is extracted through the execution of the process model during service invocation.

Therefore, the Web Services philosophy of interaction between a service requester and a service provider is that a requester would need to know the information that a service provider requires at different stages of the interaction. For example, in industrial standards, the requester-provider interaction is governed by knowledge of the provider's Web Services Description (WSD) given in WSDL, and in Semantic Web Services, the requester-provider interaction presupposes knowledge on the part of the requester of the Process Model (plus WSD) of the provider.

Overview of the Broker

Brokers have been widely applied in many different applications and domains, therefore, not surprisingly, there are many different definitions of what a Broker is. We adopt the definition of the Broker protocol based on [9], and graphically summarized in Figure 1. Any transaction involving a Broker requires three parties. The first party is a requester that initiates the transaction by requesting information or a service to the Broker. The second party is a provider

[8] The execution semantics that we use was originally proposed for DAML-S 0.6. While many aspects of the language changed in the evolution to OWL-S 1.0 that we use here, the execution semantics of the basic constructs of the Process Model is still valid.

[9] We provide here a very superficial explanation of the OWL-S execution semantics. A complete presentation is in [1].

which is selected among a pool of provider as the best suited to resolve the problem of the requester. The last party is the Broker itself.

The protocol in Figure 1 can be divided in two parts: the advertisement protocol, and the mediation protocol. In the advertisement protocol, the Broker first collects the advertisements of Web services that are available to provide their services. These advertisements, shown in Figure 1 by straight thin lines, are used by the Broker to select the best provider during the interaction with the requester. The mediation protocol, shown in Figure 1 using thick curve lines, requires (1) the requester to query the Broker and wait for a reply while the Broker uses its discovery capabilities to locate a provider that can answer the query. Once the provider is discovered, (2) the Broker reformulates the query for that provider, and finally queries it. Upon receiving the query, (3) the provider computes the reply to the Broker and finally (4) the Broker replies to the requester.

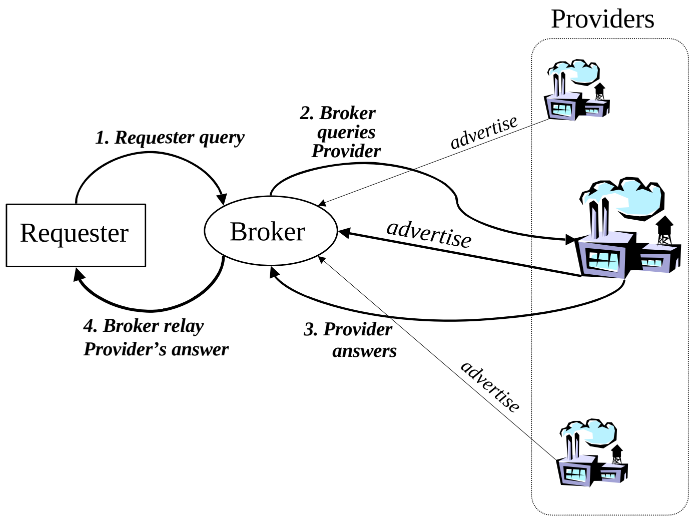


Figure -1. **The Broker's Protocol**

The protocol can be more complicated because it may require more interactions between the provider and the requester, when the provider instead of replying directly asks for further information. For example, the requester may have asked the Broker to book a flight from Pittsburgh to New York. Since there are multiple flights between the two cities, the provider may ask the Broker, and in turn the requester, to select the preferred flight. These interactions are resolved with multiple loops through the protocol. For example, the Broker translates the list of flights retrieved by the provider for the requester, through steps (3) and (4) of the protocol, and then translates the message with the selected flight from the requester to the provider, steps (1) and (2) of the protocol. The only exception is that step (1) does not require any discovery since the provider is already known.

In some cases, the Broker may be able to answer the request of information from the provider directly; therefore, it does not have to involve the requester in the interaction. For example, the requester's query may request a seat on the cheapest flight from Pittsburgh to New York. When the provider reports all flights between the two cities, the Broker selects the cheapest one and responds directly to the provider without asking anything to the requester. Following the protocol in Figure 1, these interactions are the result of the inner loop produced by the steps (3) and (2). The provider sends message (3) and the Broker responds directly with (2) without contacting the requester.

The protocol described above shows that the Broker needs to perform a number of complex reasoning tasks for both the discovery and mediation part of its interaction. The discovery task requires the Broker to use the query to describe the capabilities of the desired providers that can answer that query, and then match those capabilities with the capabilities advertised by the providers. During the mediation process, the Broker needs to interpret the messages that it receives from the requester and the provider to decide how to translate them, and whether it has enough information to answer directly the provider without interaction with the requester. In the next two sections, we will analyze these reasoning tasks in more detail.

Discovery of Providers

The task of discovery is to select the provider that is best suited to reply to the query of the requester. Following the protocol, providers advertise their capabilities using a formal specification of the set of capabilities they posses, i.e. the set of functions that they compute. These capability specifications implicitly specify the type of queries that the provider can answer.

The discovery process requires two different reasoning tasks. The first one is to abstract from the query of the requester to the capabilities that are required to answer that query. The second process is to compare the capabilities required to answer the query with the capabilities of the providers to find the best provider for the particular problem.

The first problem, the abstraction from the query to capabilities, is a particularly difficult one. Capabilities specify what a Web service or an agent does, or, in the case of information providing Web services, what set of queries it can answer. For example, capabilities of Web services may be to provide weather forecasting, or sell books, or register the car with the local department of transportation. Queries instead are requests for a very specific piece of information. For example, a query to a weather forecasting agent may be to provide the weather in Pittsburgh, while a query to a book-selling agent may be to buy a particular book. Because of their difference, queries and capabilities are also expressed in very different formats. The task of the Broker therefore is to abstract from the particular query, to its semantics, i.e. what is really asked. Finally, the broker must identify and describe in a formal way the capabilities that are needed to answer that query.

The second task of the discovery process is to match the capabilities required to answer the query with the advertisements of all the known providers. Since it is unlikely that the Broker will find a provider whose advertisement is exactly equivalent to the request, the matching process can be very complicated, because the Broker has to figure to what extent the provider can solve the problems of the requester.

Management of Mediation

The second reasoning task that the Broker has to accomplish is to transform the query of the requester into the query to send to the provider. This process of mediation has two aspects. The first one is the efficient use of the information provided by the requester to the Broker, the second one is the mapping from the messages of the requester to messages to the provider and vice versa.

Since the requester does not a priori know which is the relevant provider, the (initial) query it sends to the Broker and the query input that the (selected) provider may need in order to provide the service may not correspond exactly. The requester may have appended to the query information that is of no relevance to the provider, while the provider may expect information that the requester never provided to the Broker. In the example above, we considered the example of a requester that asks to book the cheapest flight from Pittsburgh to New York. But, besides the trip origin and destination, the selected provider may expect date and time of departure. In the example, the requester never provided the departure time, and the provider has no use for the "cheaper" qualifier. It is the task of the Broker to reconcile the difference between the information that the requester provided and the information that the provider expects, by (1) recognizing that the departure time was not provided, and therefore it should be asked for, and (2) finding a way to select the cheapest flight among the ones that the provider can find.

The other type of inference on the message passing that the Broker has to perform is the mapping between ontologies and terms used by the two parties. For example, the requester may have asked for information on IBM whereas the provider expects inputs in terms of International Business Machine Corporation. Another, more complicated mismatch may be at the level of concepts and their relations in the ontologies used for inputs and outputs of the provider vis a vis the ontological information used by the requester. For example, the requester may have asked for the weather in Pittsburgh, but instead the provider can report only the weather at major airports. The task of the Broker in this case is to infer which is the most appropriate airport, and use it in the query to the provider. Therefore, instead of asking for the weather in Pittsburgh, the Broker asks the provider for the weather at PIT, where PIT is the code of the Pittsburgh International Airport.

Finally, the Broker has the non-trivial task of translating between the different syntactic forms of the queries and replies. The examples that we discussed above assume semantic mismatches between the different messages that the Broker has to interpret and send. These messages have to be compiled in an appropriate syntactic form, and despite their semantic similarity, the messages would be realized in very different ways. The task of the Broker is to resolve syntactic differences, and to formulate messages that all the parties can understand.

In conclusion, the Broker performs a number of complex reasoning tasks that range from discovery to the interpretation, translation and compilation of messages. To accomplish these tasks, the Broker needs the support of a formal framework that allows complex reasoning about agents, what they do and how to interact with them. Furthermore, the Broker needs a way to translate the semantics of the information that it wants to communicate, into the syntactic form that the provider or the requester expects.

OWL-S support of Broker'S REASONING

The OWL-S language and ontology provides constructs to support the Broker in both discovery and mediation between Web services. The OWL-S Profile supports the discovery process by providing a representation of capabilities of Web services and agents. The OWL-S Process Model and Service Grounding provide support for the interaction between the Broker and the requester and provider of the service.

The discovery process requires a representation of capabilities of provider services and a representation of the capabilities that are required to answer the query. These capabilities are represented in OWL-S by the Service Profile. In addition to the representation of capabilities, the analysis above showed that the Broker needs an abstraction process from the query to the capabilities needed to answer it, and a matching mechanism from the capabilities required for the query to the capabilities of the providers. This matching mechanism allows the selection of the best service provider to answer the query.

A number of capability matching algorithms for OWL-S based Web services have been proposed (see [2][10][15][23]) which exploit OWL ontologies and the related logics to infer which advertisements satisfy a request for capabilities. These algorithms can be used to solve the second problem: the matching from the capabilities required for the query to the capabilities of the providers.

The first problem, the abstraction from the query to the capabilities, is more complicated. First of all, there is no explicit support in OWL-S for queries, nevertheless, it is easy to use the OWL Query Language (OWL QL) [7] [12] which relies on the same logics required by OWL-S. The transformation is still an open problem, which, to our knowledge, has never been addressed. In the next section, we will propose an abstraction algorithm to transform queries into capabilities.

After selecting a provider, the Broker has access to the provider's Process Model from which it can derive the provider's interaction protocol by extracting what information the provider will need, in what order, and what information it will return. For the rest of the interaction the Broker acts as the provider's direct requester. However, this relation is not straightforward. Since the Broker acts on behalf of the requester, it must somehow transform the requester's initial query (and all subsequent messages) into a query (or a sequence of queries) to the provider. This transformation is necessary since the requester cannot "see" directly the Process Model of the provider, but interacts with the provider only through the Broker. We show how this transformation can be done in section 6.1.

Furthermore, since the requester initiated its query without having access to the provider's Process Model (since the provider was not known at the time of the requester's query initiation), the Broker needs to infer what additional information it needs from the requester. Once it has done that, it then uses this knowledge to construct a new Process Model. This new Process Model is presented by the Broker to the requester, not as the Process Model of the selected provider but as the process Model of the Broker. This makes sense since the requester interacts only with the Broker. The new Process Model indicates to the requester what information is needed and in what order. How the Broker infers the additional information it needs from the provider and how it constructs the new Process Model is presented in section 6.2.

The Service Grounding provides a mapping from the semantic form of the messages exchanged as defined in the Process Model, to the syntactic form as defined in the WSDL input and output specifications. The Grounding provides to the Broker the mapping from the abstract semantic representation of the messages to the syntactic form that these messages adopt when they become concrete information exchanges. The Broker uses this mapping to interpret the messages that it receives and compile the messages that it sends to the requester or to the provider.

A Process Model for the Broker

Every interaction between agents using OWL-S must be effected in accordance with the provider’s Process Model. Interactions with a Broker are no exception. Since, from the point of view of the requester, the Broker is the provider, it expects the Broker to publish a Process Model that is to be used during the interaction. In this section, we show that the Broker’s Process Model pushes the boundaries of the current specification of OWL-S.

The Broker’s Paradox

A requester will interact with the Broker using the Broker’s Process Model. The Broker’s Process Model should specify how the requester can submit its query, but it should also allow the requester to provide any additional information that the Broker needs to interact with the provider. Since, to the requester, the Broker is a (representative of) the provider, the Process Model of the Broker should contain the crucial elements of the Process Model of the provider. However, since the Broker is unaware of the provider until it has discovered and selected the provider based on a requester’s query, the Broker is faced with a challenge: it must publish a Process Model that depends on the provider’s Process Model, but the provider is not known until the requester reveals its query. On the other hand, the requester cannot query (interact with) the Broker until the Broker publishes its Process Model. The result is a paradoxical situation in which the Broker cannot reveal its Process Model until it receives the query of the requester, but cannot receive the query from the requester until it publishes its Process Model.

Essentially, the Broker’s paradox is due to the fact that the discovery of the provider depends on the requester’s query, while the rest of the interaction between the requester and the Broker depends on the provider selected. Ultimately, the Broker paradox results from an inflexibility of the OWL-S specification of service invocation, which requires the specification of the Process Model before the interaction, and it does not allow any means to modify the Process Model during the interaction.[10]

Extending the OWL-S Process Model

The solution of the Broker’s Paradox that we propose requires an extension of the specification of the OWL-S Process Model to allow the flexibility to dynamically modify an agent’s Process Model during the interaction. As a result, the Broker can provide an initial, provider-neutral, Process Model to the requester, and then modify it consistently with the requirements of the Process Model of the provider. The changes are then adopted by the requester in its interaction with the Broker.

To implement this solution, we propose to extend the OWL-S Model Processing language by adding a new statement, that we call *exec*. The *exec* statement takes as input a Process Model and executes it. Therefore, the Broker can compile a new Process Model, return it as an output of one of its processes, and then use the exec statement to turn the new Process Model into executable code that specifies the Broker’s new interaction protocol.

[10] Of course, the current industry proposed standards have the same inflexibility, since the Web Services interface Description must be specified once and for all with no provisions for on-the-fly loading or modification.

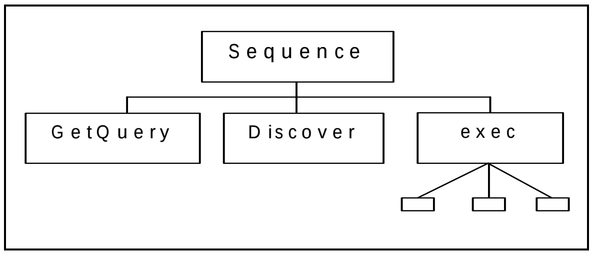


Figure -2. **Broker's Process Model**

The provider-neutral Process Model of the Broker is shown in Figure 2. It shows that the Broker performs a sequence of three operations, where the first operation is `GetQuery` in which the Broker gets the query from the requester. The second operation is `Discover` in which the Broker uses its discovery capabilities to find the best provider. The result of the Discover process is a new Process Model that depends on the provider found. Finally, the Broker performs the `exec` operation that passes control to a new Process Model. This change of control is shown in the figure by the three small rectangles that display processes that will be run as a consequence of the exec.

The use of the exec solves the Broker's Paradox by removing the inflexibility of the OWL-S Process Model. The exec operation allows the separation of service discovery from service invocation and interaction. First the discovery is completed, then the interaction, which depends on the discovered provider, is initiated through the exec.

One important question that is left unanswered is whether there is a clever way to use OWL and OWL-S that does not require the extension of the language that we propose. Unfortunately, such an extension does not exist, because neither OWL nor OWL-S provides a way to transform a term into a predicate of the logic, which is the essential step that is performed by the exec.

Formal Semantics of exec

Intuitively, the semantics of the exec operation is to execute the processes that it contains as arguments. In other words, the state transformation produced by exec(P) is equivalent to the state transformation produced by the direct execution of P. This intuition is captured by the axiomatic semantics of exec that we describe in Table 2, which is produced as a natural extension of the axiomatic execution semantics of OWL-S shown in Table 1.

exec(P)	$\dfrac{\Pi,(E[P],\varphi)\rightarrow\Pi',(E[P'],\varphi')}{\Pi,E[exec(P)],\varphi)\rightarrow \Pi',(E[P'],\varphi')}$

Table -1. **The execution semantics of the exec statement**

The execution of an exec statement is shown in Table 2. This rule specifies that the execution of *exec(P)* in the state ($\prod,\varphi$) should produces the same results that are produced by the execution of P in the same state in the state ($\prod,\varphi$). This definition allows us to transform the specification of a process P into the execution of the process, which is exactly what we are seeking with the definition of exec.

Implementation

We have implemented a prototype of a Broker that makes use of OWL-S with the exec extension described above to mediate between agents and Web services. We based our implementation of the Broker on the OWL-S Virtual Machine (OWL-S VM) [24], which is a generic OWL-S processor that allows Web services and agents to interact on the basis of the OWL-S description of the Web service and OWL ontologies. In the implementation of the Broker, we extended the OWL-S VM to include the semantics of the exec. Furthermore, we developed the reasoning that allows the Broker to perform discovery and to mediate the interaction between the provider and the requester.

In this section, we analyze how we implemented the different aspects of the Broker. We will first discuss the implementation of the discovery process, and then we will analyze the modification of the interaction protocol that allows the Broker to mediate between the provider and the requester. Finally, we will discuss the use of the OWL-S VM in the implementation that allows us to actually mediate between the two parties.

Supporting discovery

The Broker expects from the requester a query in OWL-QL format [12], where the predicate corresponds to a property in the ontology, the terms in the query are either variables, or instances that are consistent with the semantic type requirements of the predicate.

The discovery process takes as input the query of the requester and generates as output the advertisement of a provider (if any is known to the Broker) that can answer the query. The discovery process has three steps: first the Broker abstracts from the query to the capabilities that are required to answer that query, thus constructing a service request. Second, the Broker finds appropriate providers by matching the capabilities required to solve the query (the service request) with the capability advertisements of providers that have advertised with the Broker. Third, the Broker uses similarity of the match of the service request and the returned advertisements as well as non-functional parameters in the returned Service Profiles to select the most appropriate provider. The matching of the service request against the advertised capabilities was implemented using the OWL-S matching engine reported in [23] and [24].

The automatic abstraction from the requester's query to a service request is, to our knowledge, an unexplored problem. The abstraction process must respect the constraints of the OWL-S discovery process, namely generation of an OWL-S service profile with the appropriate required service inputs and outputs that reflected the semantic content of the query and also reflected the requirements of the generated service request so that the matching process would return an appropriate service provider. The abstraction procedure that we implemented distinguishes between variables, and the terms that are instantiated in the query. Since the result of the query should be an instantiation of the variables, ideally, the selected provider agent would take as inputs the instantiated terms and return as output an instantiation for the variables.

```
1. set V = set of variables in the query
2. set T= set of instantiated terms in the query
3. set I= abstraction of each term in T to its immediate class
4. use predicate definition in the ontology to abstract variables in V
   to their class
5. set O= abstraction of each variable in V to its class
6. generate a service request  with input I and outputs O
```

Figure 3: **The abstraction algorithm**

The instantiation algorithm follows the 6 steps listed in Figure 3. In the steps 1 and 2, terms from the query are extracted distinguishing between variables and instantiated terms. In step 3, the set of inputs of the service request is derived by abstracting the instantiated terms to their immediate class. For instance, if one term were Pittsburgh, it would be abstracted to City (assuming the presence of a location ontology). Step 4 is needed to handle variables. In OWL-QL variables are of class Variable, but there is no constraint on the type that they have to assume. We use the definition of the predicate in the ontology to constrain the type of the values of the variable to the most restrictive class of values that they can be assigned to. In step 5, we use the abstraction in step 4 to generate the set of outputs O. Finally, in step 6, the service request is generated by specifying the inputs and the outputs[11].

Supporting mediation

After the Broker has selected a provider, it must mediate between the provider and the requester. The mediation process depends on the Process Model of the provider which specifies what information is required and when. In theory, the Broker may just present to the requester the Process Model of the provider and limit mediation to message forwarding. But this solution is very inefficient, since it ignores the information that the requester already provided to the Broker. For example, the requester may ask the Broker to book a trip to Pittsburgh. The Broker may find a Travel Web service that asks for departure and arrival location. The task of the Broker is to recognize that arrival location information has already been specified so the Broker needs to ask the requester for the departure location only.

```
1. KB= knowledge from query
2. I= input of process
3. for $i \in I$
4.     select k from KB with the same semantic type of I
5.     if  k exists
6.         remove i from I
```

Figure -4. **Algorithm for pruning redundant information**

The algorithm for pruning redundant information is shown in Figure 4. It hinges on removing from processes inputs that should be provided by the requester, but that can be filled by the

[11] Inputs and outputs are the most important information for matching; if the query includes additional information, this could also be abstracted. Currently, we did not concern ourselves with this issue.

information the Broker already has. First, the Broker records the information provided by the query in a KB (step 1), and the inputs of the process (step 2). Next for each input i, the Broker looks in the KB for information that it can use in place of i. If any is found, i is removed from the inputs of a process.

For example, suppose that the requester's query asking for the booking of a trip used the concept `ArrivalLocation=Pittsburgh` to indicate the destination of the travel. Furthermore, suppose that the Process Model requires two inputs of type `DepartureLocation` and `ArrivalLocation`. Our algorithm would generate a Process Model in which the Broker asks only for first input (the departure city), while the second input `ArrivalLocation` will be pruned by the algorithm because it has the same semantic type of the information provided in the query.

Managing Message Passing

The last aspect of the Broker is to instantiate a message passing mechanism that allows consistent data transfer between the provider and the requester. The architecture of the Broker is shown in Figure 5. To interact with the provider and the requester the Broker instantiates two ports: a server port for interaction with the requester (since the Broker acts as a provider vis a vis the requester) and a client port for interaction with the provider (since the Broker acts as a client vis a vis the provider). The functionalities of the server port are described using OWL-S. Specifically, the Broker exposes to the requester its Process Model, Grounding and WSDL specification. The client (requester) uses these descriptions to instantiate an OWL-S Virtual Machine to interact with the Broker. Since the provider-neutral Process Model exposed by the Broker makes use of the exec extension described in section 5, the OWL-S Virtual Machine used by the requester should also include an implementation of the axioms for exec that we presented in section 5.3. The client port is also implemented as an OWL-S Virtual Machine that uses the Process Model, Grounding and WSDL description of the provider to interact with it.

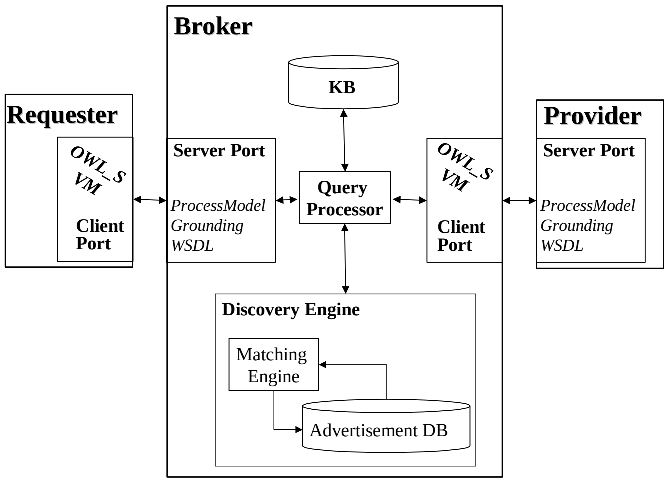


Figure -5. **Broker's Architecture**

The reasoning of the Broker happens in the *Query Processor* (see Figure 5) that is responsible for the translation of the messages between the two parties and for the implementation of the algorithms in Figures 3 and 4. Specifically, the Query Processor stores information received from the query in a *Knowledge Base* to be used as needed during the execution. Furthermore, the Query Processor interacts with the *Discovery Engine*, which provides the storage and matching of capabilities, when it receives a capability advertisement and when it needs to find a provider that can answer the query of the requester.

Conclusion

Despite the wide use of Brokers in different aspects of distributed systems, and despite the many uses Brokers can have in the discovery and mediation of Web Services, no detailed analysis of what tasks a Broker should carry on has been proposed. One contribution of this paper is to provide such as analysis. In the course of this analysis, a few challenges were uncovered, and solutions for these challenges were presented.

The first of the challenges is the “Broker’s paradox”, namely that the Broker cannot publish a Process Model that is based on a yet unknown provider before it receives a request query but the requester cannot send a query until it knows the Broker’s process Model. This paradox arises from the OWL-S (and WSDL among others) Web Service interaction specification that is based on the declarative specification of a process model that guides the requester and provider interaction. To address the Broker paradox, we extended the OWL-S Process Modeling language with an exec operation that allows the dynamic modification of the Broker’s Process Model during its execution to include Process Models of dynamically discovered new parties. We provide a formal semantics for the exec operator that is grounded in the formal execution semantics of OWL-S, and we show how it can be used as a basis for the use of OWL-S to represent the interactions of more than two parties.

A second set of challenges derives from the management of the mediation between the provider and the requester. To address these challenges, we developed a method for abstracting from a service query to a service request. We proposed an algorithm to address this issue. Furthermore, we used the knowledge provided by the requester during the interaction with the provider.

Crucially, the issues emerging with the mediation between the provider and the requester are not unique to Web services Brokering, rather they comes up in web services composition as well. In the context of Web service composition, a planner may issue a goal that it wants to subcontract. The task of the Web service is first to abstract from the specific goal to a capability description of a provider that can solve the goal, then use its current knowledge, and the goal, to interact with the provider. In current research, we are looking to integrate our work in the context of Brokering to automated composition.

Acknowledgements

We would like to thank Khalid El-Arini for his contribution to early development of this work. This work has been was funded by the Defense Advanced Research Projects Agency as part of the DARPA Agent Markup Language (DAML) program under Air Force Research Laboratory contract F30601-00-2-0592 to Carnegie Mellon University.

References

[1] Ankolekar, A, Huch, F, and Sycara, K. "Concurrent Execution Semantics for DAML-S with Subtypes." In *The First International Semantic Web Conference*, 2002.

[2] Benatallah, B, Hacid, M, Rey, C, and Toumani F. "Towards Semantic Reasoning for Web Services Discovery", *In Proc. of the International Semantic Web Conference (ISWC'03)*, Springer Verlag, Sanibel Island, Florida, USA Oct 2003.

[3] Berners-Lee, T, Hendler, J, and Lassila, O. "The semantic web" *Scientific American*, 284(5): 34--43, 2001.

[4] Booth, D., Haas, H., McCabe F., Newcomer, E., Champion, M., Ferris, C., Orchard. D. "Web Services Architecture, W3C Working Draft 8 August 2003", http://www.w3.org/TR/2003/WD-ws-arch-20030808/.

[5] Christensen, E, Curbera, F, Meredith, G, and Weerawarana, S.: *Web Services Description Language*: http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

[6] Chen, H, Finin, T, and Joshi, A. "Semantic Web in the Context Broker Architecture", In *Proceedings of the IEEE Conference on Pervasive Computing and Communications (PerCom)*, Orlando, March, 2004.

[7] DAML Joint Committee, "DAML Query Language (DQL) Abstract Specification", August 2002, http://www.daml.org/2002/08/dql/dql.

[8] Dean, M, Schreiber, G, Bechhofer, S, van Harmelen, F, Hendler, J, Horrocks, I, McGuinness, D. L., Patel-Schneider P. F. and Stein, L. A. "OWL Web Ontology Language Reference", *W3C Candidate Recommendation* 18 August 2003 http://www.w3.org/TR/owl-ref/.

[9] Decker, K, Sycara, K, and Williamson, M. "Matchmaking and Brokering." In Proceedings of *the Second International Conference on Multi-Agent Systems* (ICMAS-96), The AAAI Press, 1996.

[10] Di Noia, T, Di Sciascio, E, Donini, F, and Mongiello, M. "A system for principled matchmaking in an electronic marketplace**.**" In *Proceedings of the twelfth international conference on World Wide Web*. ACM Press, 2003.

[11] Faisst, W. "Information Technology as an Enabler of Virtual Enterprises: A Life-CycleOriented Description." In *Proceedings of the European Conference on Virtual Enterprises and Networked Solutions*, Paderborn, Germany, April 1997.

[12] Fikes, R., Hayes, P., and Horrocks, I. "OWL-QL - A Language for Deductive Query Answering on the Semantic Web." *Technical Report Knowledge Systems Laboratory, Stanford University, Stanford, CA*, KSL-03-14, 2003.

[13] Foundation for Intelligent Physical Agents (FIPA). "*FIPA Communicative Act Library Specification.*" www.fipa.org/specs/fipa00037/SC00037J.html.

[14] Jennings, N. R, Faratin, P, Norman, T. J, O'Brien, P. and Odgers, B. "Autonomous Agents for Business Process Management" *Int. Journal of Applied Artificial Intelligence* 14 (2) 145-189, 2000.

[15] Li, L, and Horrocks, I. "E-commerce: A software framework for matchmaking based on semantic web technology." In *Proceedings of the twelfth international conference on World Wide Web*, pages 331-339. ACM Press, 2003.

[16] Lu, J, Mylopoulos, J. "XIB: eXtensible Information Broker." *International Journal on Artificial Intelligence Tools*, Vol. 11, No. 1, March 2002.

[17] Drew McDermott. "Estimated-Regression Planning for Interactions with Web Services**.**" In *Proceedings of the AI Planning Systems Conference*, 2002.

[18] McIlraith, S. and Son, T. "Adapting Golog for Composition of Semantic Web Services." In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, 2002.

[19] Martin, D. L., Cheyer, A. J, and Moran, D. B. "The Open Agent Architecture: A Framework for Building Distributed Software Systems". *Applied Artificial Intelligence*, vol. 13, no. 1-2, pp. 91-128, January-March 1999

[20] Mitra, N. "SOAP Version 1.2 Part0: Primer" *W3C Recommandation* 24 June 2003. url: www.w3c.org/TR/2003/REC-soap12-part0-20030624.

[21] Motta, E, Domingue, J, Cabral, L, and Gaspari, M. "IRS-II: A Framework and Infrastructure for Semantic Web Services" *In Proc. of the International Semantic Web Conference (ISWC'03)*, Springer Verlag, Sanibel Island, Florida, USA Oct 2003.

[22] The OWL Services Coalition: *Semantic Markup for Web Services (OWL-S)*: http://www.daml.org/services/owl-s/1.0/.

[23] Paolucci, M,, Takahiro Kawamura, Payne, T. R, Sycara, K.; "*Semantic Matching of Web Services Capabilities*" In *In Proc. of the International Semantic Web Conference (ISWC'02)*, Springer Verlag, Sardegna, Italy, June 2002.

[24] Paolucci, M, Sycara, K., and Kawamura, T. "Delivering Semantic Web Services." In *Proceedings of the 12Th international conference on World Wide Web*. ACM Press, 2003.

[25] Paolucci, M,, Ankolekar, A, Srinivasan, N, and Sycara, K., "The DAML-S Virtual Machine," In *Proceedings of the Second International Semantic Web Conference (ISWC)*, 2003, Sandial Island, Fl, USA, October 2003, pp 290-305.

[26] Smith, I. A, Cohen, P. R, Bradshaw, J. R, Greaves, M, and Holmback, H. "Designing conversation policies using joint intention theory." In *Proceedings of the Third International Conference on Multi-Agent Systems*, 1998.

[27] UDDI.org "*UDDI Technical White Paper*", 2000, http://www.uddi.org/whitepapers.html.

[28] Wong, H. C, and Sycara, K. "A Taxonomy of Middle-agents for the Internet." In *Proceedings of the fifth International Conference on Multia-Agent Systesms* (ICMAS'2000), 2000.

[29] Wu, D, Parsia, B, Sirin, E, Hendler, J, and Nau, D. "Automating DAML-S Services Composition Using SHOP2." In *Proceedings of the 2nd International Semantic Web Conference*, 2003.

App E-Towards a Semantic Choreography of Web Services: from WSDL to DAML-S.

Abstract— **The relation between DAML-S, a language for the description of Web services grounded in the Semantic Web, and the growing Web services infrastructure based on WSDL is, by an large, still an open question. In this paper we describe a mapping from**

Massimo Paolucci[1], Naveen Srinivasan[1], Katia Sycara[1], Takuya Nishimura[1,2]

[1]The Robotics Institute, Carnegie Mellon University, USA

Technology Development Division, SONY Corporation, Japan

WSDL to DAML-S whose contribution is twofold: on the theoretical side it clearly shows what information is contributed by the DAML-S specification, on a more practical side it facilitates the compilation of DAML-S descriptions.

***Index Terms*—Web services, WSDL, DAML-S.**

INTRODUCTION

Existing specifications of Web services describe the primitive units of interaction. In the real world, there is a need to describe the ordering of business activities and their interactions in terms of lower level services and compose their execution. Such descriptions of Web service linkages and interactions have been described in industry using terminology such as orchestration, collaboration, coordination, composition and choreography. In this paper, we follow the definition of the Web services Architecture Working Group of the World wide Web Consortium (W3C) [11] and the definition in the recently chartered W3C Choreography Working Group [12] and call these set of activities *Choreography*. The activities comprising choreography can be different steps within a particular web service or belong to different web services. Current industry description languages, such as WSDL [4] have proven very useful in describing the interface of Web services. WSDL has recently started to be used extensively by industry (e.g. Amazon.com, Google, Acrobat to mention just a few). However, currently, natural language descriptions (understandable only by human programmers) must accompany WSDL descriptions in order to outline how to use the service (e.g. operation sequencing, state management), the participants' obligations, compositionality of results etc. It is therefore desirable to replace these natural language imprecise instructions with formal semantically meaningful and program understandable descriptions. Such precise specification could reduce the cost of businesses to integrate their processes using Web services. The Darpa Agent Markup Language for Services (DAML-S) [7] provides Web service providers with a core set of markup language constructs for describing the concepts and capabilities of their Web services in unambiguous and computer interpretable terms. The current paper reports on our work in developing and implementing a method and tool for translating WSDL descriptions to DAML-S descriptions.

WSDL provides declarative information to map abstract messages[12] into concrete messages and it expresses the bindings to specify the port where to post a message or to read the message from. In this capacity WSDL provides the foundation for composition of Web services, by providing the information that supports information exchange between Web services. But WSDL is not rich enough to specify the semantics of the composition or of the interaction protocol needed for composition. In contrast to WSDL, DAML-S, rather than describing Web services in terms of their ports or the messages that they receive, it describes the capabilities of Web services in terms of the abstract function that they provide, their Process Model, (ie what the workflow of the service steps is) and the Grounding, which describes how services interact. WSDL and DAML-S are complementary to each other: DAML-S provides the abstract information about composition of operations and information exchange, while WSDL describes how such abstract information is mapped into actual messages and how these messages are transmitted. Because of its compementarity with DAMLS, WSDL has been incorporated in the specification of the DAML-S Grounding to provide binding information.

Given the above analysis, a top-down approach, which first formalizes the abstract information exchanged in DAML-S, and then specifies WSDL to meet the needs of the DAMLS specification, seems the correct way to represent Web services. Yet many times the opposite path is more appropriate: given a WSDL specification, the problem becomes to enrich it and transform it in a DAML-S specification. This process has both theoretical and practical value. On the theoretical side, it pinpoints exactly the information that is added by DAML-S specifications, which cannot be derived from WSDL. On the practical side, the description of a mapping from WSDL to DAML-S simplifies the compilation of DAML-S documents. Indeed, WSDL specifications of Web services are widely available, providing an obvious starting point for their DAML-S formalization; furthermore WSDL specifications can be generated automatically using Java2WSDL [2].

The research contribution of this paper is the description and implementation of WSDL2DAML, a tool for the translation of WSDL into DAML-S specifications. WSDL2DAMLS takes as input a WSDL specification, and it returns as output a partial DAML-S description of the Web service. The translation is based on the assumption that there is a 1:1 correspondence between DAML-S atomic processes and WSDL operations, which allows a partial specification of DAML-S Process Models. In addition, WSDL2DAMLS generates a complete specification of the Grounding which is used to map DAML-S Atomic Processes to WSDL, a primitive DAML-S Profile, and a DAML ontology of concepts based on the data types adopted by WSDL. Such ontology can be used to complete the specification of the DAML-S Process Model and Profile as well as for mapping the new DAML-S specification into existing DAML ontologies. Such a completion requires information that is not contained in the WSDL document and it should be done through human intervention.

In the rest of the paper, we first give a brief overview of DAML-S; we then describe the algorithms behind WSDL2DAMLS and the assumptions behind such a tool. We will then describe how this tool has been used to map the WSDL of the amazon.com Web service into a DAML-S Web service. The result of this experiment is a DAML-S specification that when used

[12] WSDL supports two modes of interaction between Web services: Remote Procedure Call (RPC) and asynchronous messaging. The distinction between them is not relevant for this paper; with *message* we intend any information exchange between Web services which may or may not be implemented using RPC.

by a DAML-S processor automatically generates a client for the Web service. Finally, we conclude with an analysis of the tool and future directions of this project.

DAML-S

DAML-S is emerging as a Web Services description language that enriches Web Services descriptions based on WSDL with semantic information from DAML [6] ontologies and the Semantic Web [3]. DAML-S is organized in three modules: a Profile that describes capabilities of Web Services as well as additional features that help to describe the service; a Process Model that provides a description of the activity of the Web Service provider from which the Web Service requester can derive the interaction; Grounding that is a description of how abstract information exchanges described in the Process Model is mapped onto actual messages that the provider and the requester exchange.

The DAML-S Profile describes the Web Service capabilities, as well as additional features of Web Services such as provenance and quality of cost specifications of the Web service. The role of the DAML-S Profile is to support different modalities of discovery [10] and to support the requester decision of whether to use a given Web service. DAML-S describes capabilities of Web Services by the transformation that they produce. This transformation is described at two levels: at the information level a set of inputs are transformed in a set of outputs; at the domain level a set of conditions become true, while others become false. For example, if we consider a travel booking Web service, at the information level it may require departure and arrival information and it provides a flight schedule and a confirmation number; while at the domain level it books a flight, generate a ticket, and charges a credit.

In addition to capabilities, DAML-S Profiles provides *provenance* information that describes the entity (person or company) that deployed the service; and *non-functional parameters* that describe features of the services such as quality rating for the service.

The second module of DAML-S is the Process Model; the Process Model fulfils two tasks: the first one is to specify the interaction protocol in the sense that it allows the requester to know what information to send to the provider and what information will be sent by the provider at a given time during the transaction. In addition, to the extent that the provider makes public its own processes, it allows the client to know what the provider does with the information.

A Process Model is defined as an ordered collection of processes, where each process produces a state transformation or a data exchange with the Web service clients. The DAML-S Process Model distinguishes between two types of processes: composite processes and atomic processes. Atomic processes correspond to operations that the provider can perform directly. Composite processes are used to describe collections of processes (either atomic, or composite) organized on the basis of some control flow structure. For example, a sequence of processes is defined as a composite process of type sequence. Similarly, a conditional statement (or *choice* as defined in DAML-S) is also a composite process. The DAML-S process model allows any type of control flow structure including loops, sequences, conditionals, non-deterministic choice and concurrency.

The last module of DAML-S is the Grounding that describes how atomic processes; which provide abstract descriptions of the information exchanges with the requesters, are transformed into concrete messages that can be exchanged on the net, or through procedure call. Specifically, the DAML-S Grounding is defined as a one to one mapping from atomic processes to WSDL specifications of messages. From WSDL it inherits the definition of abstract message and binding, while the information that is used to compose the messages is extracted by the execution of the process model.

WSDL2DAMLS

The goal of WSDL2DAMLS is to provide a translation between WSDL and DAML-S. The results of this translation are a complete specification of the Grounding and an incomplete specification of the Process Model and Profile. The incompleteness of the specification is due to differences in information contained in DAML-S and WSDL. Specifically WSDL does not provide any process composition information, therefore the result of the translation will also lack process composition information; furthermore, WSDL does not provide a service capability description, therefore the DAML-S Profile generated from WSDL is also necessarily sketchy and must be manually completed. Nevertheless the outputs of WSDL2DAMLS provide the basic structure of a DAML-S description of Web services and saves a great deal of manpower[13].

The mapping produced by WSDL2DAMLS is roughly based on the following two observations.

1. *A WSDL operation is equivalent to a DAML-S atomic process:* in other words we can guess the atomic processes of the DAML-S Process Model from the operations of the WSDL description.
2. *XSD types are realized as DAML concepts:* DAML-S descriptions make use of DAML concepts to specify the content of inputs and outputs, while WSDL makes use of XSD types to specify inputs and outputs. Since the DAML-S Grounding that specifies the mapping between DAML-S Process Models and WSDL does not provide any mapping from concepts to types we are forced to assume a 1:1 correspondence between them.

The first observation provides the basic mapping between WSDL and DAML-S. It is used for both the generation of the basic Process Model and for the generation of the Grounding. The second rule generates the basic data used by DAML-S. The two rules correspond to the two main modules of translation as shown in figure 1.

Upon loading a WSDL file, WSDL2DAMLS first uses the *XSD→DAML Converter* to translate XSD types into the corresponding DAML concepts; then it uses the constructed mapping in the *Operation Converter* to translate WSDL operations into DAML-S atomic processes, generating the Grounding as well as a rough Profile. In the next two sections we describe the two modules in details.

[13] The translation of a complex WSDL document such as the specification of the Amazon Web service takes about a week of man time, where most of the time is spent dealing with the syntactic transformation from WSDL to DAML-S and only a few hours to construct the composition of processes in the Process Model and compiling the description of the Profile. Using WSDL2DAMLS the syntactic translation takes less than a minute and the programmer can finally concentrate on the Process Model composition and Profile description. We observed that the tool significantly reduced the time and effort to create the DAML-S augmented web service. We measured that the tool generated 100% of the DAML-S Grounding, 90% of the needed Process Model information.

XSD→DAML Converter

The task of the *XSD→DAML Converter* is to translate the XSD types defined in the WSDL specification into corresponding DAML ontologies. There are two design alternatives. The first is to generate DAML-S specifications that make use of XSD types and no use of DAML ontologies. This solution would prevent any type of reasoning about the concepts used in the DAML-S specification [9]. The second alternative is to force a translation to generate concepts that could be totally unrelated to ontologies available in the Semantic Web and therefore effectively useless from the automatic reasoning point of view. We chose the second solution because it allows programmers or automatic ontology-mapping programs to map the generated ontologies to existing ontologies on the Semantic Web.

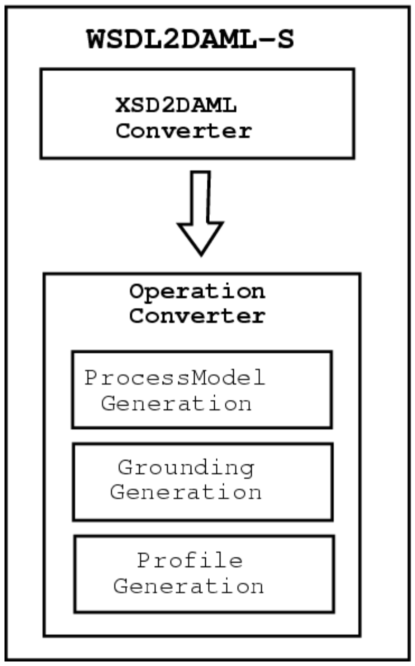


Figure 1. WSDL2DAMLS Architecture

XSD2DAML converter parses the WSDL file and extracts the XSD definitions defined between the WSDL type tags. The extracted XSD definitions are converted into DAML classes.

The conversion process is defined as follows:

- Primitive XSD types like string and integer are not converted to DAML definitions, rather they are defined directly as inputs or outputs of atomic process in the process model file.
- Complex XSD types are translated into DAML concepts whose properties correspond to the elements in the translated type.

This translation generates correct DAML ontologies, which, as described above, need to be mapped onto existing ontologies in the Semantic Web to become useful for automatic process composition.

Operation Converter

The conversion of the WSDL operations into DAML-S processes is based on rule 1 mentioned above where the basic idea is that WSDL operations map into DAML-S atomic processes, with the result that the WSDL **`portType`** description defines a primitive Process Model. The complete mapping is described in figure 2.

The mapping of WSDL Operations into DAML-S Atomic

Processes is realized in the following way:

- *The name of the operation becomes the name of the corresponding atomic process*
- *The inputs messages of the operation become the inputs of the atomic process*
- *The outputs and faults messages of the operation become the outputs of the atomic process*

Once the atomic processes are generated WSDL2DAMLS proceeds with the specification of the grounding. This specification is quite straightforward since all the pieces are in place and we make explicit the link between the operations and their parts with atomic processes and their parts. In contrast to other DAML-S modules, the Grounding is completely specified during the translation and it should not require any modification.

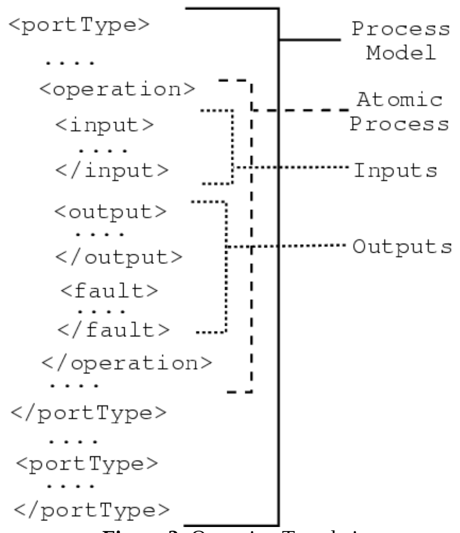


Figure 2: Operation Translation

One issue with the mapping described here is that DAML-S Atomic Processes may include inputs and outputs that are local to the process, and not reflected in the WSDL document. These inputs and outputs should be added by a programmer when she refines the DAML-S description. Furthermore the list of atomic processes may not be complete since DAML-S allows the addition of atomic processes that do not generate any message, but provide (partial) visibility on the internal processes of the Web service.

Service Profile Generator

The last translation performed by WSDL2DAMLS is the generation of the Service Profile; the result of this transformation is a skeleton Profile. DAML-S Profiles consist of three sections: the first one is *provenance* information that describes the entity (person or company) that deployed the service; the second consists of *non-functional parameters* that describe features of the services such as quality rating for the service and finally it provides a description of the capability/functionality of the service in terms of the inputs it receives, the outputs it generates, the furthermore, the preconditions that should be satisfied for the Web service to execute and the effects that will result as consequence of such execution. Since WSDL provides only input and output information, the rest of the DAML-S profile must be completed manually.

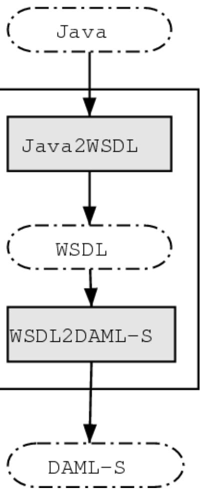


Figure3: From Java to DAML-S

One important contribution of the work described here is that it provides the basis for an automatic generation of DAML-S specifications starting from the Java implementation of the Web service. The complete process is described in figure 3. The first step is to use JAVA2WSDL [2] to generate the WSDL specifications directly from the Java code, and then compile the WSDL specification into the corresponding DAML-S specifications using the tool described here.

Case Study: Amazon Web Service

Amazon.com has exposed a Web service [30] that allows to browse Amazon's catalogue and to fill a shopping cart with books that a customer would like to buy. Specifically, Amazon's Web Service allows 16 different varieties of searches (e.g. author search, actor search, ISBN search etc.) to browse and find items in their inventory, and five shopping cart operations including getting and clearing the shopping cart, add, removing and modifying items from the shopping cart.

The WSDL description of Amazon's Web service contains the 21 operations described above, 66 types, and 42 input/output messages (there is no use of fault messages). Using WSDL2DAMLS we were able to compile the WSDL specification in the corresponding DAML-S specification, which resulted of course in the 4 files describing the different aspects of DAML-S: Profile, Process, Grounding and a file that represents the XSD types into DAML concepts. After this translation a programmer is left with only three tasks to complete the DAML-S specification: the first one is to analyze the interaction flow between Amazon's Web service and its clients to complete the Process Model; the second one is to map the concepts derived from the

XSD types into concepts in existing ontologies; finally the third task is to complete the specification of the DAML-S Profile adding provenance information and the non-functional parameters.

The first task proves to be quite simple since the interaction flow is easily extracted from the control flow described in the documentation of the Web service, and it resulted in the addition of 5 composite processes. The other two tasks are more challenging because the mapping between the types used in the WSDL file are quite arbitrary and do not match with any ontology for bibliographic information such as the Dublin Core [8]. For example, one type is called *AuthorArray*, which has properties specific to both author and Amazon web service, and does not have any obvious semantic status. The Profile is also difficult to generate because the concepts used in the Profile are also arbitrary as a consequence of the ontologies used. We believe that the arbitrariness of the concepts used is a consequence of the general attitude in the construction of the Web service, which is targeted to human users rather than to automatic interaction with other Web services which is what DAML-S attempts to facilitate; furthermore, it is a consequence of the expectation that any activity involving the Web service will be mediated by programmers that will hardcode the connection between their Web services and Amazon's. Once DAML-S or other semantically oriented technologies become widespread, more principled use of ontologies can be expected.

Conclusions

The contribution of this paper is twofold: on the practical side it describes a tool for generating DAML-S descriptions of Web services starting from its WSDL description; on the theoretical side it highlights the contribution provided by DAML-S to the description of Web services.

Upon deciding to extend a WSDL specification into a DAML-S specification, a programmer can use WSDL2DAMLS described in this paper to generate the DAML and DAML-S code. The result of the translation is a first approximation of the final DAML-S description; which contains all the DAML-S modules, but still requires work to be completed.

The first task of the programmer is to map the ontologies generated into existing DAML ontologies. Our example shows that such mapping may prove very challenging since the XSD types may not have any ontological status.

The second task is to construct the composite processes that complete the Process Model. These composite processes specify the order of execution of the WSDL operations to implement the expected interaction protocol.

The Process Model may contain some atomic processes and information flow that are private of the Web service and not shared with its clients. Since this information is not reflected in the WSDL specification, the third task is to add it to the DAML-S description under construction.

Once the Process Model is completed, the programmer should complete the DAML-S Profile. Specifically, she needs to select which input and output information better specifies the service provided. This process may require the removal of some inputs and outputs and the generalization of others. Furthermore, the programmer needs to add other two pieces of information; the first one is provenance information that specifies the company or institution that provides the service; the second one is information about non-functional parameters and service classification.

The work described in this paper may be generalized to include Web services information contained in UDDI [13] as well as BPEL4WS [5]. While this mapping would be a natural extension of the work presented here, some predictions on its results can already be drawn. The addition of this information may add some automatism to the translation: UDDI provides provenance information possibly some functional parameters, while BPEL4WS provides processes composition. Yet, neither of the two provides semantic information so the XSD→DAML mapping described above would still be needed with the relative mapping to existing ontologies; furthermore, neither of the two provides capability description; which would still require the programmer intervention.

Bibliography

[30] Amazon.com: *Web Services V.2.0*: http://associates.amazon.com/exec/panama/associates/ntg/browse/-/1067662/ref=gw_hp_ls_1_3/.

[31] Apache Software Foundation: *WSIF*: http://ws.apache.org/wsif/.

[32] T. Berners-Lee, J. Hendler, and O. Lassila.: *The semantic web*.: Scientific American, 284(5):34--43, 2001.

[33] E. Christensen, F. Curbera, G. Meredith, and S.Weerawarana.: *Web Services Description Language (WSDL)*: http://www.w3.org/TR/2001/NOTE-wsdl-20010315 2001.

[34] F. Curbera, Y. Goland, J. Klein, Microsoft, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana: *Business Process Execution Language for Web Services, Version 1.0*: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[35] DAML Joint Committee: *Daml+oil (march 2001) language*: http://www.daml.org/2001/03/daml+oil-index.html, 2001.

[36] DAML-S Coalition: *Daml-s: Web service description for the semantic web*: In ISWC2002.

[37] Dan King: *The Dublin Core Element Set Ontology*: http://www.daml.org/ontologies/201.

[38] M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks. *The relation between ontologies and xml schemas*. In Electronic Trans. on Artificial Intelligence, 2001.

[39] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara.: *Semantic matching of web services capabilities*. In ISWC2002, 2002.

[40] W3C Web Services Architecture Working Group: *Web services Glossary*: http://www.w3.org/TR/2002/WD-ws-gloss-20021114/.

[41] W3C Web services Choreography Working Group: *Charter*: http://www.w3.org/2003/01/wscwg-charter

[42] UDDI: The UDDI Technical White Paper: http://www.uddi.org/, 2000.

Appendix F

An Extension of the Service Oriented Architecture to Support Resource Discovery in Virtual Organizations[14]

Massimo Paolucci[†15], Xiong Liu[‡], Naveen Srinivasan[*], Katia Sycara[*], Paul Kogut[+]

[†] *Future Networking Lab, DoCoMo Communications Laboratories Europe Munich, Germany*

[‡] *School of Information Sciences, University of Pittsburgh, Pittsburgh, PA, USA*

[*] *Software Agents Lab, Carnegie Mellon University, Pittsburgh, PA, USA*

[+] *Lockheed Martin Integrated Systems & Solutions, King of Prussia, PA, USA*

Abstract

Web services are often deployed within the limited scope of single organizations' intranets. This type of deployment guarantees control over the infrastructure and the security and safety of the services, but it also prevents other parties from accessing the information that these services provide. In this paper we propose an extension of the basic protocol of the Service Oriented Architecture that supports discovery of Web services across organization boundaries. We show how this protocol can be used in emergency response operations that involve a number of different organizations such as hospitals, police, government agencies that normally would not work together nor share information. The protocol that we suggest supports the rapid creation of a virtual organization so that time critical information sharing can be guaranteed. We provide a detailed discussion of both the architectural and implementation considerations, and we provide an empirical evaluation that shows that our protocol scales with the number of Web services and the number of organizations involved.

1.Introduction

Virtual organizations form when different parties gather to solve a problem that none of them would be able to solve otherwise. Emergency response is one such case, where different government agencies and private corporations contribute their knowledge, skills and capabilities to prevent an accident from happening or to address the consequences of such an accident.

One limiting factor that stems the creation and the operation of virtual organizations is the difficulty to allow free information flow across organizational boundaries. This limitation is due to many aspects such as the corporate culture of the different organizations, security concerns and organizational policies. But one fundamental problem is that the IT infrastructures of the different organizations are often unable to work together and interoperate. Therefore, even when the policies and intentions of the different virtual organizations would allow information to flow freely, information sharing would still be hampered. Furthermore, it is often unfeasible to invest any effort in enabling the interoperation of the IT infrastructures of the different organizations since the reasons for forming the virtual organization may disappear long before the interoperation effort is completed. Rather the creation of virtual organizations crucially depends

[14] This paper is an extended version of a paper titled "Discovery of Information Sources across Organizational Boundaries" published by the same authors at the IEEE International Conference on Service Computing 2005, held in Orlando, FL, USA in July 2005.

[15] Massimo Paolucci participated in this work as a member of the Software Agents Lab, Carnegie Mellon University, Pittsburgh, PA, USA, before joining DoCoMo European Laboratories.

on an infrastructure that enables the automatic interoperation of all the composing organizations in a plug and play fashion.

Web services technology provides the first step toward such automatic interoperation. Standards like WSDL and SOAP provide a way to abstract implementation and interaction details to support the automatic creation of clients facilitating interaction between services. Furthermore, UDDI provides the standard for a discovery registry that allows loose coupling between Web services and their clients. When all combined Web service technology provides the backbone of an infrastructure for discovery and provisioning of services that goes under the name of Service Oriented Architecture (SOA).

Despite its contributions, Web service technology has infrastructural limitations. Specifically, Web service discovery is limited to the scope of the SOA infrastructure in which the service is deployed. Specifically, the visibility of the service is limited by the scope of the UDDI registry that usually ranges only within the organizations' boundaries. Therefore, while in principle a Web service could be discovered, its use is restricted within the space of the organizations' intranets, but rarely exposed for a wider use. Such a limited visibility is problematic for the creation of virtual organizations where the services offered by one organization should become accessible to other organizations. The realization of virtual organizations requires the opening of the IT infrastructures to its participants, so that information can flow easily within the virtual organization.

A way to extend SOA across organizations is to allow Web services to cross register with the UDDIs of the other partners within the virtual organization, or to allow UDDI servers to exchange registrations of services. The UDDI standard specification provides such a replication schema, and the associated API, that allows different UDDI servers to exchange information about their content. This solution is problematic because it results in multiple copies of registrations of Web services across the Virtual organization. Such copying has a number of disadvantages: first, every detail of the description of the Web service should be maintained since every simple change, like the Web service moving to a different port, requires an update of all the copies of the registration of the Web service. A number of lazy and eager replication schemata have been proposed to address this problem [21], but they all highlight a trade-off between performance of the registry and the ability to have up to date information. Second, if a Web service does not register with all the UDDIs within the virtual organization it may not be visible and discoverable. Finally, whenever the virtual organization is dismantled copies of the registrations of Web services may still be present in the partners' UDDI registries allowing Web services to be visible and potentially accessible by unauthorized users. Despite its problems, replication has the advantage of adding redundancy to the systems so that even though one UDDI server may no longer be available, the services that register with it may still be reachable using the information stored by other servers.

In our application domain, replication is problematic because we need to be able to set up an overarching infrastructure for the whole virtual organization in a very short time, and we need to be able to modify the virtual organization dynamically. As the application scenario described in section 2 shows, even a change in wind direction may require a change in the virtual organization. In such a context, replication will be just too expensive; furthermore, most of the service descriptions that would be exchanged by the UDDI servers may be useless, since the corresponding services may never be used

Instead of replication, in this paper we suggest a schema that allows UDDI registries to exchange meta-information about the Web services that they have registered. Following this solution, a Web service registers with only one UDDI, as it normally would. Furthermore, any application that is also a UDDI client would inquiry only one UDDI. But upon receiving an inquiry, the UDDI may solve it directly, if it has the required information, or it may inquiry other UDDIs to find the required service, and then report to the client what it found.

The difficulty of this solution is that UDDI registries themselves should discover each other; furthermore, there is a need of limiting the querying among UDDI registries since in general all these queries but one will succeed; furthermore, the amount of queries will increase polynomially with the increase of partners in the virtual organization. Therefore, in order to realize such a distributed UDDI schema we need also to provide a selection mechanism that allows a UDDI to query only those UDDIs that are most likely to have the required service. The latter problem can be solved by providing an indexing mechanism that allows the specification of what kind of Web services are registered with a UDDI registry.

In this paper, we propose a way to extend SOA to virtual organizations. Our proposal, which we call Extended SOA, or ESOA in short, is based on the observation that UDDI registries are themselves Web services also. In ESOA, the virtual organization establishes a UDDI registry, called Main Registry, that collects the registrations of the partners' registries and answers queries about which registry is more likely to contain a specified type of service. To provide an effective discovery mechanism that does not depend only on syntactic similarity, as currently supported by UDDI, but extends to matching the deeper semantics of what the service does, we rely on OWL-S. OWL-S is a language for the description of Web services that is based on the OWL language for the semantic web. By providing a representation of services that is grounded in OWL ontologies, an OWL-S-based discovery process exploits the logical relations between terms that are specified in those ontologies, as has been proposed in many papers [1, 3, 4].

Since the ontologies are used to specify the concepts that are needed to represent Web services, they also provide a summary of what type of Web services are available within the registry. Therefore, we use ontologies in ESOA to provide an effective indexing mechanism for UDDI registries. Such an indexing can be used to support the selection of the most appropriate UDDI registry in the Main UDDI.

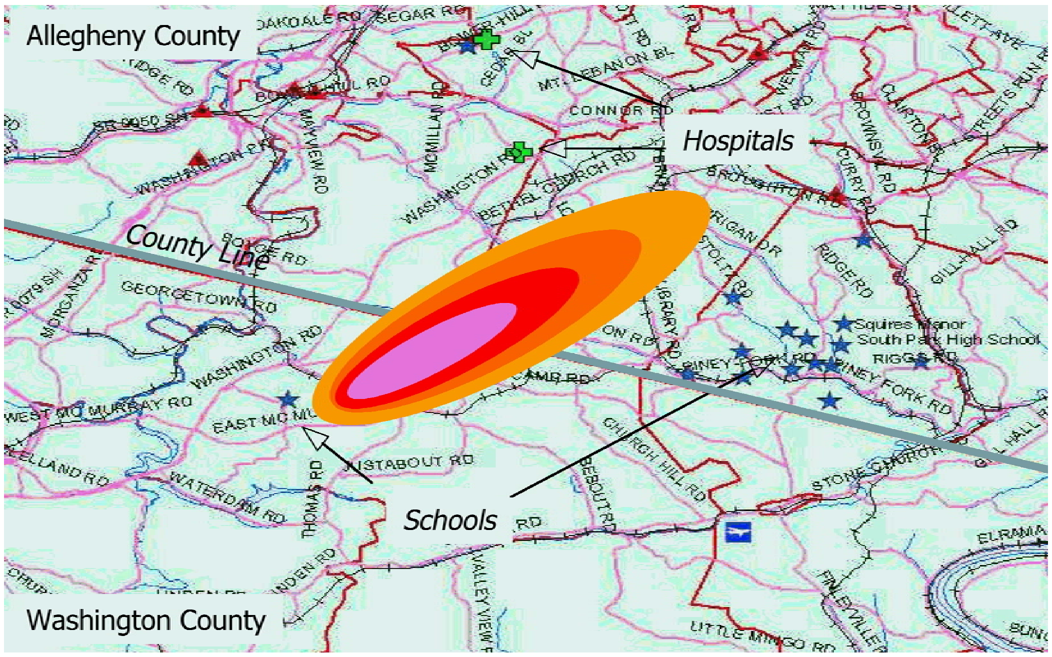


Figure 9: A map of the emergency area

The rest of the paper is organized as follows: in section 2, we will give a motivating example of ESOA; in section 3, we will provide a more detailed description of SOA and the replication mechanism provided by UDDI; in section 4, we will discuss ESOA; in section 5, we show how ESOA can be implemented using UDDI; in section, we will discuss the semantic matching, and the changes that it requires to UDDI; in section 7, we will relate ESOA with similar approaches; in section 8, we will discuss the implementation of a discovery mechanism for ESOA; in section 9, we will present a performance evaluation; and finally, in section 10, we will conclude.

2.A Motivating Example: Supporting Emergency Operations

Emergency operations to react to environmental disasters require many different organizations to work together to limit the damage and address the emerging problems. One such case may occur when a Chlorine gas spill is detected after a train accident near the border between Washington and Allegheny counties in the state of Pennsylvania, of the USA[16]. Such a scenario is shown in Figure 9, which displays the predicted diffusion of the gas spill as well as the locations of schools and hospitals in the area.

Police, firefighters and paramedics are typically the first responders in the case of such accidents, on the bases of their initial assessment, emergency offices at all levels of the

[16] The scenario reported was provided by the Pennsylvania Emergency Management Agency (PEMA) to evaluate how Agent and Web service technology may be used to facilitate the management of emergencies.

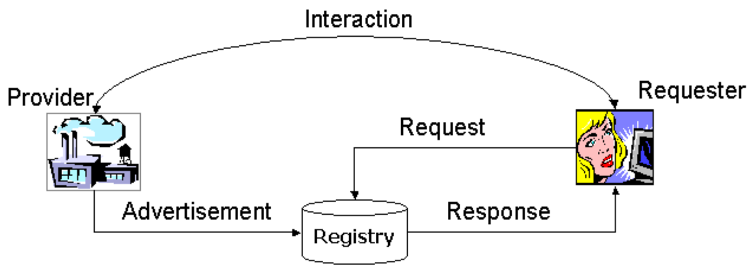


Figure 10: Basic Service Oriented Architecture

government structure may be involved, starting from the municipal level to the federal level when the disaster is of major proportions. In turn the emergency offices should alert other government agencies such as the Environmental Protection Agency (EPA) that provides experts on management of environmental damages, and private corporations that may provide skills that are needed to address the problem.

In the case of the emergency shown in Figure 9, computational models of the spread of the gas spill, shown by the ovals in the center of the picture, show that the gas is expected to cross over County lines involving a number of municipalities in two different counties. As the crises develops, the projected dispersion shows that the gas spreads near schools and hospitals, those schools and hospitals will also need to be alerted and possibly evacuated if the wind direction changes. In turn, the eventuality of an evacuation requires that the school boards are also alerted to be aware of the problem and possibly give the order of schools evacuation. Similarly county and state-wide health organizations should be alerted to address potential hospital evacuations.

Emergency operations to address the problems described in our scenario are typically guided by protocols that specify what each organization should do within the first 20 minutes from the accident. By that time, an emergency plan should be in place and other organizations may be involved on a per need bases. In those 20 initial minutes a communication infrastructure that allows free information flow should also be put in place so that each one of the organizations is aware of the development of the situation and of what role it is playing in the overall operations.

The simple scenario above shows how different organizations may be called to react to the problem even though these organizations may not know how to work together. At its minimum, addressing such a scenario involves two counties, and numerous police departments and firefighters squadrons. In addition, it may require the involvement of school boards, EPA, railroad operators and so on. Opening up the IT infrastructure of the different organizations is essential to allow quick and proactive information diffusion, since lack of communication and information usually results in lack of coordination and ultimately in a disaster which could be avoided otherwise. As Web services become the linchpin of the IT infrastructure of the organizations involved in the emergency operations, facilitating the information flow also requires facilitating the interaction between Web services provided by different organizations.

2.1Requirements

The scenario above highlights the requirements that should be satisfied by any infrastructure that aims at facilitating information flow for emergency management, and more generally for virtual organizations. The overarching goal is to facilitate the interaction of Web services across organization boundaries, but it can be easily recognized that this goal should be achieved in such a way that a number of requirements that address implementation issues, performance issues, the information that is available to the members of the virtual organization and security issues.

On the organizational side, the virtual organization changes dynamically, with new organizations joining and others leaving. As shown above, even minor changes in the weather conditions may trigger changes in the virtual organization. The infrastructure should support the dynamic changes of the virtual organization over time, without imposing an eccessive overhead.

On the implementation side, the ability to access information available within other organizations should minimize software changes. Specifically, it should not require any change in the clients that users and applications use to query UDDI registries. Any such change would essentially prevent the applicability of such a solution. Essentially, registries need to interact with their clients using the UDDI Standard API [12].

On the performance side, any solution should require minimal overhead and scale with the number of parties involved. When responding to emergencies or to similar problems, there is no time for massive data transfers as required by the replication mechanisms, rather there is a need for lean protocols that allow parties to dynamically join or leave the virtual organization. Furthermore, since it is impossible to predict how many parties will be involved in the virtual organization, the solution adopted should scale up with the number of parties involved. Finally, requests for services should be directed to the registry that is more likely to provide that service, rather than being broadcasted to all existing registries.

On the information side, the solution adopted needs to support requests for services that flexibly match the service advertisements, since members in one organization do not exactly know the services provided by other organizations. It is paramount that the requesters are able to express what capabilities they seek and find the corresponding services.

On the security side, virtual organizations require complex trade-offs. On one hand, the members of the virtual organization need to be open and exchange information, on the other hand, the security of the IT infrastructure and of the information provided by the different organizations should be guaranteed. The emergency operation should not present a security loophole where anybody can access any of the services provided even though he or she has no role in the established virtual organization.

The infrastructure that we will present in the rest of the paper will strive to satisfy these requirements with the exception of the security requirement. While we recognize that security is a fundamental feature of the system, it also requires additional infrastructure that is beyond the pure discovery processes that is the contribution of this paper. Although we do not address the security problem in this paper, the solutions proposed in [13] could be used in combination with ESOA.

3.Service Oriented Architecture

The support of emergency operations like the one described in the previous section, requires an infrastructure for web services that allows Web services to discover each other and interact. The blueprint for such an infrastructure is defined by the Service Oriented Architecture (SOA).

SOA, as defined by the Web services Architecture Working Group at the W3C[17], identifies three stakeholders: the provider of the service, the requester of the service and a registry, typically UDDI, that has the responsibility of finding the best provider that matches the requester's requirements.

A detailed view of the basic interaction protocol that is used within SOA is shown in Figure 10. The provider advertises with the registry a description of the services that it provides. The requester sends a request to the registry asking it to find the services that satisfy a set of requirements. The registry is responsible for finding those services and it responds to the requester with a report on the services that it found. Finally the requester selects the best service for the specific task and begins the interaction.

UDDI is the de-facto standard SOA registry, partly because it is a standard at the OASIS standardization body, and partly because major players in the software industry from IBM to Microsoft to Oracle and SAP, to mention only a few, heavily invested in the development of UDDI registries and participate in the work of the UDDI Technical Committee. UDDI supports SOA by providing a publish API that is used by the provider to advertise Web services, and an inquiry API that allows requesters to search for Web services and to retrieve information about the registered Web services.

Web services are described in UDDI by a set of data structures that are used to specify the business that deployed the service, the service itself and all the bindings of the service. Furthermore, UDDI allows the expression of TModels that support the expression of properties of UDDI entities such as businesses, services and bindings. For example using TModels it is possible to express whether a given service is a bookselling service or a stockbroker service, or any other type of service.

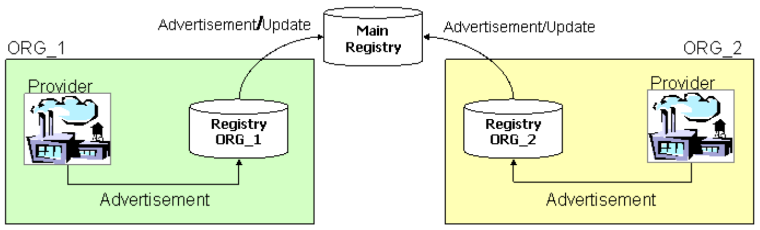
Since different registries may know of very different services, UDDI supports also a replication API that allows multiple registries to exchange information about the services that they registered. The UDDI replication protocol assumes that each registration has an "owner" registry that periodically pushes updates to other registries. Furthermore, UDDI requires the existence of a special registry, called key-generating registry, which generates unique keys for each entity registered in one of the replicating registries. The replication API allows UDDI registries to extract updated information from other registries as long as they are partner of the same key-generating registry.

The replication mechanism in UDDI can be used very effectively to copy Web services registrations from registry to registry, but, as we pointed out before, it does not solve the problems of virtual organizations since it has a high computation overhead that virtual organizations responding to emergencies cannot affort.

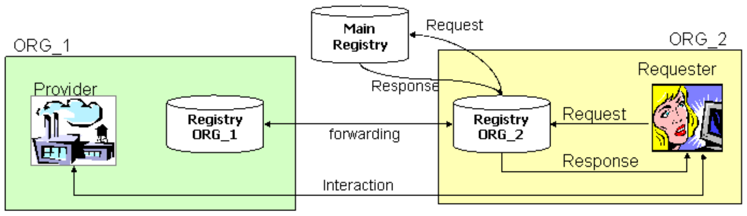
4.ESOA: Extended SOA

To support the need of virtual organizations, each organization's infrastructure should be extended to span across organizational boundaries, without requiring an organization's UDDI registry to copy its Web services descriptions, to the other UDDI registries. A way to extend the

[17] Web Services Architecture Working Group Technical Note: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.



Advertising in the Extended SOA



Inquiry in the Extended SOA

Figure 11: Protocols in Extended SOA

span of registries to cross-organizational boundaries is to think UDDI registries as Web services themselves. As a consequence we can deploy a UDDI registry that acts as a registry of registries to support the discovery of registries belonging to different organizations, rather than the discovery of services.

This approach suggests an extension of the original SOA protocols to Web services registries is what we call *Extended SOA* or ESOA. ESOA has four stakeholders. The first two stakeholders are service requesters, service providers that are defined as in SOA. The third stakeholder of SOA is a registry; instead in ESOA, the third stakeholder is a set of registries that store advertisements of providers and match them with the requirements of the requesters. Finally, the fourth stakeholder is a *Main Registry* that collects descriptions of the type of advertisements collected by the registries.

ESOA extends SOA because the three protocols that underlie the SOA architecture, namely *advertisement*, *discovery loop* and *interaction*, are extended to registries as well as to services. Figure 11 provides an overview of the ESOA protocols.

During the advertisement phase, shown in the upper part of Figure 11, each registry transfers to the Main registry information about itself and the type of services that it has. For example, a registry that is adopted within the HR department of a company will advertise to the Main registry that all its services are about Human Resources. Crucially, no information about the specific services is exchanged; such information remains private to the HR department. Upon receiving the advertisement of a new service, a registry, such as Registry ORG_1 in Figure 11, verifies whether the new service extends the types of services available, and when that is the case, the registry updates its registration with the main registry to reflect the additional capabilities that just acquired.

During the discovery phase, shown in the bottom part of Figure 11, the query from the requesting application is propagated by the organization's registry to the main registry and possibly to other registries. Specifically, clients query their local registry looking for a given service type. If the registry knows of services that fit the requirements of the application, it will report on those services. In addition, as shown in Figure 11, the registry queries the Main Registry to discover what other registries may have the type of services that the application needs. The Main Registry uses its knowledge of the available organizations to suggest the registries that are likely to discover the needed service. In the example in Figure 11, the Main Registry may suggest Registry ORG_1. At this point, Registry ORG_2 sends the query that it received from the application directly to Registry ORG_1 which performs the discovery process, locates the best set of services, and reports them back to Registry ORG_2. In turn, Registry ORG_2 reports these services, and all the other services that it discovered, to the application that generated the original query. Finally, the application selects the best provider and invokes that provider directly. Such invocation in ESOA is consistent with the invocation process in SOA, where applications contact providers directly. The only complication in the ESOA case may emerge from the policies and security constraints that the provider's organization imposes on the use of its own services.

The use of ESOA has a number of advantages. As shown in Figure 11, it allows SOA to span across organizational boundaries supporting discovery of services within other organizations. As a result, ESOA supports the creation of virtual organizations. The second advantage is that ESOA requires very limited additional infrastructure. The only requirement is the additional Main registry. Finally, by avoiding copying of advertisements it reduces the redundancy between the registries, and it does not require any overhead when the virtual organization is dismantled.

5. Using UDDI to implement ESOA

UDDI plays an essential role in SOA, therefore it is obvious to use UDDI in ESOA, furthermore, one of the main objectives of our SOA extension is to re-use as much as possible existing components so that the different organizations can interact without being force to modify their own infrastructure. To provide a blue-print for the use of UDDI in ESOA we need to solve three problems: first, we need to provide an extension of the UDDI architecture that allows registries to interact; second, we need to provide a general description of how registries are represented in the Main UDDI; third, we need to provide a key management mechanism that guarantees uniqueness of keys and that allows the clients to find the information that they seek.

5.1 UDDI Architecture for ESOA

The interaction process that ESOA requires from UDDI registries is different than the interaction process that is expected in SOA. Whereas in SOA registries are independent and, with the exclusion of the UDDI replication mechanism, do not interact with each other; ESOA explicitly requires that an organization's registry queries another organizations' registries and the Main UDDI. Therefore, a UDDI registry, while acting as a server for the services within an organization, also acts as a client of the Main UDDI and potentially of other registries in the virtual organization.

The resulting architecture for UDDI registries in ESOA is shown in Figure 11. A new client port is added to the UDDI registry. This client acts as a standard UDDI client and performs two operations: the first one is to push a description of the UDDI registry to the Main UDDI, and to update such a description as a function of the services that become available in the registry. The second operation is to query other UDDI registries to find new services.

Only two additional changes are required to implement an ESOA enabled UDDI. The first one is to provide a message that notifies of the existence of a Main UDDI and provides address information of such a UDDI. The effect of receiving such a message is the registration of the Main UDDI within the registry tables, and the automatic registration with the Main UDDI. An organization can belong to multiple virtual organizations by being notified of the existence of other multiple Main UDDIs to use during the discovery process. The second change required is the addition of a table for a key management system as described below.

The use of the client port in the UDDI registry has the effect of hiding the invocation of other UDDIs to find desired services. Since such an invocation is totally transparent to the requester, the UDDI client adopted by the requester is the standard UDDI client. Additional complexities such as querying the Main UDDI or the key management, discussed below, need to be implemented only at the registry level not at the client level.

5.2 Representation of registries in the Main Registry

To use the Main UDDI as a registry of registries, we need to provide a representation schema of the contents of UDDI registries. We take the approach that every UDDI registry is essentially a Web service, therefore we can exploit the normal representation provided by UDDI. Specifically, each one of the organizations' UDDIs is specified as a business entity and a business service in the Main UDDI. The business entity is associated with a TModel that specifies that the business is a registry. Furthermore, the publish and the inquiry ports of the registry are specified as binding templates associated with the business service.

The representation of registries can be enhanced by providing a set of keywords that facilitates the discovery of such a registry. As described below in section 6, the keyword system that we propose is based on the URIs of the ontologies used to describe the services registered with the registry. We therefore created a specialized TModel within the Main UDDI, named *ontology TModel,* which is used in to register the URI of the ontologies.

The representation that we provide allows the use of the standard UDDI technology without any modification. Standard UDDI clients can be used to interact with the Main UDDI and retrieve the registries using business keys, the ontology TModel key, business service keys, and binding template keys. Furthermore, to locate the registry that is associated with a given ontology, or more generally a given keyword, we can exploit the UDDI native keyword search mechanism. No change to the UDDI API or to the UDDI clients is required to implement the Main UDDI.

The solution that we adopted allows a great degree of flexibility to the representation of services and organizations in the overall virtual organization. Adding a new organization to the virtual organization is straightforward, since the organization can just publish its UDDI registry with the Main UDDI and associate that publication with the URI of the ontologies used to describe the services in the registry. Furthermore, whenever a new service that adds new ontologies to represent new capabilities available to the virtual organization is registered with one of the UDDI registries, the record of that UDDI registry is updated in the Main UDDI by adding the URI of the new ontologies to the registry's description. The flexibility provided by our solution satisfies our requirements put forward in section 2.1, allowing the virtual organization to change dynamically while the conditions in which the virtual organization operates and problem evolves.

During Web service discovery, a registry extracts all the ontologies that are needed to describe the requested service, and then it queries the Main Registry to locate all the registries that loaded those ontologies. The discovered registries will then be queried to find the requested service, knowing that they have the knowledge needed to interpret the description of the service required.

5.3 Key management

Entities such as service and business descriptions in UDDI are associated to unique keys, and such keys can be used to extract information from a UDDI registry about the corresponding entity. Since ESOA does not assume to copy anything between registries, we need to provide a key management mechanism that allows a registry to refer to keys on other registries.

To address this problem we propose that ESOA enabled UDDIs maintain a table that maps keys in their own registry to keys in another registry. Specifically, when a UDDI registry discovers a service, or any other entity on another UDDI registry, it creates a new key and adds a new entry in the key table associating the new key to the foreign key and to the port registry that contains the information on that key. As a consequence any "`find_XXX`" operation in UDDI can be performed in of two ways: first, the key specified in the find operation is used to extract the corresponding foreign key; if such a key is found, the corresponding UDDI registry is inquired. Otherwise, if the key specified in the query does not exist, the information is searched in the registry DB following the standard UDDI inquiry.

The advantage of the solution adopted is that it guarantees the uniqueness of keys since the registry generates a new key for each foreign key it receives.

6.Exploiting the Semantic Web in ESOA

The success of ESOA, as well as the success of SOA hinges on the ability to provide an effective discovery mechanism. The problem of service discovery is that there is always a discrepancy between the advertisement of a service and the request of the same service. The reason for the discrepancy is that the provider and the requester of the service may have very different perspectives on the service advertised. The discovery mechanism should therefore abstract the apparent differences between the advertisement and the request to recognize deeper similarities between the meaning of the request and the meaning of the advertisement.

Rather than relying on syntactic matching, we want to rely on semantic matches that have been proposed in the context of the Semantic Web. The Semantic Web, and more specifically OWL [7], provides ontologies that can be used to specify the capabilities of services. Ontologies are conceptualizations of a specified domain, that provide a set of terms naming objects and relations between objects in the domain; furthermore ontologies are defined in function of a logic that allows the derivation of consequences of the statements specified in the ontology.

The use of ontologies enables a more powerful discovery mechanism that recognizes the relations between the advertisement and the request even when the syntactic matching fails. The reason of the added power is that the logic behind the ontologies exploits relations between terms that are not available to purely syntactic methods.

The service representation that we adopt is based on OWL-S, which is an OWL ontology for describing Web services. OWL-S provides multiple views of the capabilities of services. The first view is to represent these capabilities in terms of the information transformation that is produced by the execution of the service. Such a transformation requires a set of inputs and returns a set of outputs. Related to the information transformation is a state transformation from an initial state to a final state, which is described by the transformation from preconditions for the execution of the service, to the effects that result from the service execution. The second view of capabilities is characterized by a set of additional features of the service, such as quality of the service provided, or its security capabilities and requirements.

Formally, an OWL-S Web service is described by a tuple <τ,π,C,S,I,O,P,E> where τ is the *service type* as defined by an OWL class in a service ontology, π is the *type of products* that the web service produces as defined by an OWL class in a service ontology, I and O are set OWL classes that specify the service *inputs* and *outputs*, and finally P and E refer to SWRL [3] predicates that specify *conditions* on inputs and outputs, and how they affect the environment of the Web service. In addition OWL-S supports the specification of *service categories* and *service parameters*, indicated in the tuple as C and S respectively. Service Categories and Service parameters can be used to specify additional qualities of the service. Service categories are used to refer to existing taxonomies that may not be codified in OWL, such as NAICS [8]. Service parameters can be used to specify additional features of the service.

```
   <profile  rdf:ID=" BravoAir ">
     <serviceName>BravoAir </serviceName>
     <contactInformation rdf:resource="BAco"/>
     <serviceCategory rdf:resource="Airline"/>
     <product rdf:resource="FlightReservation"/>
     <serviceCategory
rdf:resource="NAICS_Airline"/>
     <hasInput rdf:resource="Dep_Airport"/>
     <hasInput rdf:resource="Arr_Airport"/>
     <hasOutput rdf:resource="Reservation"/>
   </profile>
```

Figure 12: XML serialization of OWL-S

The tuple representation is of course isomorphic to the more common XML serialization of OWL-S shown in Figure 12 that corresponds to the tuple `<Airline,FlightReserv,NAICS_Airline, _,{Dep_Airport,Arr_Airport},Reservation,_,_>`.

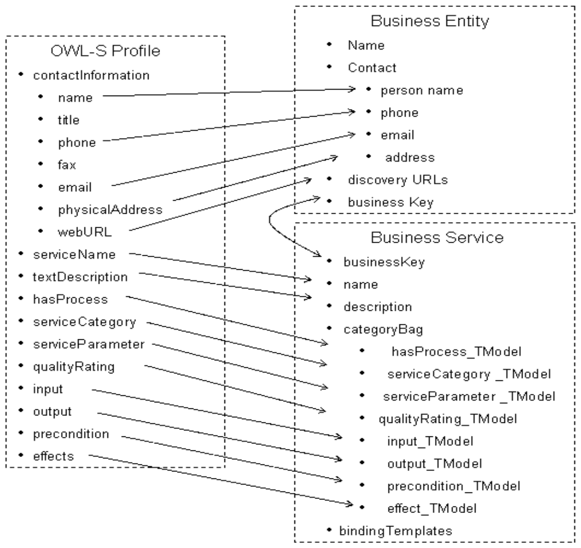


Figure 13: OWL-S to UDDI mapping

The discovery process for OWL-S exploits the subsumption inference provided by the semantics of OWL. Specifically, if R is the requested service that is specified by the tuple $<\tau^R,\pi^R,C^R,S^R,I^R,O^R,P^R,E^R>$, the discovery process detects a match with a provided service P, whose profile is expressed by the tuple $<\tau^P,\pi^P,C^P,S^P,I^P,O^P,P^P,E^P>$ if there is a pair wise match between the description of R and the description of P. In other words τ^R should match $\tau^{P,}$ π^R should match π^P and so on. A match of a pair of τs is recognized when the following formula holds:

$$\tau^R = \tau^P \vee \tau^R \subset \tau^P \vee \tau^P \subset \tau^R \vee \tau^R \cap \tau^P$$

The products π are matched using a similar rule by substituting π for τ.

The matching of inputs and outputs is based on the extension of the matching rule to set of requirements. Specifically, outputs are matched as described below; while the rule for the matching of inputs is obtained by substituting O^P and O^R with I^P and I^R.

$$\forall x^R \in O^R\ \exists x^P \in O^P\ (x^R = x^P \vee x^R \subset x^P \vee x^P \subset x^R \vee x^R \cap x^P)$$

The four disjuncts in the formula specify four degrees of matching between R and P. The first conjunct, $x^R=x^P$, specifies that x^R matches x^P exactly, this provides a guarantee to R that P will be able to answer its queries. The second conjunct, $x^R \subset x^P$ specifies that if P is more general than R, which in turns assumes that P will provide the service required by R, but with lower guarantees. The third conjunct, $x^P \subset x^R$ specifies that P is more specific than R and therefore it may not satisfy some of R's needs. Finally, $x^R \cap x^P$ specifies that the two services intersect so P may be able to satisfy some of the needs of R, but not all. If none of the four conjuncts is satisfied, the matching fails.

With the rules listed above the matching engine is able to verify whether the requested service and the provided service are of the same type, and whether they are functionally equivalent, ie whether they describe a two functions with similar domain and range.

6.1 OWL-S in UDDI

The adoption of OWL-S does not preclude the use of UDDI, rather there is a mapping from OWL-S to UDDI that allows the storage of service capabilities specified in OWL-S within UDDI [4]. Some elements of the OWL-S Profile, such as contact information, map directly to equivalent information in the description of businesses and services in UDDI. Other information, such as Inputs and Outputs does not have direct corresponding UDDI elements. In these cases the mapping is realized by defining a set of specialized TModels that associate the ontological information with the UDDI entities. Figure 13 illustrates the details of the OWL-S/UDDI mapping highlighting what parts of OWL-S are mapped directly in UDDI and what parts require TModels instead.

6.2 Representing registries

A fundamental problem of ESOA is to provide a representation of the organizations' registries in the Main Registry. Within SOA, UDDI provides an effective way to represent Web services. Furthermore, above we showed how such a representation can be enriched with semantic information. Since ESOA extends SOA to discover registries, it also requires a representation of the distinctive features of registries.

The problem of representing registries of Web services has two aspects. The first one is how do we represent the registry so that the other registries can contact it and interact with it; the second one is to provide a representation of the types of services that are stored in the registry.

The first problem is solved by viewing UDDI registries as Web services, and therefore there is virtually no difference between registering a web service or UDDI with the Main UDDI. The second problem is more difficult to solve. In its general form, this problem requires an abstraction from the descriptions of the services registered to the type of those services. Such an abstraction is very difficult to express, since it is not quite clear what information such an abstract representation should contain.

Rather than providing abstract descriptions of services, we try to take a different approach: we represent registries using the URIs of all the ontologies that are needed to represent the Web services in that registry. The intuition behind this decision is that the ontology represents the type of information that the registry contains. For example, referring back to our emergency operation scenario presented in Section 2, the EPA publishes its registries by specifying that they contain ontologies about environment and chemical compounds reflecting the capabilities provide. Other organizations such as police or hospitals would publish other ontologies reflecting different capabilities. The use of the URIs has two advantages: first URIs are unique, so two different ontologies are guaranteed to be associated with two different URIs; furthermore, to find URIs there is no need of any search mechanism beyond the keyword matching proposed by UDDI, therefore finding the desired registry is very quick.

7.Relation with similar approaches

ESOA is not a one of a kind, a number of solutions have been proposed to allow UDDI registries to interact with each other. Much of this work has tried to extend the UDDI standard replication scheme using lazy and eager replication schemata [12, 14, 15, 16], but there have been other proposals, most notably the UX [17, 18] and MWSDI [19], that do not require any replication. Since UX and MWSDI distribute the discovery process without any replication they are the closest to our work. In this section we review the most relevant proposals and try to highlight the contribution that we provide with this paper.

UX proposes a connection between UDDIs that is based on a P2P message schema propagation in which a registry queries its neighbors which in turn propagate the query to their neighbors until the requested service is found, or the propagation reaches its limits. Specifically, each request is bound by a *Time to Live* (TTL) which specifies how many times a query can be propagated.

The solution proposed by UX has a number of advantages: first, it does not require any specification of the type of information that is stored in a UDDI server, nor it requires any specification of a central UDDI registry; second, the network can expand and modify dynamically allowing new registries to join in and registries that are already present to disappear; third, the network of UDDIs is very resilient since there is no single point of failure, and the failure of any of the UDDIs does not prevent the overall community from continuing to work; finally, by providing some minimal structure and load balancing mechanisms within the network it is possible to increase the likelihood that the network works efficiently and that services will be found.

Despite its numerous advantages, the solution adopted by UX has drawbacks that make it problematic to deal with the generation of virtual organizations. First, the management of Emergency operations is intrinsically a centralized process, therefore it is natural to have a centralized IT infrastructure, with a Main UDDI as proposed for ESOA. Second, in a P2P network, the resilience of the network comes at a cost of speed and quality of results reported. Specifically, using UX, or extensions of it, it is difficult, if not impossible, to propagate the service request in the direction of the most likely registry to record the service. The request therefore may propagate in the network until it reaches a TTL without reaching the one registry that can report the service. Alternatively, the Main UDDI allows the request to be forwarded directly to the registries that most likely contain the service avoiding all the registries that would report a failure, reducing response time and network traffic, with minimal risk that the Main UDDI will fail since it will be outside of the problematic area. In general, UX and ESOA strike two different trade-offs, and it is impossible to determine that one or the other will be the best in every possible situation.

MWSDI is very close to our architecture, as in ESOA MWSDI uses a meta registry, a registry of registries, to collect information about the registries, and registries are advertised through the ontologies that they use to describe their services. While these two similarities are very striking, the two systems are very different in their architecture and in the assumptions that they make.

The goal of MWSDI is twofold, first it aims at providing a federation mechanism for UDDIs in which different registries provide different views on the services that they register along different dimensions such as geographical, technical, trust and so on. The second goal is to support heterogeneity of business registries. Some may be based on UDDI other on EBXML, or they may use different ontologies and data structures to represent services. The overarching objective of MWSDI is to provide a uniform way to provide an architecture and an inference engine that distributes the information and it is still able to retrieve services registered in other UDDIs.

Architecturally, MWSDI requires meta registry, named XTRO, provides both the ontology to describe the relation between the different registries, as well as the actual registry operations. XTRO as an ontology supports the representation of *Registries*, each registry is associated with one or more *Domain*, in which are described by a set of *Ontologies*. In turn Registries belong to one or more *Registry Federation* that also is associated with one or more Domain. Registries are abstractly characterized by peers in a JXTA peer-to-peer network in which there are different types of peers. Specifically MWSDI distinguishes four types of peers: *Gateway Peers* that act as entry points for registries to join MWSDI and to update the XTRO. *Operator Peers* that operate UDDI registries as well as XTRO. *Auxiliary Peers* that provide backup capabilities; and finally *Client Peers* that provide access to the MWSDI. To find a service, a client needs to send a query that specifies in which domain and directory federation to look for the service, and the set of ontologies that are required to describe the service. Essentially therefore the client should have a clear knowledge of the structure of the federations and of the domains in which to look for services.

The architecture proposed by ESOA greatly simplifies the architecture proposed by MWSDI. There is no need to define an explicit ontology and data structure for XTRO, rather ESOA exploits the standard data structure provided by UDDI. Furthermore there is no need to specify the different types of peers in the federation. ESOA assumes the existence of a Main Registry and of a number of peer registries that interact freely. In addition, the ESOA clients do not have any knowledge of the structure of the organizations and of the structure of the federation. Rather, the registry takes care of exposing services cross-organizational boundaries. Finally, although we did not analyze how different ontologies can be used in ESOA, there is no hidden assumption that every party knows all the ontologies (in the sense of OWL files) used in the federation. For example, a client may discover new ontologies because they are used to describe the services that the client just discovered; also, registries may be aware of the bridging axioms [20] between different ontologies and use them during the discovery process.

Part of the difference between the two initiatives stems from the fact that they have very different goals. The main goal of ESOA is to support the dynamic creation of a federation of registries to enable interoperation across organizational boundaries, while the main goal of MWSDI is to support the creation of market places that may have a complex internal structure and are supported by multiple registries. While the two goals overlap, they are not the same. For example, ESOA requires that the users do not need to be aware of the structure of the federation of registries and that clients do not need to be changed, while clients of MWSDI needs to be aware of the structure of the market place to form queries. To support this awareness, MWSDI requires a complex organization of clients and registries in terms of domains and gateways that is not required in ESOA where no structure is assumed on the provider's side, and only one coordinating registry is needed.

8.Implementation

In our prototype implementation of discovery within ESOA we aimed at verifying that the architecture that we envisioned indeed supports discovery of Web services across organizational boundaries. The second goal of the implementation has been to evaluate how much UDDI needs

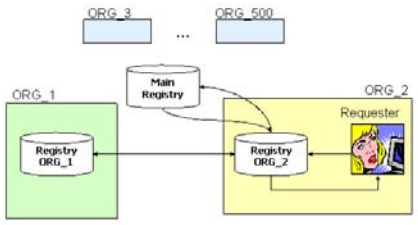


Figure 14: Experimental Setup

to be modified to support ESOA, and more importantly, how much a UDDI clients need to be modified. The latter is important since any change of UDDI clients would have a major effect on any IT infrastructure since there are many more clients than UDDI registries. Finally, we used our implementation to evaluate the performance and scalability of ESOA.

The organization's UDDI were implemented using the CMU OWL-S/UDDI [10] which is an extension of UDDI that implements the discovery mapping described in section 6.1. Specifically, the CMU OWL-S/UDDI is based on jUDDI [6] that is an open source UDDI v2 registry. Web services are published to UDDI following the standard API dictated by the UDDI specifications. Similarly for the standard set of inquiry APIs. The only difference is that the CMU OWL-S/UDDI uses an additional API, called *Capability Search* that invokes a matching engine based on the OWL-S discovery mapping.

In order to support their role in ESOA we had to extend the Organizations' UDDI registries in two directions. First, they had to become clients of other UDDIs. Since jUDDI does not implement any UDDI replication API, we extended each UDDI with a UDDI client so that they became able to query other UDDI registries. The second extension was to introduce a new data-structure in UDDI, that we called *key table* that allowed to record associations between internal keys and foreign keys, as described in section 5.3.

During discovery, a registry uses the client to contact the Main Registry and other UDDI registries to find the desired Web services. Upon finding a Web service, the registry saves a reference specifying the unique id of the Web service on the original registry, and the name and bindings of the registry where the Web service was found. If the requester needs additional information about the Web service, the registry uses its key table to find to which UDDI server to send the request for information.

While the implementation of the organizations' UDDI registries required some changes, the main UDDI did not require any change. Organizations' UDDI registries were published in the Main Registry as any other service and the ontologies that the used were encoded using TModels in the businesses category bags. In turn this implementation allowed us to use UDDI native search mechanisms since the matching of ontologies can be reduced to a string matching of URIs.

While the implementation of our ESOA prototype required some changes to UDDI registries, these extensions were limited to two changes in the Organizations' UDDIs. Crucially, no application or Web service needs to be changed to implement architectures that are consistent with ESOA, and the Main UDDI could be implemented using the plain UDDI registry.

9.Performance Evaluation

To evaluate the performance and the scalability of our implementation we performed a number of performance evaluation experiments in which we discovered new Web services using the ESOA protocol. The basic structure of the experimental setup is described in Figure 14. We used a client on an organization (ORG_2) to request services that has been published on another organization (ORG_1). In addition, a number of organizations that varies between two (just ORG_1 and ORG2) and 500 were registered with the Main Registry.

We also controlled for other two parameters. The first one is the number of services that are registered in ORG_1, which varied between 10 and 800. The second one is the complexity of the query that is measured by the number of inputs and outputs used. In our experiments this number varies between 1 and 10. All the experiments are run using 10 ontologies of 1000 concepts each, for a total 10,000 concepts.

While these numbers may seem quite arbitrary, the logic that we followed in selecting them is the following: the biggest Web services deployment we are aware of has been discussed in a white paper published by Systinet [11] that places the amount of services at about 700. By assuming a maximum of 800 Web services we include the case of 700 Web services. The justification for the maximum of 10,000 concepts is that the biggest ontology we are aware of is the Sumo ontology [9] that has about 8000 concepts. Following the same logics discussed above, we rounded at 10,000.

The ontologies and the service descriptions used in the experiments were automatically generated. To generate the ontologies we created concept names of the form *conceptX*, where X varies from 1 to 10,000. The concept names were then randomly linked through a subclass relation. The OWL-S advertisements and requests were generated by randomly selecting concepts for inputs and outputs from the randomly generated ontologies.

9.1 Discovery: SOA vs ESOA

ESOA requires an overhead with respect to SOA since it requires the discovery of UDDI registries in addition to the service discovery. The first evaluation that needs to be done is to estimate the size of such an overhead.

Figure 15 shows the average response time of discovery within the SOA architecture versus discovery under the ESOA condition with 2 and 500 organizations. The difference between the lines gives a visual estimate of the overhead of the ESOA conditions.

Our evaluation is intrinsically biased against ESOA since within ESOA we had registered 500 times more services than in the equivalent condition using SOA. For example, in the case of 10 services registered, ESOA would have registered 5000 services (10 for each 500 registries). If we had queried jUDDI with 5000 services its performance would have been much worse than what we report.

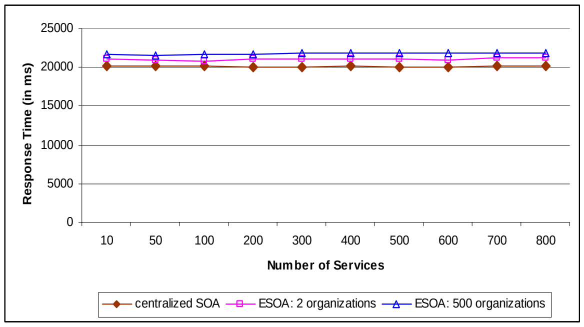


Figure 15: Query Time for SOA and ESOA

The data shows that the querying overhead of using ESOA is very limited, and that it does not produce any major impact in the overall performance of a system. Indeed, the overhead of ESOA in the case of 500 organizations is 1,639.524 ms, corresponding to a time increase of just 8.1% with respect to SOA. Furthermore, changing from 2 to 500 organizations produced an overhead of only 709 ms corresponding to an increase of just 3.4%.

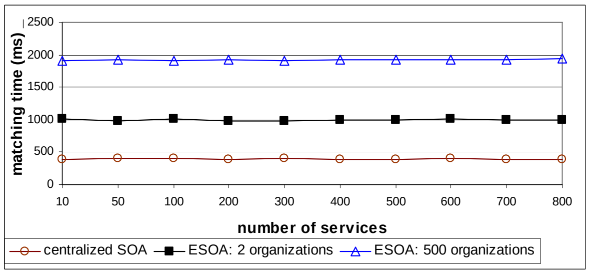


Figure 16: SOA vs ESOA excluding ontology loading

9.2 Factoring in Ontology Loading

Ontology loading is an expensive operation because it requires an expensive HTTP Get operation to get the ontology file from a Web Server, and a reconciliation of the knowledge base of the application that performs the loading. In our experiments ontology loading took on average 2 seconds per ontology, for a total of 20 seconds to load 10 ontologies.

Since ontology loading is more expensive than any other operation performed in our experiments it affects our experimental data. If we exclude ontology loading from our data the total matching time is shown in Figure 16. The matching time required by ESOA with two organizations takes on average 602.6 ms, which is about 1.5 times bigger than the centralized SOA (394.9 ms). The case of 500 organizations requires about four times as much as the centralized SOA for a total time of 1584.7ms.

In summary, although discovery ESOA requires more time than SOA, the total time does not increase dramatically. Furthermore, it scales with the increase of organizations, extending from 2 to 500 organizations takes only twice the time.

9.3 Matching time vs. query complexity

Finally, we evaluated how the discovery time varies with the complexity of the query and with the number of services that are registered with the registry of ORG_1. Intuitively it is expected that the discovery time increases with the number of services registered, and with the complexity of the query.

Figure 17 shows the total amount of time that is required to discover a repository and match the request. The time in ms is reported in the Y, while the X axis reports the amount of services that were registered with the UDDI repository that had the correct service. The different lines represent the complexity of the query measured in terms of inputs and outputs that needed to be matched. The simplest query has only 1 input and 1 output, while the more complex have 10 inputs and 10 outputs.

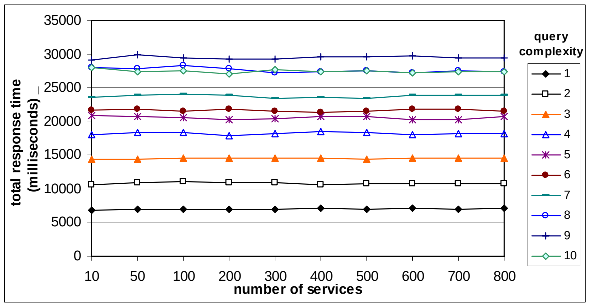


Figure 17: Matching time vs Query complexity

The result of this experiment show that the time of matching does not increase with the number of services, this is a good news because it shows that the matching process scales very well with respect to the services registered. But it also shows that the time of match increases with the complexity of the query. This is an artifact of the ontology problem discussed in section 6.2 since the more complex the queries that more ontologies need to be loaded by the client.

10.Conclusions

This paper presented an exploration into a way to extend SOA to cross-organizational boundaries. The solution adopted supports the discovery of Web services across organizational boundaries, allowing as a consequence the creation of virtual organizations.

In section 2.1, we put forward requirements that our infrastructure needs to satisfy to support the discovery across in virtual organizations organization boundaries. The organizational requirement is satisfied since the only operation required to enter and exit the virtual organization is to add, or remove, an entry in the Main UDDI registry. The implementation requirement is also satisfied since we require changes only to registries, while clients are not affected. The evaluation above proves that the performance requirement is also satisfied since ESOA scales very well with the number of parties and services in the virtual organization;

finally, the semantic matching supports the satisfaction of the last requirement since the member of one organization need to know only their own goals to find services in other organizations.

Furthermore, the solution adopted is respectful of the organizations IT infrastructure in the sense that it does not require any of these organizations to change their deployed Web services, nor to modify their applications to take advantage of new information that is available through the participation in a virtual organization. Crucially, the creation of a virtual organization is totally transparent to the organizations and Web services of each organization. The only changes that are required are in the architecture of the organizations' UDDI registries that need to be able to query other registries to gather the information that they need and they have to be able to maintain information about other Web services.

Future research will address problems of security that have not been addressed here and more general policy specifications [13] that allow different registry operators to specify which services they want to expose and to whom so that their whole infrastructure is opened in a selective way rather than allowing everybody to access every service.

From the architectural point of view, the search mechanism proposed in this paper aims at finding the service that addresses the problems of the requester. In a highly distributed environment such as the one proposed here, no single organization may address the whole problem rather the answer may be achieved by composing services provided by different organizations. The problem in this case is to provide a query distribution algorithm that will spread the query across multiple registries and collect references to services that, if coordinated, may solve the requester's problem. Such an algorithm has been provided for single registries in [2] but it is still a research issue on how it can be distributed across multiple registries.

References

[1] Akkiraju et al: A Method For Semantically Enhancing the Service Discovery Capabilities of UDDI, Workshop on Information Integration on the Web IJCAI 2003.
[2] Benatallah, B., Hacid, M.H., Rey, C. & Toumani, F.,"Request Rewriting-based Web Service Discovery*",Proceeding of the Second International Semantic Web Conference*,Sanibel Island, Fl, USA, 2003.
[3] Horrocks et al: SWRL: A Semantic Web Rule Language Combining OWL and RuleML, http://www.w3.org/Submission/SWRL/.
[4] Paolucci et al: Importing the Semantic Web in UDDI. In Proceedings of Web Services, E-business and Semantic Web Workshop, 2002.
[5] Paolucci et al; Semantic Matching of Web Services Capabilities. In Proceedings of the 1st International Semantic Web Conference (ISWC2002).
[6] jUDDI: http://ws.apache.org/juddi/.
[7] W3C: Web Ontology Language. http://www.w3.org/2001/sw/WebOnt/.
[8] North American Industry Classification System (NAICS), http://www.census.gov/epcd/www/naics.html.
[9] Niles, I., & Pease, A. 2001. Towards a Standard Upper Ontology. In Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001), Chris Welty and Barry Smith, eds, Ogunquit, Maine, October 17-19, 2001.
[10] Srinivasan, N., Paolucci, M., & Sycara, K. (2004). An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. SWSWPC 2004: 96-110.
[11] Systinet Case studies: Global Investment Bank Deploys SOA, http://www.systinet.com.
[12] UDDI Committee Specification: *UDDI Version 2.04 API Specification,* 19 July 2002, http://uddi.org/pubs/ProgrammersAPI_v2.htm.
[13] Rompothong, P. and Senivongse, T: *A query federation of UDDI registries*. In Proceedings of the 1st international Symposium on information and Communication Technologies, Dublin, Ireland, September 24 - 26, 2003.
[14] Sun, C., Lin, Y., & Kemme, B. (2004). Comparison of UDDI registry replication strategies. *Proceedings of IEEE International Conference on Web Services (ICWS2004).*
[15] Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., & Alonso, G. (2002). Improving the scalability of fault-tolerant database clusters. *IEEE 22nd Int. Conf. on Distributed Computing Systems, (ICDCS'02),* pp. 477-484. Vienna, Austria.
[16] Surgihalli, M. & Vidyasankar, K. (2005). A Lazy replication scheme for loosely synchronized UDDI Registries. *Proceeding of 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005),* Phoenix, USA.

[17] Zhou, C., Chia, L., Silverajan, B. & Lee, B. (2003). UX- An architecture providing QoS-aware and federated support for UDDI. Proceeding of the first International Conference on Web Services (ICWS03).

[18] Zhou, C., Chia, L., & Lee, B. (2004). QoS-aware and federated enhancement for UDDI. International Journal of Web Service Research. 1(2): 58-85.

[19] Oundhakar, S., Verma, K., Sivashanugam, K., Sheth, A., & Miller, J. (2005). Discovery of web services in a multi-ontology and federated registry environment. *International Journal of Web Services Research*, 2 (3), July-September, 2005, pp.1-32.

[20] Dou, D., McDermott, D., & Qi, P. (2003). Ontology translation on the semantic Web. *Proceedings of International Conference on Ontologies, Databases and Applications of Semantics, (ODBASE2003)*. LNCS 2888, pp. 952-969.

[21] C. Sun, Y. Lin, B. Kemme (2004). Comparison of UDDI Registry Replication Strategies, *Proc. of the IEEE Int. Conference on Web Services, San Diego, California*, July 2004.