SEA: a Semantic Web Services Context-aware Execution Agent

António Lopes, Luís Botelho

"We, the Body and the Mind" Research Lab – ADETTI/ISCTE
Av. Forças Armadas, Edifício ISCTE, 1600-082, Lisboa, Portugal
{antonio.lopes,luis.botelho}@we-b-mind.org

Abstract

This paper presents the on-going research on agent technology development for context-aware execution of semantic web services, more specifically, the development of SEA (Service Execution Agent), a semantic web services execution agent that uses context information to adapt the execution process to a specific situation, thus improving its effectiveness and providing a better service. The agent uses standards such as OWL-S service descriptions and WSDL grounding information. Also, an Agent Grounding definition has been proposed to enable the execution of semantic web services provided by agents.

Introduction

Standard initiatives such as OWL-S and WSDL [Martin, 2004] enable the automation of discovery, composition and execution of semantic web services, i.e. they create a Semantic Web, such that computer programs or agents can implement an open, reliable, large-scale interoperation of Web Services. This paper describes the on-going research, part of an MsC work, on agent technology development for context-aware execution of semantic web services.

We have decided to adopt the agent paradigm, creating the SEA (Service Execution Agent) agent, because we intend to integrate this work in open societies of agents, enabling these to execute semantic web services. [Paolucci, 2004] used the same approach in the Web Services infrastructure because of its capability to perform a range of coordination activities and *anonymising* between requesters and providers. The capacity for engaging in flexible interactions that were not previously defined by the agent developer makes this agent a good candidate for providing a value-added brokerage execution service. Furthermore, the use of context information helps improve the execution process, by adding valuable situation-aware information that will contribute to its effectiveness.

The major contributions of the present work to advance the state-of-the-art are:

1. The development of a broker agent capable of executing received OWL-S/WSDL service descriptions;
2. The inclusion of context-aware capabilities into semantic web services execution;
3. An extension to the OWL-S Grounding specification to include a formal representation of interactions with service provider agents;
4. And the execution of the complete OWL-S specification, including handling logical expressions in conditional constructs.

The rest of this paper is organized as follows: first section gives a brief overview of context and context-awareness; second section describes the created broker agent in detail; third section presents a concrete example of execution of a value-added service using SEA for context-aware execution of semantic web services; in the fourth section, we relate our approach with existing work; finally, in the fifth section, we conclude and present guidelines for future work.

Context and Context-awareness

Context-aware computing is a computing paradigm in which applications can discover and take advantage of contextual information. The definition that seems to be most widely accepted for "*context*" comes from [Dey, 1999]:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves"

We can enhance this definition of context by stating that the context of a certain entity is any information (provided by external sensors or other entities) that can be used to characterize the situation of that entity individually or when interacting with other entities. The same concept can be transferred to application-to-application interaction environments.

A general mechanism for context-aware computing is summarized in the following steps [Abowd, 1998]:

1. Collect information on the user's physical, informational or emotional state;
2. Analyze the information, either by treating it as an independent variable or by combining it with other information collected in the past or present;

3. Perform some action based on the analysis;
4. And repeat from step 1, with some adaptation based on previous iterations.

SEA and context-awareness

SEA uses a similar approach to [Abowd, 1998] to enhance its activity, by adapting it to the specific situation that the agent and its client are involved in, at the time of the execution process. The use of context information is not only valuable for execution processes. [Broens, 2004] describes the use of context information for enrichment of the semantic web services discovery process.

SEA interacts with a generic context system [Costa, 2005] in order to obtain context information, subscribe desired context events and to provide relevant context information. Other agents, web services and sensors (both software and hardware) in the environment will interact with the context system as well, by providing relevant context information related to their own activities, which may be useful to other entities in the environment.

Acquisition of context information is made through a specific interface of the context framework by querying it in a pre-determined query language.

Context events are event listeners that monitor certain changes in context. Whenever context information changes, the system notifies the entities that subscribed the corresponding class of events. This is useful for SEA to be aware of availability of certain entities in the environment on which it operates.

Throughout the execution process, SEA provides and acquires context information from and to this context system. For example, SEA provides relevant information such as the queue of its service execution requests and the average time of service execution. This will allow other entities in the environment to determine the service execution agent with the smallest work-load, and hence that can provide a faster execution service. During the execution of a compound service, SEA invokes atomic services from specific service providers (both web services, and service provider agents). SEA also provides valuable information regarding these service providers' availability and average execution time. Other entities can use this information to rate service providers or to simply determine the best service provider to use in a specific situation. Furthermore, SEA uses its own context information (as well as information from other sources and entities in the environment) to adapt the execution process to a specific situation. For instance, when selecting among several providers of some desirable atomic service, SEA will choose the one with better availability (lesser history of down time) and lower average execution time.

In situations such as the one where service providers are unavailable, it is faster to obtain the context information from the context system (as long as service providers can also provide their own availability context information) than by simply trying to use the services and discover that they are unavailable (because of the time lost waiting for connection time-outs to occur). After obtaining this relevant information, SEA can then contact other service-oriented agents (such as service discovery and composition agents) for requesting the re-discovering of service providers and/or re-planning of composed services. This situation-aware approach using context information on-the-fly helps SEA to provide a value-added execution service.

SEA: the Broker Agent

SEA is a broker agent that provides context-aware execution of semantic web services. It was implemented using Java and component-based software as well as other tools that were extended to incorporate new functionalities into the service execution environment. These tools are the JADE agent platform [Bellifemine, 1999] and the OWL-S API [Sirin, 2004].

The agent was designed and developed based on the interactions described in the previous section and the internal architecture was clearly designed to enable the agent to engage in the these required interactions: receive and reply to requests from client agents; acquire\provide relevant context information; request for re-discovering and re-planning of services; and execute remote semantic web services.

This section of the paper is divided into four sub sections. The first sub section describes the internal architecture of the agent, explaining in detail the functioning of the internal components. The second sub section gives a brief overview of the OWL-S and WSDL description languages and a description of the OWL-S/WSDL execution process. The third sub section describes the extension made to the OWL-S Grounding specification and to the OWL-S API to support the execution of services provided by agents. Finally, in the fourth sub section, the external interface of the agent is described.

SEA's Internal Architecture

The developed agent is composed of three components: the Agent Interaction Component (AIC), the Engine Component (EC) and the Service Execution Component (SEC).Figure 1 illustrates the internal architecture of the agent and the interactions that occur between the components.

The AIC was developed as an extension of the JADE platform [Bellifemine, 1999] and its goal is to provide an interaction framework to FIPA-compliant agents, such as SEA's clients (requesting the execution of specified services – Figure 1, step 1) and service discovery and composition agents (when SEA is requesting the re-discovering and re-planning of specific services – Figure 1, steps *). This component extends the JADE platform to provide extra features regarding language processing, behavior execution, database information retrieval and components' communication.

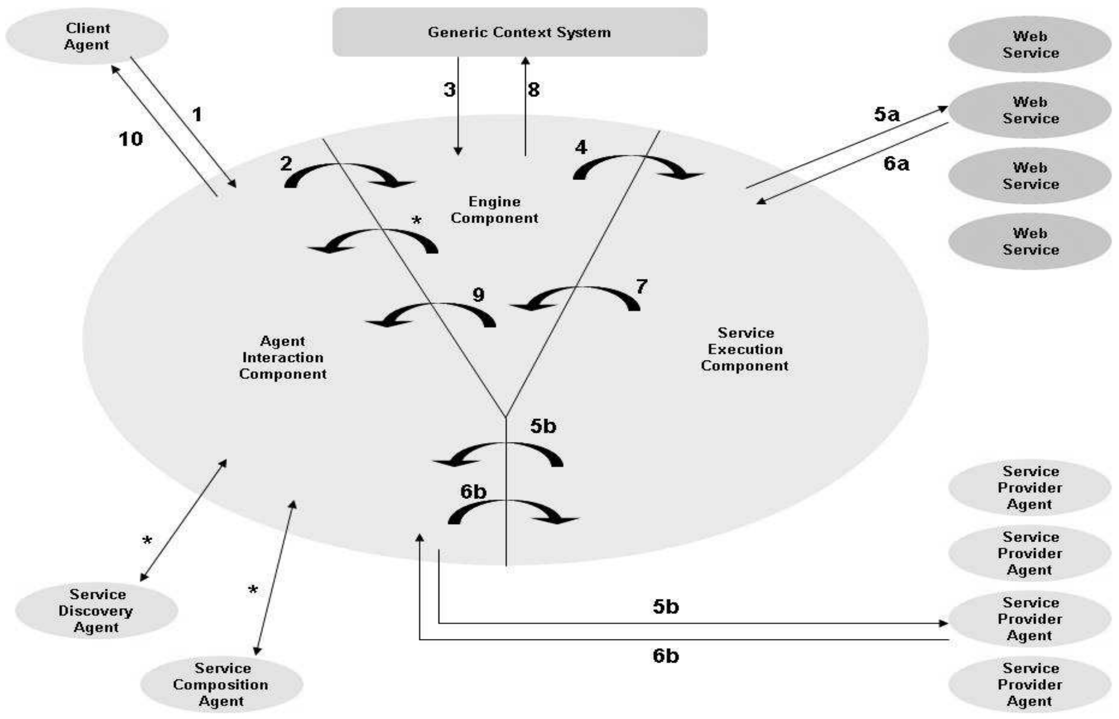

Figure 1 – Interactions and Internal Architecture of SEA

Among other things, the AIC is responsible for receiving messages, parsing them and processing them into a suitable format for the EC to use it (Figure 1, step 2). The reverse process is also the responsibility of AIC – receiving data from the EC and processing it into the agents' suitable format to be sent as messages (Figure 1, step 9).

The EC is the main component of SEA as it controls the agent's overall activity. It is responsible for pre-processing service execution requests, interacting with the context system and deciding when to interact with other agents (such as service discovery and composition agents).

When the EC receives an OWL-S service execution request (Figure 1, step 2), it acquires suitable context information (regarding potential service providers and other relevant information, such as client location – Figure 1, step 3) and plans the execution process.

If the service providers of a certain atomic service (invoked in the received composed service) are not available, SEA interacts with a service discovery agent (through the AIC – Figure 1, steps *) to discover available providers for the atomic services that are part of the OWL-S compound service. If the service discovery agent can not find adequate service providers, the EC can interact with a service composition agent (again through the AIC – Figure 1, steps *) asking it to create an OWL-S compound service that produces the same effects as the original service.

After having a service ready for execution, with suitable context information, the EC sends it to the SEC (Figure 1, step 4), for execution. Throughout the execution process, the EC is also responsible for providing context information to the context system, whether it is its own information (such as service execution requests' queue, average time of execution), or other entities' relevant context information (such as availability of providers and average execution time of services).

The SEC was developed as an extension of the OWL-S API [Sirin, 2004] and its goal is to execute semantic web services (Figure 1, steps 5a and 6a) described using OWL-S service description and WSDL grounding information. The extension of the OWL-S API allows for the evaluation of logical expressions in conditioned constructs, such as the *If-then-Else* and *While* constructs, and in the service's pre-conditions and effects. OWL-S API was also extended in order to support the execution of services that are grounded on service provider agents (Figure 1, steps 5b, 6b). This extension is called *AgentGrounding* and it is explained in detail in the third sub section.

When the SEC receives a service execution request from the EC, it executes it according to the description of the service's process model. This generic execution process is described in the next sub section.

During the execution process, SEC collects relevant context information (such as providers' availability, quality of service and execution times).

After execution of the specified service and generation of its results, the SEC sends them to the EC (Figure 1, step 7) for further analysis and post-processing, which includes sending gathered context information to the context system (Figure 1, step 8) and sending the results to the client agent (through the AIC – Figure 1, steps 9, 10).

Executing OWL-S/WSDL Services

OWL-S is an OWL-based (Web Ontology Language) ontology used to describe semantic web services. OWL-S Services are described in three parts: a *Profile* (which tells "what the service does"); a *Process Model* (which tells "how the service works"); and a *Grounding* (which tells "how to access the service"). The Profile and Process Model are considered to be *abstract* specifications, in the sense that they do not specify the details of particular message formats, protocols, and network addresses by which a Web service is instantiated. This role of providing more concrete details belongs to the grounding part. WSDL (Web Service Description Language) provides a well-developed means of specifying these kinds of details.

For the execution process, the most relevant parts of an OWL-S service description are the *Process Model* and the *Grounding*. The *Profile* part is more relevant for discovery, matchmaking and composition processes, hence no further details will be provided in this paper.

Process Model. The *Process Model* (or *Service Model*) describes the steps that should be done for a successful execution of the service. These steps represent two different views of the process: first, a process produces a data transformation of the set of given inputs into the set of produced outputs; second, a process produces a transition in the world from one state to another. This transition is described by the preconditions and effects of the process [OWL, 2003].

The *Process Model* identifies three types of processes: *atomic*, *simple*, and *composite*. *Atomic* processes are directly evocable (by passing them the appropriate messages). Atomic processes have no sub-processes, and can be executed in a single step, from the perspective of the service requester. *Simple* processes are not evocable and are not associated with a grounding description, but, like atomic processes, they *are* conceived of as having single-step executions. *Composite* processes are decomposable into other (non-composite or composite) processes. These represent several-steps executions, which can be described using different control constructs, such as sequence (representing a sequence of steps) or If-Then-Else (representing conditioned steps).

Grounding. The *Grounding* specifies the details of how to access the service. These details mainly include protocol and message formats, serialization, transport, and addresses of the service provider. The central function of an OWL-S Grounding is to show how the abstract inputs and outputs of an atomic process are to be concretely realized as messages, which carry those inputs and outputs in some specific format. The *Grounding* can be extended to represent specific communication capabilities, protocols or messages. WSDL and *AgentGrounding* (described in the next sub-section) are two possible extensions.

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate [Christensen, 2001].

Execution process. The execution of an OWL-S service is made in the following steps:

1. Validation of the service's pre-conditions. The execution process continues only if all pre-conditions are true;
2. If the service to be executed is a compound service, it must be decomposed into individual atomic services, which are executed by evoking their service providers. Invocation is done through the description of the service providers (and their interfaces) contained in the grounding section of the service description (WSDL or *AgentGrounding*);
3. After execution, validation of the service's effects is made by comparing them with the actual service execution results (if the service was executed as expected in the effects, then proceed);
4. Collect the results (if any – the service may be only a "*change-the-world*" kind of service).

OWL-S Grounding Extension: AgentGrounding

WSDL describes the access to a network service, more specifically, web services provided by network service providers. The WSDL representation lacks, however, the necessary expressivity for representing interactions with service provider agents. To overcome this limitation, we decided to create an extension of the OWL-S *Grounding* specification, named *AgentGrounding*. This extension is the result of an analysis of the necessary requirements for interacting with agents when evoking the execution of atomic services.

The *AgentGrounding* definition includes the following elements:

- *agentName* – the name of the service provider agent
- *agentAddress* – the address of the service provider
- *serviceName* – name of the service to be evoked
- *serviceType* – type of the service to be evoked (action, referential expression, proposition)
- *hasArgumentParameter* – used to represent arguments of the service
    - *argumentType* – type of the argument (string, integer)
    - *owlsParameter* – reference to OWL-S service defined parameters (inputs)
    - *paramIndex* – order on which the argument appears in the service invocation
- *serviceOutput* – used to represent the outputs of the service
    - argumentType
    - owlsParameter

- *protocol* – communication protocol to be used when invoking the service
- *serviceOntology* – ontology which describes the terms to be used in the invocation of the service
- *agentCommunicationLanguage* – agent communication language to be used when communicating with the agent (FIPA-ACL, KQML)
- *contentLanguage* – content language to be used when communicating with the agent (FIPA-SL, KIF)

Figure 2 is an example of an OWL-S service *Grounding* description using the proposed *AgentGrounding* extension. This description illustrates a service, provided by a FIPA compliant agent, of finding books (within several different sources) with a given input title. This example can be used in the value-added service described in the next section, where the context-aware execution process is described.

```
<!-- Grounding description -->
<agentGrounding:AgentGrounding
    rdf:ID="BookFinderGrounding">
  <service:supportedBy
    rdf:resource="#BookFinderService"/>
  <grounding:hasAtomicProcessGrounding
    rdf:resource="#BookFinderProcessGrounding"/>
</agentGrounding:AgentGrounding>

<agentGrounding:AgentAtomicProcessGrounding
  rdf:ID="BookFinderProcessGrounding">
  <grounding:owlsProcess
    rdf:resource="#BookFinderProcess"/>
  <agentGrounding:agentName>
    bookeeper@services.we-b-mind.org
  </agentGrounding:agentName>
  <agentGrounding:agentAddress>
    http://localhost:9669/acc
  </agentGrounding:agentAddress>
  <!-- Service Identification -->
  <agentGrounding:serviceName>
    find-book
  </agentGrounding:serviceName>
  <!-- Service Arguments -->
  <agentGrounding:hasArgumentParameter>
    <agentGrounding:ArgumentParameter
      rdf:ID="input-string">
      <agentGrounding:argumentType>
        java.lang.String
      </agentGrounding:argumentType>
      <agentGrounding:owlsParameter
        rdf:resource="#InputString"/>
      <agentGrounding:paramIndex>
        1</agentGrounding:paramIndex>
    </agentGrounding:ArgumentParameter>
  </agentGrounding:hasArgumentParameter>
  <agentGrounding:serviceOutput>
    <agentGrounding:ArgumentVariable
      rdf:ID="book-info">
      <agentGrounding:argumentType>
        java.lang.String
      </agentGrounding:argumentType>
      <agentGrounding:owlsParameter
        rdf:resource="#BookInfo"/>
    </agentGrounding:ArgumentVariable>
  </agentGrounding:serviceOutput>
  <!-- Other information -->
  <agentGrounding:serviceType>
    action</agentGrounding:serviceType>
  <agentGrounding:protocol>
    fipa-request</agentGrounding:protocol>
  <agentGrounding:agentCommunicationLanguage>
    fipa-acl
  </agentGrounding:agentCommunicationLanguage>
  <agentGrounding:contentLanguage>
    fipa-sl</agentGrounding:contentLanguage>
  <agentGrounding:serviceOntology>
   book-finder-ontology
  </agentGrounding:serviceOntology>
</agentGrounding:AgentAtomicProcessGrounding>
```

Figure 2 – Example of *AgentGrounding* description

The extension made to the OWL-S API, allows the execution of groundings such as the one described in Figure 2 by invoking the specified service. This invocation is made by sending a message directly to the agent providing the service. All the information that is needed for sending the message is included in the *AgentGrounding* description.

The example depicted in Figure 2 describes a service named *BookFinderService*, which is grounded to an action *find-book* that accepts as input a single string named *input-string*. This *input-string* argument is linked to the OWL-S service input parameter *InputString*. The action returns as output, also a string, named *book-info*, which is linked to the OWL-S service output parameter *BookInfo*. Other information that can be extracted from this grounding is the protocol (*fipa-request*), the agent communication language (*fipa-acl*), the ontology (*book-finder-ontology*) and the content language (*fipa-sl*) to be used in the invocation message.

SEA can use this information to send the FIPA message that is described in Figure 3.

```
(REQUEST
  :sender
    (agent-identifier
    :name sea@services.we-b-mind.org)
  :receiver
  (set
    (agent-identifier
    :name bookeeper@services.we-b-mind.org
    :addresses
      (sequence http://localhost:9669/acc)))
  :content
    "((action
      (agent-identifier
        :name bookeeper@services.we-b-mind.org
        :addresses
          (set http://localhost:9669/acc))
      (find-book
        :input-string \"Da Vinci Code\")))"
  :language fipa-sl
  :ontology book-finder-ontology
  :protocol fipa-request
  :conversation-id SEA-AGR-CID-1117800292257)
```

Figure 3 – Message generated from the example in Figure 2

The information extracted from the *AgentGrounding* example in Figure 2 is enough for the agent to be able to create the message. However, the agent must also add the OWL-S Service Input information to the generated grounding message. In this case, the client agent requested the execution of the service with the input *"Da Vinci Code"*.

The *AgentGrounding* specification allows the representation of several instances of messages that can be sent to FIPA compliant agents, including the use of different message performatives, agent communication languages and content languages. However, this is a work in progress and some possibilities are not yet covered.

SEA's Interface

When requesting the execution of a specified service, client agents interact with SEA through the FIPA-request interaction protocol [FIPA, 2002]. This protocol states that when the receiver agent receives an action request, it can either agree or refuse to perform the action. It should then notify the other agent of its decision through the corresponding communicative act (FIPA-agree or FIPA-refuse). SEA performs this decision process through a service execution's request evaluation algorithm that involves acquiring adequate context information. SEA will only agree to perform a specific execution if it is possible to execute it, according to currently available context information. For example, if necessary service providers (for the execution of the service) are not available and the time that takes to find alternatives is longer than the timeframe the client agent expects to obtain a reply to the execution request, then SEA refuses to perform it. On the other hand, if SEA is able to perform the execution request (because service providers are immediately available), but not in the time frame requested by the client agent (again, according to available context information) it also refuses to do so. SEA can also refuse to perform execution requests if its work-load is already too high (if its requests queue is bigger than a certain amount).

The FIPA-request also states that after successful execution of the requested action, the executer agent should return the corresponding results through a FIPA-inform message. After executing a service, SEA can send one of two different FIPA-inform messages: one sending the results obtained from the execution of the service; other sending just a notification that the service was successfully executed (when no results are produced by the execution).

Context-aware execution process example

We have stated before that context information and context-awareness can enhance the semantic web service execution process. In this section, we present an example that illustrates such a relation within the execution process carried out by SEA. This example is based on the existence of commercial areas, such as shopping malls, where this kind of value-added services can be provided.

Imagine Sara, a thoughtful young girl that easily forgets her friends' birthdays. Today is one of those days, where she forgot the birthday of her friend Hugo. In 15 minutes she is supposed to meet him and some other friends for his birthday party dinner and she has no gift for him. Knowing that Hugo loves novels, Sara decides to buy him the *"Da Vinci Code"* book. However, she does not know where is the best place to buy the book, taking in account the fact that she is late and that she wants to pay the lowest price possible.

When entering the shopping mall, Sara has access to a range of several different services by using the personal agent situated in her handheld device, including searching for books. She requests the execution of a service that will allow her to know which is the store with the best price for the book entitled "Da Vinci Code". However, she wants this service to reply in less than 1 minute, so that she can have time to buy the book, catch a taxi to the place where the dinner party will take place and still arrive on time.

Sara's personal agent obtains a service description from a service composition agent. The obtained service description is a composed service with the following steps (represented by atomic services):

- Find book's ISBN information from the title "Da Vinci Code";
- Use book's ISBN information to obtain the price from several different sources;
- Compare the obtained prices and determine which one is cheaper;

When SEA receives the service description, it acquires context information related to the client (by retrieving the location from a GPS sensor in Sara's handheld device) and the available service providers for the service. The service providers' context information includes the average execution time (i.e. the time they take, in average, to reply to a service invocation), their availability (retrieved from ping-like sensors), their location (if the service providers represent book stores, then their location is the book store location) and their work-load (current requests queue). After acquiring such information, SEA performs a match between client's location and providers' location and determines whether these providers are adequate to be used in the search. Providers that are close to Sara's position are preferred, since this is a time and location-constraint situation. Also, the availability of the service providers is an important factor. When some of the providers are not available at execution time, SEA must adapt and find alternatives.

After discovering new providers (by interacting with a service discovery agent) and acquiring relevant context information, SEA analyses if the time it will take for the execution of the composed service exceeds the deadline given by the client. By analyzing average execution times and work-load from providers, SEA determines it will take just a few seconds to retrieve the information Sara needs.

At this time, SEA informs Sara's personal agent that it agrees to perform the action, by accepting the time-constraint given by her. SEA then proceeds with the execution.

After obtaining the results, SEA discovers it has obtained several stores with the same price, being this the cheapest. However, SEA returns only the name and information of the store that is located nearest to Sara. In the store information, SEA could have included an image of a map for the trajectory that Sara would have to make from her current position to the store. However, context information (Sara's handheld is a cell-phone with a very small display) allowed SEA to adapt the store information to the device and instead of sending the map image, it sends a set of textual instructions that represent the itinerary from Sara's current position to the store.

In this example, the use of context information for determining who the best providers are, allowed SEA to provide a faster and more useful service to a user with time, device and location constraints.

Related Work

The need to add semantic information to web-accessible services has created a growing research activity over the last years in this area. Regarding semantic web services, two major initiatives rise above the others, mainly because of their wide acceptance by the research community: WSMO [Roman, 2004] and OWL-S. A comparison between the two service description languages, [Lara, 2004] states that WSMO is more suitable for specific domains related to e-commerce and e-business, rather than for generic use. OWL-S was designed to be generic and to cover a wide range of different domains. We decided to use OWL-S as the Service Description Language given its capability to represent several different and complex domains.

[Paolucci, 2004] proposes a broker-agent approach for the discovery and mediation of semantic web services in a multi-agent environment. However this approach does not take into account the use of context information as a way of improving the services provided in dynamic multi-agent environments, such as the ones operating on pure peer-to-peer networks.

Other semantic web service execution approaches, such as WSMX [Fensel, 2002] [Domingue, 2004] are available but all rely on WSMO. However, since OWL-S was the chosen service description language to be used in this research, it is important to analyze the existing developed technology related to this standard in particular.

Two main software tools referred in the OWL-S community as the most promising ones, regarding OWL-S services interpretation and execution: OWL-S VM [Paolucci and Srinivasan, 2004] and OWL-S API. However, OWL-S VM doesn't have a public release yet.

OWL-S API is a recently developed Java API that supports the majority of the OWL-S specifications. For being the only OWL-S tool publicly available, we have chosen to use and extend the OWL-S API. Currently, it only supports control constructs that depend on formal representation of logical expressions in SWRL [Horrocks, 2004].

Conclusions and Future Work

We have presented a framework to enable the execution of semantic web services using a context-aware broker agent.

As said in the introduction this is part of on-going MsC research. We have not yet reached the end of the research as some of the goals initially proposed, were not yet achieved. This includes the execution of the complete OWL-S specification, including handling logical expressions in conditional constructs. So far, the only support for conditional constructs is provided by the SWRL[Horrocks, 2004] extension of the OWL-S API [Sirin, 2004]. We intend to analyze which is the suitable language to represent such elements in OWL-S service descriptions and implement the necessary extension in the OWL-S API.

We will also analyze the expressiveness of the *AgentGrounding* representation, to make sure that it is abstract enough to represent any instance of messages that can be sent to service provider agents.

The use of SEA in the CASCOM project [Helin, 2005] will allow us to determine its applicability and usability in different scenarios operating in highly dynamic environments.

Acknowledgments

This work has been supported in part by the European Commission under the project grant FP6-IST-511632-CASCOM. We would like to thank the reviewers for their useful comments.

References

Abowd, G. D., Dey, A., Orr, R. and Brotherton, J., 1998, *Context-awareness in wearable and ubiquitous computing*, Virtual Reality, 3:200–211.

Bellifemine, F., Poggi, A., Rimassa, G., 1999, *JADE – a FIPA-compliant agent framework*, CSELT internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, pp.97-108.

Broens, T., Pokraev, S., Sinderen, M.V., Koolwaaij, J., Costa, P.D., 2004, *Context-aware, Ontology-based Service Discovery*, Proceedings of the Second European Symposium of Ambient Intelligence.

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., 2001, *Web Services Description Language (WSDL) 1.1*. Available on-line at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

Costa, P., Botelho, L., 2005, *Generic Context Acquisition and Management Framework*, First European Young Researchers Workshop on Service Oriented Computing. Forthcoming.

Dey, A. K. and Abowd, G. D., 1999. *Towards a better understanding of context and context awareness*. GVU Technical Report GIT-GVU-99-22, College of Computing, Georgia Institute of Technology.

Domingue, J., Cabral, L., Hakimpour, F., Sell, D., and Motta, E. 2004, *IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services*, 3rd International Semantic Web Conference (ISWC2004), LNCS 3298.

Fensel, D. and Bussler, C., 2002, *The Web Service Modeling Framework - WSMF*, Electronic Commerce: Research and Applications, 1 (2002) 113-137.

FIPA Members, 2002, *Foundation for Intelligent Physical Agents website*, http://www.fipa.org/.

Helin, H., Klusch, M., Lopes, A., Fernández, A., Schumacher, M., Schuldt, H., Bergenti, F., Kinnunen, A., 2005, *Context-aware Business Application Service Co-ordination in Mobile Computing Environments*, Workshop on Ambient Intelligence – Agents for Ubiquitous Environments, Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems.

Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., 2004, *SWRL: A Semantic Web Rule Language combining OWL and RuleML*, W3C Member Submission, available on-line at http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/

Lara, R., Roman, D., Polleres, A., Fensel, D., 2004, *A Conceptual Comparison of WSMO and OWL-S*, Proceedings of the European Conference on Web Services, ECOWS 2004, Erfurt, Germany.

Martin, D., Burstein, M., Lassila, O., Paolucci, M., Payne, T., McIlraith. S., 2004, *Describing Web Services using OWL-S and WSDL*, DARPA Markup Language Program.

OWL Services Coalition, 2003, *OWL-S: Semantic Markup for Web Services*, DARPA Markup Language Program.

Paolucci, M. and Srinivasan, N., 2004, *OWL-S Virtual Machine Project Page*, http://projects.semwebcentral.org/projects/owl-s-vm/.

Paolucci, M., Soudry, J., Srinivasan, N., Sycara, K., 2004, *A Broker for OWL-S Web Services*, First International Semantic Web Services Symposium, AAAI Spring Symposium Series.

Roman, D., Keller, U., Lausen, H., 2004, *Web Service Modeling Ontology (WSMO)* – version 1.2, available at http://www.wsmo.org/TR/d2/v1.2/.

Sirin, E., 2004, *OWL-S API project website*, http://www.mindswap.org/2004/owl-s/api/.