# Executing Semantic Web Services with a Context-Aware Service Execution Agent

António Luís Lopes, Luís Miguel Botelho

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

**Abstract.** The need to add semantic information to web-accessible services has created a growing research activity in this area. Standard initiatives such as OWL-S and WSDL 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 dynamic network of Web Services. This paper presents the research on agent technology development for context-aware execution of semantic web services, more specifically, the development of the Service Execution Agent (SEA). SEA uses context information to adapt the semantic web services execution process to a specific situation, thus improving its effectiveness and providing a faster and better service to its clients. Preliminary results show that context-awareness (e.g., the introduction of context information) in a service execution environment can speed up the execution process, in spite of the overhead that it is introduced by the agents' communication and processing of context information.

**Keywords:** Context-awareness, Semantic Web, Service Execution, Agents.

## 1 Introduction

Semantic Web Services are the driving technology of today's Internet as they can provide valuable information on-the-fly to users everywhere. Information-providing services, such as cinemas, hotels and restaurants information and a variety of e-commerce and business-to-business applications are implemented by web-accessible programs through databases, software sensors and even intelligent agents.

Research on Semantic Web standards, such as OWL-S [16] [19] and WSDL [3], opens the way for the creation of automatic processes for dealing with the discovery, composition and execution of web-based services. We have focused our research on the development of agent technology that allows the context-aware execution of semantic web services. We have decided to adopt the agent paradigm, creating SEA to facilitate the integration of this work in open agent societies [12], enabling these not only to execute semantic web services but also to seamlessly act as service providers in a large network of interoperating agents and web services.

In [21] the same approach was used in the Web Services infrastructure because of its capability to perform a range of coordination activities and mediation between requesters and providers. However, the broker-agent approach for the discovery and mediation of semantic web services in a multi-agent environment described in [21] does not take into account the use of context information. Thus we have decided to introduce context-awareness into the service execution process as a way of improving the services provided in dynamic multi-agent environments, such as the ones operating on pure peer-to-peer networks. Furthermore, the use of context information helps improve the execution process by adding valuable situation-aware information that will contribute to its effectiveness.

Being able to engage in complex interactions and to perform difficult tasks, agents are often seen as a vehicle to provide value-added services in open large-scale environments. However, the integration of agents as semantic web services providers is not easy due to the complex nature of agents' interactions. In order to overcome this limitation, we have decided to extend the OWL-S Grounding specification to enable the representation of services provided by intelligent agents. This extension is called the *AgentGrounding* and it is further detailed in [15].

We have also introduced the use of *Prolog* [4] for the formal representation of logical expressions in OWL-S control constructs. As far as we know, the only support for the formal representation of logical expressions in OWL-S (necessary for conditions, pre-conditions and effects) is done through the use of SWRL [13] and PDDL. Performance tests show that our *Prolog* approach improves the execution time of logical expressions in OWL-S services.

The remaining of this paper is organized as follows: section 2 gives a brief overview of related work; section 3 describes the use of context information and the introduction of context-aware capabilities in SEA; section 4 describes a motivating example which depicts a scenario where SEA is used; section 5 fully describes SEA by presenting its internal architecture, external interface, execution process and the implementation details; section 6 presents the performance tests and the overall evaluation of our research; finally, in section 7 we conclude the paper.

## 2 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 other, mainly because of their wide acceptance by the research community: WSMO [23] and OWL-S. A comparison between the two service description languages [14] concludes that the use of WSMO is more suitable for specific domains related to e-commerce and e-business, rather than for generic use, i.e., for different and more complex domains. OWL-S was designed to be generic and to cover a wide range of different domains but it lacked the formal specification to deal with logic expressions in the service description. We decided to use OWL-S as the Service Description Language due to its power in representing several different and complex domains. Other semantic web service execution approaches, such as WSMX [8] [9] 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 are referred in the OWL-S community as the most promising ones, regarding OWL-S services interpretation and execution: OWL-S VM [20] and OWL-S API [24]. However, at the time this research work has started, the OWL-S VM did not have a public release. OWL-S API is a developed Java API that supports the majority of the OWL-S specifications. For being the only OWL-S tool publicly available at the time, we have chosen to use and extend the OWL-S API.

In the interest of making the created technology interoperable with other systems that were already developed, we decided to ground its design and implementation on FIPA specifications, which are widely accepted agent standards. There are several existing FIPA-compliant systems that can be used: AAP [11], JADE [2], ZEUS [18], AgentAcademy [17] are just a few to be mentioned. We decided to use the JADE multi-agent platform because of its large community of users that constantly contribute to the improvement of the technology.

## 3 Service Execution and Context-awareness

Context-aware computing is a computing paradigm in which applications can discover and take advantage of contextual information. As described in [7] "context is any information that can be used to characterize the situation of an entity, being an entity 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.

Context-aware computing can be summarized as being a mechanism that collects physical or emotional state information on a entity; analyses that information, either by treating it as an independent variable or by combining it with other information collected in the past or present; performs some action based on the analysis; and repeats from the first step, with some adaptation based on previous iterations [1].

SEA uses a similar approach as the one described in [1] 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.

SEA interacts with a generic context system [5] 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.

Throughout the execution process, SEA provides and acquires context information from and to this context system. For example, SEA provides relevant context information about itself, such as its queue of 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 to the context system. Other entities can use this information (by contacting the context system) 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 service, SEA will choose the one with better availability (with less history of being offline) 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 context information about their own availability) than by simply trying to use the services and finding out 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 the re-planning of composed services. This situation-aware approach using context information on-the-fly helps SEA providing a value-added execution service.

## 4 Example: Search Books' Prices

In this section we present an example scenario in the domain of books' prices searching, in order to better prove the need for the existence of a broker agent for the execution of semantic web services.

Imagine a normal web user that wants to find a specific book (of which he doesn't recall the exact title) of a certain author, at the best price available. Probably, he would start by using a domain-specific search engine to find the intended item. After finding the exact book, he would then try to find websites that sell it. After doing an extensive search, he would finally find the web site that sells the book at the best price, but it only features the price in US dollars. The user is Portuguese and he would like to know the book's price in Euros, which leaves him with a new search for a currency converter service that would help him with this task. As we can see, this user would have to handle a lot of different web sites and specific searches to reach its objective: find the best price of a certain book.

The composition of atomic semantic web services into more complex value-added services would present an easy solution to this problem. The idea is to provide a unique compound service with the same features as the example described above, but the use of which would be a lot simpler, since it would be through the interaction with a service execution broker agent.

Fig. 1 shows the overall scenario description, by presenting all the participants and the interactions between them. We will assume the existence of a compound service called “book best-price search”. The dashed gray lines represent the interactions that are not subject of this paper.

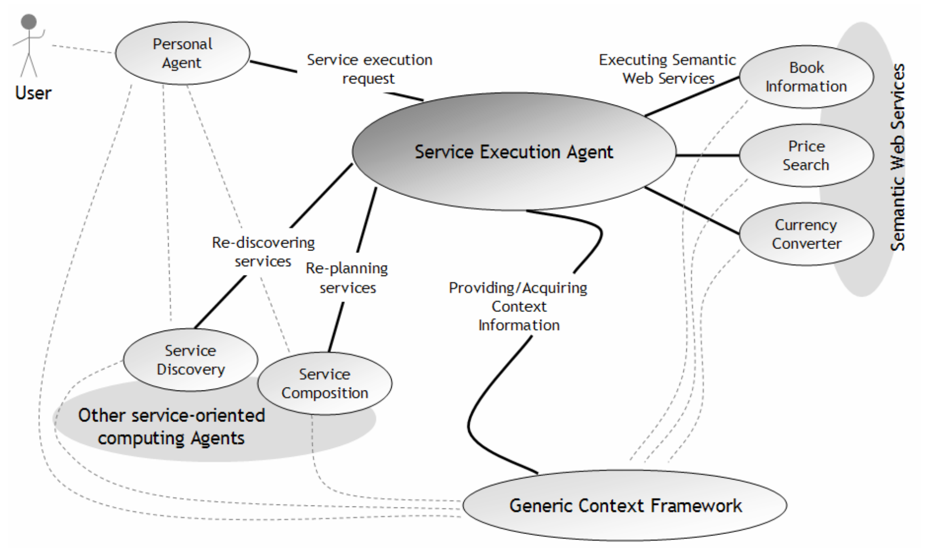

**Fig. 1.** Overall Scenario Description

The user of this service is represented in the figure as the client user and by his personal agent. The client user will provide the necessary information (author’s name, book’s title and expected currency) to his personal agent so that this can start the interaction with the remaining participants in order to obtain the desired result of the “book best-price search” service.

In order to request the execution of the “book best-price search” service, the personal agent needs to have the service’s OWL-S description. This could have been previously obtained from a composition planner agent or service discovery agent. This interaction and the composition of the compound service are not covered by this paper.

After sending the “book best-price search” service’s OWL-S description and the instantiation information (input parameters) provided by its user to the service execution agent, the personal agent will wait for the result of the service to then inform its user.

Semantic Web Services can be any web-based computer application, such as web services or intelligent agents, as long as they are described using a formal description language, such as OWL-S. They are represented in the figure on the opposite end to the client user. For this example, we’ll consider the existence of the following semantic web services:

- *Book Information Agent* – this web-accessible information agent provides information about books and its authors
- *Price Search Web Services* – these web-accessible services provide an item's best-price search service (such as Amazon and Barnes & Noble)
- *Currency Converter Web Service* – this web service provides simple conversion of a monetary value from one currency to another

The service execution agent interacts with these semantic web services to obtain the required information, according to instructions in the compound service's OWL-S Process Model and Grounding descriptions.

The service execution agent bases the execution on the service's OWL-S description. This OWL-S description can, sometimes, be incomplete, i.e., missing atomic services information regarding Grounding information. This can compromise the service's execution simply because the service execution agent doesn't have the necessary information to continue. On the other hand, if the service's OWL-S description is complete but the service execution agent is operating on very dynamic environments (such as pure P2P networks), the information contained in the service's OWL-S description can be easily out-dated (previously existing semantic web services are no longer available in the network). This will also compromise the agent's service execution activity.

To solve this problem, the service execution agent can interact with other service-oriented computing agents, such as service discovery and service composition agents, for example, when it needs to discover new services or when it needs to do some re-planning of compound services that somehow could not be executed, for whatever reasons explained above.

## 5 Service Execution Agent

The Service Execution Agent (SEA) is a broker agent that provides context-aware execution of semantic web services. The agent was designed and developed considering the interactions described in sections 3 and 4 and the internal architecture was clearly designed to enable the agent to receive requests from client agents, acquire/provide relevant context information, interacting with other service coordination agents when relevant and execute remote web services.

This section of the paper is divided into four sub-sections. Sub-sections 5.1 and 5.2 describe the internal architecture of the agent, explaining in detail the internal components and their interactions both internal and external, using the agent FIPA-ACL interface. Sub-section 5.3 describes the execution process that the agent carries out upon request from a client agent. Sub-section 5.4 provides some details on the implementation of the agent.

### 5.1 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). Fig. 2 illustrates the internal architecture of the agent and the interactions that occur between the components and with external entities.

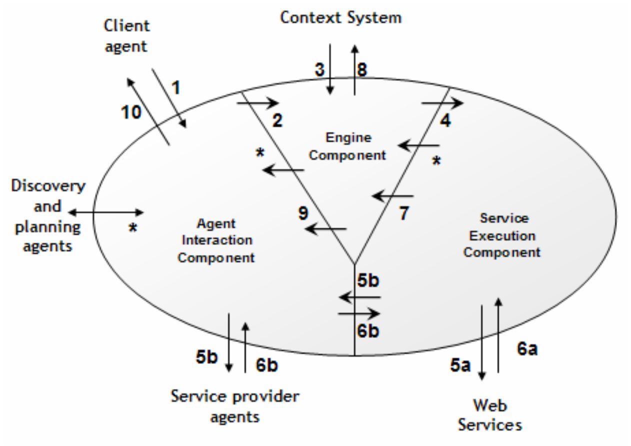

**Fig. 2.** SEA Internal Architecture and Interactions

The AIC was developed as an extension of the JADE platform and its goal is to provide an interaction framework to FIPA-compliant agents, such as SEA's clients (requesting the execution of specified services – Fig. 2, step 1) and service discovery and composition agents (when SEA is requesting the re-discovering and re-planning of specific services – Fig. 2, steps *). This component extends the JADE platform to provide extra features regarding language processing, behaviour execution, database information retrieval and components' communication. 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 (Fig. 2, step 2). The reverse process is also the responsibility of the AIC – receiving data from the EC and processing it into the agents' suitable format to be sent as messages Fig. 2, 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 (Fig. 2, step 2), it acquires suitable context information (regarding potential service providers and other relevant information, such as client location – Fig. 2, step 3) and schedules 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 – Fig. 2, steps *) to discover available providers for the atomic services that are part of the OWL-S compound service. If the service discovery agent cannot find adequate service providers, the EC can interact with a service composition agent (again through the AIC – Fig. 2, 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 (Fig. 2, 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 and 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 and its goal is to execute semantic web services (Fig. 2, 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 (see section 5.3) in order to support the execution of services that are grounded on service provider agents (Fig. 2, steps 5b, 6b). 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 section 5.3. After execution of the specified service and generation of its results, the SEC sends them to the EC (Fig. 2, step 7) for further analysis and post-processing, which includes sending gathered context information to the context system (Fig. 2, step 8) and sending the results to the client agent (through the AIC – Fig. 2, steps 9, 10).

## 5.2 Agent Interface

When requesting the execution of a specified service, client agents interact with the Service Execution Agent through the FIPA-request interaction protocol [10]. This protocol states that when the receiver agent receives an action request, it can either agree or refuse to perform the action. In the first part of this protocol, the execution agent should then notify the other agent of its decision through the corresponding communicative act (FIPA-agree or FIPA-refuse). The execution agent performs this decision process through a service execution's request evaluation algorithm that involves acquiring adequate context information. The Service Execution Agent 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 are not available and the time that requires finding alternatives is longer than the timeframe in which the client agent expects to obtain a reply to the execution request, then the execution agent refuses to perform it. On the other hand, if the execution agent 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. The execution agent can also refuse to perform execution requests if its work load is already too high (if its requests queue is bigger than a certain threshold).

The FIPA-request also states that after successful execution of the requested action, the execution 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).

## 5.3 Execution Process

OWL-S is an OWL-based service description language. OWL-S descriptions consist of three parts: a *Profile,* which tells "what the service does"; a *Process Model*, which tells "how the service works"; and a *Grounding*, which specifies "how to access a particular provider for 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. The 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.

The *Process Model* describes the steps that should be performed for a successful service execution. 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 [19].

The *Process Model* identifies three types of processes: *atomic*, *simple*, and *composite*. *Atomic* processes are directly evocable. They 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).

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* are two possible extensions.

The general approach for the execution of OWL-S services consists of the following sequence of steps: (i) validate the service's pre-conditions, whereas the execution process continues only if all pre-conditions are true; (ii) decompose the compound service into individual atomic services, which in turn are executed by evoking their corresponding service providers using the description of the service providers contained in the grounding section of the service description; (iii) validate the service's effects by comparing them with the actual service execution results, whereas the execution process only proceeds if the service has produced the expected effects; (iv) collect the results, if any have been produced since the service may be only a "*change-the-world*" kind of service, and send them to the client who requested the execution.

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 [3]. In short, WSDL describes the access to a specific service provider for a described OWL-S service.

However, WSDL currently lacks an efficient way of representing agent bindings, i.e., a representation for complex interactions such as the ones that take place with service provider agents. To overcome this limitation, we decided to create an extension of the OWL-S *Grounding* specification, named *AgentGrounding* [15]. This extension is the result of an analysis of the necessary requirements for interacting with agents when evoking the execution of atomic services. In order for an agent to act as a service provider in the Semantic Web, its complex communication schema has to be mapped into the OWL-S Grounding structure. To achieve this mapping, the *AgentGrounding* specification includes most of the elements that are present in Agent Communication Languages. At the message level, the *AgentGrounding* specification includes elements such as the name and the address of the service provider, the protocol and ontology that are used in the process, the agent communication language and the content language. At the message content level, the *AgentGrounding* specification includes elements such as name and type of the service to be evoked and its input and output arguments, including the types and the mapping to OWL-S parameters. For more details and a complete example, we refer the reader to [15].

In order to allow the specification of conditioned constructs which were based on logical expressions, we have introduced the use of *Prolog* for the formal representation of pre-conditions and *If* clauses in the descriptions of the services. Until now, the only support for control constructs that depend on formal representation of logical expressions in OWL-S was done through the use of SWRL. However, for performance reasons, which are shown in section 6, we have decided to use *Prolog*.

### 5.4 Implementation

The Service Execution Agent 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 [2] and the OWL-S API [24].

The JADE agent platform was integrated into the Agent Interaction Component (see section 5.1) of the Service Execution Agent to enable its interaction with client agents and service provider agents.

The OWL-S API is a Java implementation of the OWL-S execution engine, which supports the specification of WSDL groundings. The OWL-S API was integrated into the Execution Component of the Service Execution Agent to enable it to execute both atomic and compound services.

In order to support the specification and execution of *AgentGrounding* descriptions (see section 5.3), we have extended the OWL-S API's execution process engine. The extension of the execution engine allows converting *AgentGrounding* descriptions into agent-friendly messages, namely FIPA-ACL messages, and sending these to the corresponding service provider agents.

To enable the support for control constructs that depend on formal representation of logical expressions using *Prolog*, we extended the OWL-S API with a *Prolog* engine that is able to process logical expressions present in *If/While* clauses and pre-conditions. This extension was done through the use of *TuProlog* [6], an open source Java-based *Prolog*.

## 6 Evaluation and Results

We have stated before that context information would be used to enhance the semantic web service execution process. The referred enhancement consists of the adaptation of the execution agent to a specific situation in a way such that the execution process is done according to a certain situation characteristics, as perceived from available context information. A situation is characterized by the properties that are imposed to the execution agent when this receives an execution request from a client. These properties can be simple limitations such as the client's request for the execution to be done in a certain time-frame or that the service has to be available near a specific location. In this section, we present the evaluation process that illustrates the advantage of using context information in the execution process carried out by the Service Execution Agent.

As described in section 3, the Service Execution Agent uses context information such as availability (to check if service providers often have been offline in the past), average execution time (the time each service provider takes to execute a single service, in average) and queues of pending requests (the number of requests waiting to be processed on each service provider) to determine who the "best" service providers are. This collected context information allows the execution agent to build a sort of *rating* schema of the available service providers, by applying the following simple formula (for each service provider): *`Average Execution Time X Queue of Pending Requests + Average Execution Time = Estimated Time of Execution`*. Combined with the availability history of the service provider (`Estimated Time of Execution X Number of Times Offline in the Last 10 Minutes`) this information will provide the rating (in reverse order) of the service provider within the list of available providers for the same service. Thus, when executing, the Service Execution Agent can use this information to determine the fastest way to execute a compound service or find alternatives in case of failure.

Even though this approach of adding context information to the execution process improves the way compound services are executed (by allowing the determination of the best service providers in a specific situation and by providing a failure recovery method), it introduces an *overhead* that doesn't exist in semantic web services execution environments that do not consider using context information. This *overhead* is composed of the procedures that the execution agent must perform when dealing with the remaining elements of the service coordination infrastructure: communicating and managing conversations with other agents (client, service discovery and composition agents), retrieving and processing context information (related to the service providers) and preparing the execution of compound services. It is important to determine how this *overhead* influences the overall execution time.

Table 1 shows the average behavior of the Service Execution Agent in the repeated execution (50 times) of the compound service (with 5 atomic services) described in section 4. The Normal Execution Time (NET) is the time that it takes to execute the web services; the Overhead Execution Time (OET) is the time that SEA takes to perform the mentioned procedures related to context information processing, conversations' management and preparation of execution; the Total Execution Time (TET) is the total time of execution: NET + OET.

**Table 1.** Execution time details (in seconds) of the repeated execution (50 requests) of the "book best-price search" service.

| | **Total** | **Average** | **Max** | **Min** |
|---|---|---|---|---|
| **TET** | 1244,35 | 25,39 | 33,8 | 20,15 |
| **NET** | 1242,46 | 25,36 | 33,79 | 20,12 |
| **OET** | 1,89 | 0,04 | 0,15 | 0,01 |

Table 1 provides the execution time details of the entire test. As we can see in Table 1 the average overhead execution time is very small (0,04 seconds). This value is hardly noticeable by a user of such a system. Moreover, in some tests the use of this kind of service execution approach allowed the reduction of the overall execution time due to the fact that the execution agent always tried to find the fastest service provider (the one which was online for longer periods in the past or had a lower work load or had a lower average execution time).

It is important to detail the *overhead* execution time in order to determine which are the most time consuming processes. Fig. 3 shows the overall behavior of the *overhead* during the test described in Table 1.

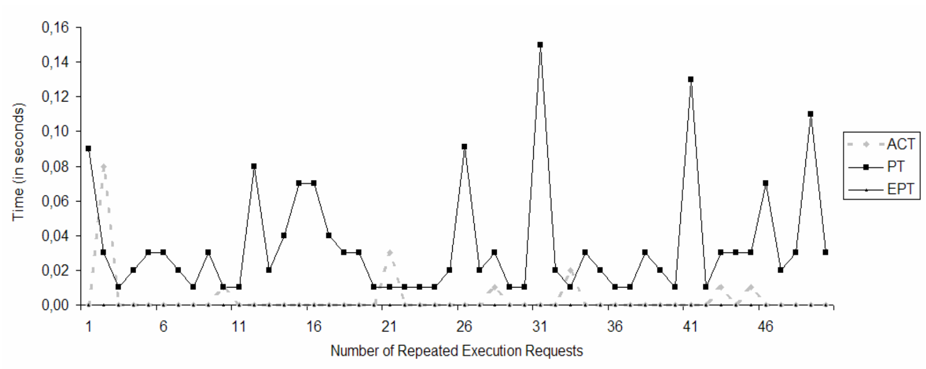

**Fig. 3.** Detailed *Overhead* Execution Time of the test described in Table 1

The gray dashed line represents the time used in the communication with other agents; the regular black line represents the service processing time (gathering context information and adapting accordingly); and the dashed black line (hardly noticeable in the figure because of being so low) represents the service execution process preparation, which mainly consists of changing the service description whenever necessary (for example, when service providers change and the grounding have to be changed accordingly). By analyzing the chart in Fig. 3, we can easily determine that the specific task that contributes to increase the *overhead* execution time is the preparation task, i.e., the access to the context system to retrieve relevant context information. We can also see that, in some cases, the time used in communication with other agents is the most time-consuming event. However, this can be explained by the environment in which agents operate, whereas the communication process in highly dynamic environments is influenced by network traffic. A deeper analysis of the preparation time allowed us to determine that the most time-consuming process when accessing the context system is in fact the communication with the context system through its JINI interface [5]. So, the preparation time can be reduced if the access to the context system can be improved through the use of a different communication interface, which proves to be faster and more reliable.

The described test was done using the "book best-price search" service description with logical expressions described in Prolog. However, to compare our approach of using Prolog with the existing approach originally implemented in the OWL-S API, we have decided to repeat the test, this time using SWRL for the representation of logical expressions. Fig. 4 details the execution times for each atomic service of the "book best-price search" service, this time using SWRL. The use of logical expressions is only done in the atomic service "Compare Prices", which consists on a *If* clause that compares the prices of both book shops (Amazon and Barnes & Noble) and determines which one is cheaper.

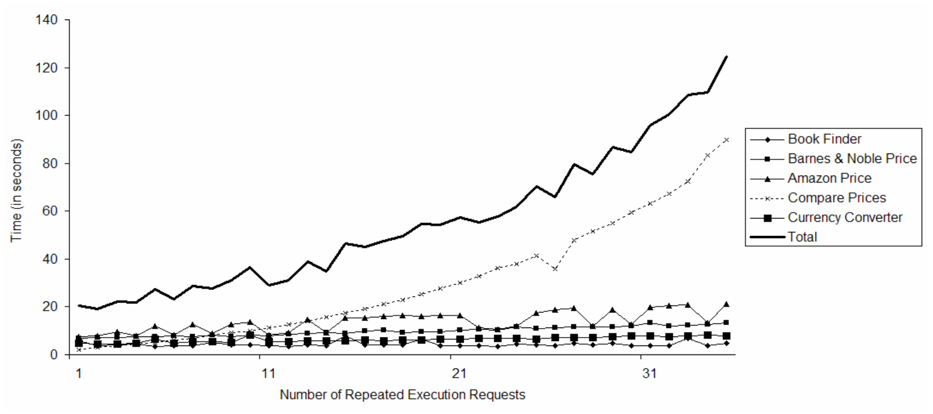

**Fig. 4.** Execution test of the "book best-price search" service with SWRL expressions

As we can see in Fig. 4, the "Compare Prices" atomic service is the only one that does not have a constant behavior throughout the entire execution test. In fact, this service increases its execution time on each cycle, thus contributing to the increase of the overall execution time of the compound service. This behavior may be explained by an inefficient implementation of the SWRL extension or the SPARQL [22] engine (used for the queries of logical expressions) in the OWL-S API, however we were not able to determine the exact cause of this inefficient behavior.

The overall evaluation of the work shows that the Service Execution Agent can be used in highly dynamic environments since it can efficiently distribute the *execution work* to the appropriate service providers, hence providing a faster and more reliable service to a user with time, device and location constraints.

## 7 Conclusions

We have presented a framework to enable the execution of semantic web services using a context-aware broker agent – the Service Execution Agent. The developed approach uses context information to determine the appropriate service providers for each situation. In doing so, SEA becomes a highly efficient broker agent capable of distributing the execution work among the available service providers, thus providing a more useful and faster service to its clients. Evaluation results show that the introduction of context-aware capabilities and the interaction within a service coordination infra-structure not only add a very small overhead to the execution process, as it can sometimes improve the overall execution time, thus offering a valid and reliable alternative to the existing semantic web services execution environments.

Future work will be based on the further analysis of the use of *Prolog* and SWRL in the logical representation of conditions, in order to determine the exact cause of the inefficient behaviour when using SWRL and whether *Prolog* is a better formal representation solution for logical expressions in the execution of semantic web services. Also, we intend to perform comparison tests of our approach with other semantic web services execution approaches that may have been recently developed.

**Acknowledgments.** This work has been supported in part by the European Commission under the project grant FP6-IST-511632-CASCOM. The authors would like to thank the reviewers for providing such useful comments for the overall improvement of the paper.

## References

1. Abowd, G. D., Dey, A., Orr, R. and Brotherton, J. (1998). Context-awareness in wearable and ubiquitous computing. Virtual Reality, 3:200–211.
2. Bellifemine F., Poggi A., Rimassa G. (2001) Developing multi-agent systems with a FIPA-compliant agent framework, Software-Practice and Experience 31 (2): 103–128 Feb 2001
3. 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.
4. Clocksin, W.F.; Mellish, C.S. (1981). Programming in Prolog. Springer-Verlag, New York.
5. Costa, P., Botelho, L. (2005). Generic Context Acquisition and Management Framework. First European Young Researchers Workshop on Service Oriented Computing. Forthcoming.

6. Denti, E., Omicini, A., Ricci, A. (2005). Multi-paradigm Java-Prolog integration in TuProlog. Science of Computer Programming, Vol. 57, Issue 2, pp. 217-250.
7. 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.
8. 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.
9. Fensel, D. and Bussler, C. (2002). The Web Service Modeling Framework – WSMF. Electronic Commerce: Research and Applications, 1 (2002) 113-137.
10. FIPA Members. (2002) Foundation for Intelligent Physical Agents website. http://www.fipa.org/.
11. Fujitsu Labs of America. (2001). April Agent Platform project website. http://www.nar.fujitsulabs.com/aap/about.html
12. Helin, H., Klusch, M., Lopes, A., Fernández, A., Schumacher, M., Schuldt, H., Bergenti, F., Kinnunen, A. (2005). CASCOM: Context-aware Service Co-ordination in Mobile P2P Environments. Lecture Notes in Computer Science, Vol. 3550 / 2005, ISSN: 0302-9743, pp. 242 - 243
13. 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/
14. 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.
15. Lopes, A., Botelho, L.M. (2005). SEA: a Semantic Web Services Context-aware Execution Agent. AAAI Fall Symposium on Agents and the Semantic Web. Arlington, VA, USA.
16. 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.
17. Mitkas, P., Dogac, A. (2002). An Agent Framework for Dynamic Agent Retraining: Agent Academy. eBusiness and eWork 2002, 12th annual conference and exhibition.
18. Nwana, H., Ndumu, D., Lee, L., and Collis, J. (1999). ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. Applied Artifical Intelligence Journal, vol. 13, no. 1 pp 129-186.
19. OWL Services Coalition. (2003). OWL-S: Semantic Markup for Web Services. DARPA Markup Language Program.
20. Paolucci, M. and Srinivasan, N. (2004). OWL-S Virtual Machine Project Page. http://projects.semwebcentral.org/projects/owl-s-vm/.
21. 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.
22. Prud'hommeaux, E. and Seaborne, A. (2004). SPARQL Query Language for RDF. W3C Working Draft. http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/
23. 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/.
24. Sirin, E. (2004). OWL-S API project website. http://www.mindswap.org/2004/owl-s/api/.