Chapter 12

Semantic Web Service Execution

Luís M. Botelho, António L. Lopes, Thorsten Möller and Heiko Schuldt

12.1 Introduction

Service execution comprises all the activities that need to be carried out at runtime to invoke one or several (Web) services in a coordinated manner. These activities include initiation, control and validation of service invocations. Since each service is supposed to create side effects as manifested by the functionality that it implements, both the service user and service provider are interested that certain properties for execution are guaranteed. The two most prominent properties are guaranteed termination and reliability, that is, sustaining a consistent state before and after execution even in the presence of failures. Those aspects become of particular interest when it comes to (i) execution in distributed environments where more than one software entity might be involved, and (ii) execution of composite services, i.e., processes.

In the conventional approach to service composition[1], the act of assembling individual atomic or composite services to a process is a manual task, done by a process designer[2]. CASCOM follows a more innovative approach in which processes are generated automatically by the service composition planning agent, using planning techniques, see Chapter 11. This chapter describes two approaches to reliable service execution, which take into account both atomic and composite services. These include augmenting the planned service composition with monitoring assertions that can be checked in order to determine if the service has been successfully executed, and in the case of failure, to allow necessary re-planning by the planning agent.

[1] In the literature also referred to as service orchestration.

[2] This is commonly referred to as business process modeling.

As input, service execution takes service descriptions represented by the Web Ontology Language for Services (OWL-S) and a set of actual values of the services input parameters. On successful execution a set of output values (results) will be returned to the invoker whereby the values are associated to the output parameters in the service description. Either input or output sets might be empty, or both – in this case the invoked service produces just side effects. Furthermore, execution requires grounding for each service. It consists of (i) an address to a concrete service instance and (ii) information about the protocol to be used to interact with the service instance. Consequently, leaving out the grounding in an OWL-S document results in an abstract service description independent from how it is realised. To integrate state of the art Web Services that are based on WSDL [4] and SOAP, OWL-S allows specifying so-called WSDL groundings. In this chapter we concentrate on this type as it is broadly used today. However, to underpin the applicability of OWL-S to agent-based systems, we also introduce an agent grounding. In principle, there are no limitations to develop other grounding types like for instance, grounding to methods of Java classes.

Two approaches were developed for delivering the service execution functionality in the CASCOM environment: a centralized approach (see Section 12.3), where a single specialized *Service Execution Agent* (SEA) can execute an entire composite service; and a dynamically distributed approach see Section 12.4), where different well coordinated and co operating Service Execution Agents in the environment contribute to the execution of parts of a composite service.

We have decided to adopt the agent paradigm, creating SEAs to facilitate the integration of this work in open agent societies [10], enabling these not only to execute Semantic Web Services but also to seamlessly act as service providers in a large network of agents and Web Services interoperation.

12.2 Composite Service Execution

Today's business, e.g., healthcare, and even scientific applications often involve interactions with several service providers to realize use cases that can be found in the applications. As a consequence, the services involved will be composed into processes that reproduce the use cases when executed. The composition can be done either manually by process designers probably supported by design tools, or automatically by service composition planning systems, see Chapter 11. Eventually, the planned processes and input data will be issued directly to execution systems. In a centralized approach composite services would be executed completely by one single software agent. On the opposite end, execution can also be split up at each inner service that is part of a composite service. The so produced chunks can be executed in a well coordinated manner by different while co-operating software agents, i.e., in a distributed way. Of course, all the properties that are declared in the composite service with respect to control and data flow must be maintained the same way as in a centralized approach to ensure consistent and reliable execution.

The motivation to distribute composite service execution is always to be found either in demands for scalability or by needs to optimize the execution process in inherently distributed environments such as service-oriented architectures. The reasons for optimization almost automatically emerge and can be manifold. Examples are mostly related to nonfunctional aspects like interoperability, efficiency, performance, scalability, availability, reliability, security, and so on. One concrete example that has a direct impact on efficiency, performance, and scalability is the reduction of overall data amount that has to be transferred in a distributed execution. Another one might be the automatic selection of execution agents that are currently idle, thus providing load balancing, which in turn has a positive effect on scalability. Since these advanced mechanisms always require the collection and availability of additional information at runtime this also gives motivation for the presence of a generic context information system that allows to store, handle and query them, see Chapter 13.

12.2.1 General OWL-S Execution Procedure

OWL-S [6] is an OWL-based service description language. OWL-S descriptions consist of three parts: a *Profile*, which describes *what the service does*; a *Process Model*, which describes *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 an abstract service description 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 of OWL-S services, 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 chapter.

The Process Model describes the steps that should be performed for a successful execution of the whole service that is described. These steps represent two different views of the process: first, it produces a data transformation of the set of given inputs into the set of produced outputs; second, it produces a transition in the world from one state to another. This transition is described by the preconditions and effects of the process.

There are 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 (atomic or composite) processes. These represent several-steps executions, whereby the control flow 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 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 described effects by comparing them with the actual effects of the service execution, whereas the execution only proceeds if the service has produced the expected effects; (iv) collect the results, if there are any results, and send them to the client who requested the execution. Notice that the service may be just a *change-the-world* kind of service, i.e., it produces just side effects.

12.3 Centralized Approach for Service Execution

This section presents the research on agent technology development for context-aware execution of Semantic Web Services, more specifically, the development of a Service Execution Agent for Semantic Web Services execution. The agent uses context information to adapt the execution process to a specific situation, thus improving its effectiveness and providing a faster and better service.

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. Using OWL-S service descriptions it was not possible up to now to have service provider agents in addition to the usual Web Services because the grounding section of OWL-S descriptions lacks the necessary expressiveness to describe the complex interactions of agents. 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* [13].

We have also introduced the use of *Prolog* [5] for the formal representation of logical expressions in OWL-S control constructs. To our knowledge, the only support for control constructs that depend on formal representation of logical expressions in OWL-S is done through the use of *SWRL* [12] or *PDDL* [14].

12.3.1 Service Execution and Context-Awareness

Context-aware computing is a computing paradigm in which applications can discover and take advantage of context information to improve their behaviour in terms of effectiveness as well as performance. As described in [2] context is any information that can be used to characterize the situation of an entity. Entities may be persons, places or objects considered relevant to the interaction between a user and an application, including users and applications themselves.

We can enhance this definition by stating that the context of a certain entity is any information 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 an entity; analyses that information, either by treating it as an isolated single 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 service execution process, by adapting it to the specific situation in which the agent and its client are involved, at the time of the execution process. This is done by interacting with a general purpose (i.e., domain independent) context system [7] (see also Chapter 13) for obtaining context information, subscribing desired context events and providing 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 workload, and hence the one that offers 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 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 after having waited for the connection timeout to occur. If SEA learns that a given service provider is not available it will contact the service discovery agent or the service composition agent requesting that a new service provider is discovered or that the compound service is re planned. This situation-aware approach using context information on-the-fly helps SEA to provide a value-added execution service.

12.3.2 Service Execution Agent

The Service Execution Agent (SEA) is a broker agent that provides context-aware execution of services on the semantic web (whether they are provided by Web Services or agents). The agent was designed and developed considering the interactions described in Section 12.3.1 and the need to adapt to situations where the interaction with other service-oriented agents is required. Its internal architecture was clearly designed to enable the agent to receive requests from client agents, acquire and provide relevant context information, interacting with other service coordination agents when necessary and execute remote services.

This section is divided into four parts. Sections 12.3.2 and 12.3.2 describe SEA internal architecture, explaining in detail the internal components and their interactions. Section 12.3.2 describes *AgentGrounding*, an extension of the OWL-S Grounding specification that allows agents to act as service providers in the Semantic Web. Section 12.3.3 provides some details on the implementation of the agent.

Internal Architecture

The developed agent is composed of three components: the Agent Interaction Component (also referred to as Interaction Component), the Engine Component and the Service Execution Component (also referred to as Execution Component). Figure 12.1 illustrates SEA internal architecture and the interactions that occur between its components.

The Interaction Component was developed as an extension of the JADE platform and its goal is to provide an interaction framework with FIPA-compliant agents, such as its client agents (Figure 12.1, steps 1 and 10), service discovery and service composition agents (whenever SEA requests the re discovery or the re planning of specific services). This component extends the JADE platform with extra features regarding language processing, behavior execution, database information retrieval and inter components communication.

Among other things, the Interaction Component is responsible for receiving messages, parsing them into a format suitable for the Engine Component to use

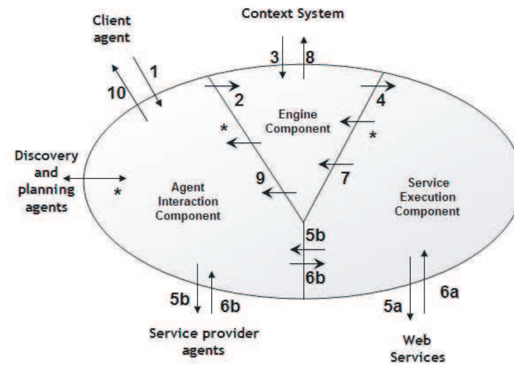

Figure 12.1: SEA's Internal Architecture and Interactions between the several components

(Figure 12.1, step 2). The reverse process is also the responsibility of the Interaction Component — receiving data from the Engine Component and converting it into the suitable format to be sent as FIPA-ACL messages (Figure 12.1, step 9).

The Engine Component is the main component of the execution agent as it controls the agent's overall activity. It is responsible for pre processing service execution requests by acquiring/providing context information from/to the context system and deciding when to interact with other agents (such as service discovery and composition agents).

When the Engine Component receives an OWL-S service execution request (Figure 12.1, step 2), it acquires suitable context information (regarding potential service providers and other relevant information, such as client location Figure 12.1, step 3) and schedules the execution process.

The Service Execution Agent interacts with a service discovery agent (through the Interaction Component — Figure 12.1, steps *) to discover available providers for the atomic services that are used in the OWL-S compound service. If the service discovery agent cannot find adequate service providers, the Engine Component can interact with a service composition agent (again through the Interaction Component — Figure 12.1, 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 Engine Component sends it to the Service Execution Component (Figure 12.1, step 4), for execution. Throughout the execution process, the Engine Component is also responsible for providing context information to the context system, whether it is its own information (such as its queue of service execution requests, average execution time) or other entities' relevant context information (such as availability of providers and average execution time of services).

The Execution Component was developed as an extension of the OWL-S API

and its goal is to execute OWL-S service descriptions (Figure 12.1, steps 5a and 6a) with WSDL grounding information. The extension of the OWL-S API allows performing the evaluation of logical expressions in conditioned constructs, such as the *If-then-Else* and *While* constructs, and in services' 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 12.1, steps 5b, 6b). This extension is called *AgentGrounding* and it is explained in detail in Section 12.3.2.

When the Execution Component receives a service execution request from the Engine Component, it executes it according to the description of the service's process model. This generic execution process is described in Section 12.2.1.

After the execution of the specified service and the generation of its results, the Execution Component sends them to the Engine Component (Figure 12.1, step 7) for further analysis and post processing, which includes sending gathered context information to the context system (Figure 12.1, step 8) and sending the results to the client agent (through the Interaction Component — Figure 12.1, steps 9 and 10).

Agent Interface

When requesting the execution of a specified service, client agents interact with the Service Execution Agent through the FIPA-request interaction protocol [9]. This protocol states that when the receiver agent receives an action request, it can either agree or refuse to perform the action. The receiver (i.e., the execution agent) should notify the sender of its decision through the corresponding communicative act (FIPA-agree or FIPA-refuse).

The decision to whether or not to accept a given execution request depends on the results of a request evaluation algorithm which involves acquiring and analysing adequate context information. The Service Execution Agent (SEA) will only agree to perform a specific execution if it is possible to execute it, according to the currently available context information. For example, if necessary service providers are not available and the time required to find alternatives is longer than the given 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 the request. The execution agent can also refuse execute requests if its workload is already too high (if its requests queue is longer than a certain defined constant).

If during the execution process the execution agent is unable to perform the entire request, it notifies the client agent by sending a FIPA-Failure message, which includes the reason for the execution failure. The FIPA-request interaction protocol 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: a message containing the results obtained from the execution of the service, or a message containing just a notification that the service was successfully executed (when no results are produced by the execution).

OWL-S Grounding Extension for Agents: AgentGrounding

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

WSDL currently lacks a 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* [13]. 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.

The AgentGrounding specification includes most of the elements that are present in Agent Communication. 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.

Figure 12.2 is an example of an OWL-S service grounding 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.

The AgentGrounding extension allows the specification of groundings such as the one described in Figure 12.2. These groundings can be executed through a request sent to the Service Execution Agent, which in turn executes the service by invoking the specified service providers. 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 Figure12.2 describes a service named HospitalFinderService, which is grounded to an action find-hospital that accepts, as input, a

```
<!-- Grounding description -->
<ag:AgentGrounding rdf:ID="HospitalFinderGrounding">
  <s:supportedBy rdf:resource="#HospitalFinderService"/>
  <g:hasAtomicProcessGrounding rdf:resource="#HospitalFinderProcessGrounding"/>
</ag:AgentGrounding>
<ag:AgentAtomicProcessGrounding rdf:ID="HospitalFinderProcessGrounding">
    <g:owlsProcess rdf:resource="#HospitalFinderProcess"/>
    <ag:agentName>hospitalinfo@cascom</ag:agentName>
    <ag:agentAddress>http://...</ag:agentAddress>
    <!-- Service Identification -->
    <ag:serviceName>find-hospital
    </ag:serviceName>
    <!-- Service Arguments -->
    <ag:hasArgumentParameter>
      <ag:ArgumentParameter rdf:ID="location">
        <ag:argumentType>java.lang.String</ag:argumentType>
        <ag:owlsParameter rdf:resource="#Location"/>
        <ag:paramIndex>1</ag:paramIndex>
      </ag:ArgumentParameter>
    </ag:hasArgumentParameter>
    <ag:serviceOutput>
      <ag:ArgumentVariable rdf:ID="hospital-info">
        <ag:argumentType>java.lang.String</ag:argumentType>
        <ag:owlsParameter rdf:resource="#HospitalInfo"/>
      </ag:ArgumentVariable>
    </ag:serviceOutput>
   <!-- Other information -->
   <ag:serviceType>action</ag:serviceType>
   <ag:protocol>fipa-request</ag:protocol>
   <ag:agentCommunicationLanguage>fipa-acl</ag:agentCommunicationLanguage>
   <ag:contentLanguage>fipa-sl</ag:contentLanguage>
   <ag:serviceOntology>hospital-finder-ontology</ag:serviceOntology>
</ag:AgentAtomicProcessGrounding>
```

Figure 12.2: Example of *AgentGrounding* description

single string named location. This location argument is linked to the OWL-S service input parameter Location. The action returns, as output, also a string, named hospital-info, which is linked to the OWL-S service output parameter HospitalInfo. Other information that can be extracted from this grounding is the interaction protocol (fipa-request), the agent communication language (fipa-acl), the ontology (hospital-finder-ontology) and the content language (fipa-sl) to be used in the invocation message. The Service Execution Agent can use this information to send the FIPA message that is described in Figure 12.3.

The information extracted from the *AgentGrounding* example in Figure 12.2 plus the information regarding the concrete service input (in this example, the string '9,8324W 38,12345N'), which comes in the received service execution request, is enough for the agent to be able to create the message in Figure 12.3.

The AgentGrounding specification allows the representation of several in-

```
(REQUEST
 :sender (agent-identifier :name sea@cascom)
 :receiver (set (agent-identifier :name hospitalinfo@cascom
   :addresses (sequence http://...)))
 :content "((action
     (agent-identifier :name hospitalinfo@cascom)
     (find-hospital :location \"9,8324W 38,12345N\")))"
      :language fipa-sl
      :ontology hospital-finder-ontology
      :protocol fipa-request)
```

Figure 12.3: Message generated from the example in Figure 12.2

stances of messages that can be sent to FIPA compliant agents, including the use of different performatives, agent communication languages and content languages.

12.3.3 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 [3] and the OWL-S API [19].

The JADE agent platform was integrated into the Agent Interaction Component (see Section 12.3.2) 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 12.3.2), we have extended the OWL-S APIs 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 [8], an open source Java based Prolog.

12.4 Distributed Approach for Service Execution

The distributed approach for service execution differs from the centralised approach (see Section 12.3) in the sense that at runtime execution is not limited to being handled by just one execution agent but might involve several distinct agents[3]. This fundamental expansion results in the advantages described earlier in this chapter but brings in new characteristics that need to be tackled in order to retain reliable execution.

The following sections present presents research results in the area of distributed execution of composite Semantic Web Services. Section 12.4.1 starts by pointing to the general assumptions. The main part of this chapter describes the basic concept and structure of the distributed approach and the protocol that

[3]Whether they are actually physically separated, i.e., run on different peers, is a matter of the agent platform used and the concrete deployment of agent instances. Consequently, the distributed approach imposes no constraints on the organisation of agents beyond what is implicated by the agent platform used. On the other hand, not physically separated would mean that they run on one peer within different threads or processes.

represents the interaction model between client agents and execution agents. The chapter concludes by briefly presenting the implementation of the service execution system that was developed at the University of Basel.

To start out, distributed service execution invariably requires a certain strategy to organise and co-ordinate distribution. Basically, such a strategy includes a method to control the flow of execution among the participating agents and the invocation of the service itself. In general, various kinds of such strategies can be designed. The properties of the approach that has been developed in the CASCOM project will be described in Section 12.4.2.

12.4.1 General Assumptions

For distributed service execution, we first assume that a composite service contains an arbitrary number of service invocations whereby the composition structure is equal to a directed acyclic graph, i.e., combined sequential and parallel flows together forming processes as denoted in [16]. Second, as a basis for correct process execution, each service invocation is assumed to be atomic and compensatable. This means that the effects of a service can be undone, if necessary, after the invocation has returned. Otherwise, unwanted side effects of aborted or reset executions may remain and exactly-once execution semantic could not be guaranteed. For services which do not comply with the atomicity requirement, we assume that a wrapper can be built which adds this functionality. Third, we assume that services are stateless, i.e., that they never have to remember anything beyond interaction. In our approach, execution state (i.e., intermediate results) is solely stored by the execution system, as part of the process (composite service) instance. Finally, our approach considers the crash failure model, which means that components such as services and machines may fail by prematurely halting their execution.

12.4.2 Execution Strategy

Carrying out execution of composite services in a distributed environment consisting of more than one execution agent requires the definition of a certain strategy. The strategy is built on three core properties: First, it defines how to divide the composite service into sections which can be executed in a distributed way afterwards. Second, it defines where and when to distribute those sections to different execution agents. Finally, it defines a mechanism for control between the participating agents to guarantee consistent execution. Altogether they imply the way execution will be actually done at runtime.

The selection or design of an execution strategy needs always to be accompanied by an analysis of the service environment. The two main aspects that such an analysis would cover are (i) the technologies used, like SOAP based Web Service interactions, and (ii) the deployment structure of service providers and their (physical) relation to service clients (agents in the CASCOM architecture).

In the following, three different dimensions for categorisation of execution strategies are described. Sections 12.4.2 and 12.4.2 describe two general purpose approaches, which have their own characteristics regarding the three dimensions. The former one was chosen for implementation of the distributed service execution agent and we will outline some advantages and disadvantages.

In general, a strategy for distributed execution of composite services must initially define whether execution control takes place in a centralised or decentralised way. The centralised approach requires a dedicated manager agent that takes the responsibility of supervising and co-ordinating execution after receipt of the composite service. In doing so, the manager agent invokes the actual execution agents according to the control structure of the composite service. The execution agents in turn invoke the actual services and await the results of the service invocations. In contrast to the centralised approach the decentralised approach does not require a dedicated agent, that is, the co-ordination role is no longer limited to stay at a single manager agent. In theory, the co-ordination role either can be forwarded among the participating agents or the agents coalesce to share the co-ordination role. Whereas the manager agent in the centralised approach turns out to become a bottleneck and a single point of failure both problems are eliminated in the decentralised approach. On the other hand, a decentralised approach comes with the trade-off that more advanced control mechanisms are required, accompanied by higher (communication) efforts in case of failure handling and initial preparation before execution (see Section 12.4.2). In addition, the decentralized approach allows for balancing the load among different agents and for taking into account the dynamics of practical settings where agents might frequently leave or others join.

Starting from the process model of OWL-S services, execution can be further classified depending on whether the services are atomic or composite (see Section 12.2.1). While execution of an atomic OWL-S service implicates just one service invocation, the number of service invocations for composite OWL-S services intuitively relates to the number of atomic services out of which they are *composed*. In general, we always refer to composite services since execution of atomic services in fact does not require an advanced distributed execution strategy. The reason is that invocation of a Web Service is defined as a request reply pattern between a service client and the provider that cannot be further split up. As a conclusion, it is now obvious what the smallest granularity for sections of a composite service is: the atomic services. At the same time it is also the preferred granularity as it directly maps to the OWL-S process model[4]. Hence, it is not remarkable that most execution strategies would size the sections for distribution equal to the atomic services. Finally, the general assumption is that agents execute those sections of the composite service. For instance, the most straightforward approach would associate each atomic service within a composite service to one execution

[4]The process model basically relates to a directed graph, whereby the nodes are (atomic) services and vertices represent the control flow.

agent which invokes it, i.e., a composite service consisting of n atomic services would require n execution agents ($n \geq 1$).

Yet another aspect for categorisation of execution strategies relates to whether execution agents and (Web) service providers are tightly integrated/coupled or not. By tightly integrated we mean that an execution agent basically wraps one or more services, that is, both run on the same machine, probably even in the same process. In addition, the implementation of the invocation of the Web Service by the wrapper agent itself might bypass the whole Web Services communication stack and might be done without this overhead by direct programmatic calls. Furthermore, this type mostly comes along with the intention that an execution agent wraps a fixed set of services, i.e., the set of services is expected to never change. As opposed to tight integration, remote coupling means that the execution agent and the (Web) service provider are distant from each other, that is, the execution agent invokes the service in the standard way using its communication layer and protocol. For instance, the today's broadly adopted Web Service stack: HTTP based SOAP or REST. Finally, this type mostly comes along with a generic design of the execution agent which is able to invoke any remote service.

Dynamically Distributed Execution Strategy with Remote Coupling of Execution Agent and Web Services

The approach described hereafter was designed to address recent developments in service-oriented architectures in which the spreading and application of Web Service standards like WSDL and SOAP have reached a level where Web Service providers almost always apply those technologies to publish and provide access to their Web Services. As a matter of fact, the question had to be raised whether a tight coupling of execution agents and Web Services is appropriate, assumable, even feasible in practice or whether a strategy can be found which fits well to the current exploitation of Web Services: A strategy which considers remote invocation of services using the common Web Services communication stack and which is still robust against failures. As a result, this strategy supports the requirement that service providers want to remain Web Services conform with respect to the service interfaces.

Figure 12.4 illustrates the course of the dynamically-distributed and decentralized excution strategy based on a very simple scenario where a composite service containing a sequence of three Web Service invocations $WS1$ to $WS3$ should be executed. It is also assumed that a composite service is split up into sections equal to the atomic services inside. The strategy works as follows.

1. The client agent submits a valid OWL-S composite service description together with the input data to one available execution agent, SEA_1 in this case.
2. SEA_1 parses the service description and immediately starts invoking the first Web Service $WS1$. After return of the result from $WS1$ the first section is

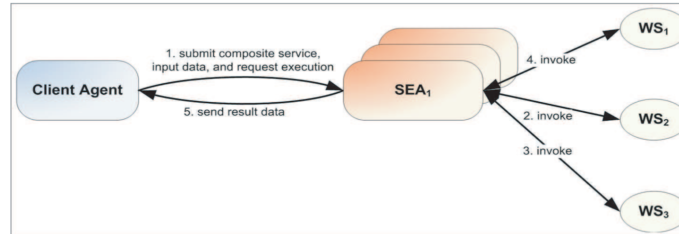

Figure 12.4: Dynamically-distributed execution strategy with remote coupling of execution agent and Web Services

finished.

3. A decision step follows to determine whether execution of the next section should be made by the same agent or whether the execution state should be transferred to another execution agent and continue there. In this decision step arbitrary heuristics can be used to figure out how to continue, i.e., on which agent to continue. For instance, it would be possible to design context based heuristics making it possible to adapt dynamically to overloaded, slow, expensive, or failure-prone context situations and transfer execution state to other execution agents with superior context situations. Additionally, it is also possible to face the effort for transferring execution state to another execution agent with the benefit available there. As a result, the system can be adapted to the circumstances that exist in practical settings by designing distribution heuristics that incorporate the characteristics[5].
4. Execution of the next section might continue on SEA_1 until $WS3$ returned its result. The same decision step is done after each section.
5. In step 5 the composite service output (result) is returned to the client agent.

The heuristics based approach for distribution is able to cope with external failure situations but not with crash situations of the current execution agent itself. To overcome this problem, two possibilities exist. It is either possible to persist the current execution state at the current agent, or to replicate the execution state online to another execution agent. Whereas the first solution is only appropriate for short interruptions whereby the agent gets rebooted immediately afterwards and restarts the interrupted execution, the second solution is more robust since execution can be immediately overtaken by the agent which has the replicated execution state. Therefore we intend to extend the implementation of the execution agent with the latter solution.

[5]Two simple heuristics have been implemented for the prototype. First, a CPU load based heuristic. A transfer of the execution state takes place only if the average CPU load on the current execution agent exceeded a certain threshold in a preceding timeframe. For validation purposes, a round-robin heuristic has been implemented that transfers to another execution agent in either case but always selects that execution agent that has been used least recently.

Provided that the heuristics based decision always returns with the result to continue at the same execution agent, execution is in fact not distributed and the communication effort is minimal. According to the actual heuristic used, this situation represents the optimal execution environment then. On the opposite end, transfer to another agent on each decision computation represents the most suboptimal execution environment according to the heuristic and also involves the highest communication overhead. Therefore it turns out that the more precarious the context environment is the more efforts must be undertaken.

Another characteristic of this strategy is that its properties do not deteriorate assuming that (Web) service providers would be tightly integrated with agents. Assuming this change the execution agent which currently does invocations should then communicate to other agents using ACL messages instead of using the Web Services stack. Consequently, only the message layer for service invocations needs to be replaced/extended by a communication layer for ACL messages.

Fully Distributed Execution Strategy with Tight Integration of Execution Agent and Web Services

The determinative characteristic of this strategy is the assumption that execution agents and (Web) service providers are tightly or locally integrated, as described above. Consequently, execution agents need to be deployed and made available for every pre-existing service provider in the infrastructure. This property distinguishs it from the dynamically-distributed strategy described in the preceding section. This approach for distributed while decentralized composite service execution is related to techniques described in [18] and [15].

Figure 12.5 illustrates the course of the fully-distributed and decentralized execution strategy. Likewise Figure 12.4 it also assumes a very simple scenario where a composite service containing a sequence of three Web Service invocations $WS1$ to $WS3$ should be executed. The client agent — according to the setting in the CASCOM architecture this would actually be the SCPA — submits the valid OWL-S composite service description together with the input data to any available execution agent, for instance SEA_0. This execution agent parses the given service description for the first atomic service provider, which is $WS1$. With this information the agent resolves the execution agent SEA_1 as it is the wrapper of $WS1$ — notice that we assume the availability of some kind of directory containing a mapping from service providers to execution agents. In the next step SEA_0 forwards the complete service description and input data to SEA_1 co-requesting execution start. By forwarding the complete service description each agent is able to resolve the next agent to forward control and results to because this information can be read from the process model of the service description. Remember that the process model specifies the control and data flow, i.e., the order and type of service invocations. In the scenario, after completion of execution of $WS1$, SEA_1 would resolve the agent for $WS2$ which is SEA_2 and again forward the complete service description together with the input data and result 1 produced by $WS1$. This

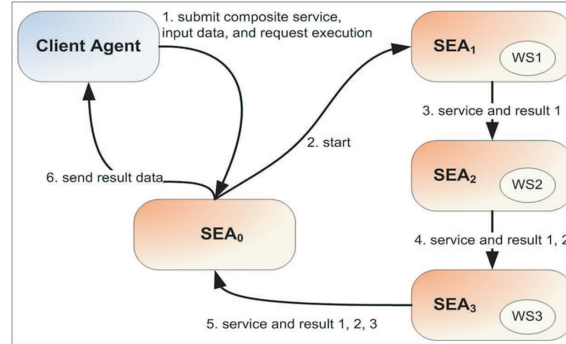

Figure 12.5: Course of fully-distributed execution strategy with tight integration of SEA and service provider

procedure continues until SEA_3 has finished invocation of $WS3$ with result 3. In step 5 SEA_3 sends results 1 to 3 back to SEA_0 which just maps them to the composite service's output (result). Provided that all service invocations returned without failures the result is sent back to the client agent in step 6.

In case of failures on one execution agent, failed service invocations, or in case of violated pre- or post-conditions execution either has to be rolled back or a re-planning could be initiated trying to find alternative services and continue execution if an alternative was found. Assuming transactional properties of the services (as stated in Section 12.4.1) a roll back can be done based on the fact that each execution agent knows its adjacent predecessor(s) from the process model of the composite service. A rollback then requires moving back along the control path step by step and rolling back each service invocation done.

The downside of the simple variant of this approach is that intermediate results produced by service invocations and input data will be forwarded in any case no matter if they are actually required on every execution agent, i.e., the data flow is not optimal and that SEA and atomic service need to be tightly coupled. For instance, in the scenario illustrated in Figure 12.5, if $WS1$ produces large volumes of data as result 1 they will be forwarded to SEA_2 and SEA_3 in any case. In a first optimised version data (input and output data, intermediate results) would only be forwarded between the agents where it is required, thus reducing the overall data communication amount. Furthermore, it is not necessary to forward the composite OWL-S service description from agent to agent. In a second optimisation step this can be optimised to split the composite service into its atomic services sections and extend the execution strategy with an initial distribution of the sections to each agent, i.e., each execution agent receives just its own task within the composite service. In the scenario above this initial step could be done by SEA_0. To complete this optimisation each agent also needs knowledge about its adjacent predecessor(s) and successor(s) for control flow navigation. The successor(s) is/are

required for normal forward navigation whereas the predecessor(s) is/are required for backwards navigation in case of roll back.

12.4.3 Interaction Model

The execution agent shows a non-proactive behaviour. It must be actively contacted by a client agent that wants to request execution of a composite OWL-S service[6]. In general, there is no limitation about what kind of agent can act as the client agent. However, the overall CASCOM architecture was designed for dynamic planning of composite services by a dedicated planner agent. This planner agent issues a service (together with the actual inputs) to the execution agent after it has finished the planning, triggers execution start, and asynchronously awaits the results. As problems might occur during execution — for instance, a Web Service invocation might fail because it might be temporarily not available — the interaction model also includes a re-planning sub-interaction. In case of a problem the execution agent would temporarily suspend execution and ask the planner agent for planning of a contingency service. As a result, the types of interactions that have to be supported are extended asynchronous request/reply interactions.

Each execution agent publishes a main interaction interface by which all communication with a client agent takes place. This main interface is formalized by an agent ontology that defines the concepts, predicates, and actions to create meaningful ACL message content. Furthermore, an internal interaction interface exists by which execution agents communicate among themselves during execution; the latter one is not described here.

Not just because of the asynchronous invocation model but especially because of the required usage patterns which are beyond the simple request/reply pattern, all interactions between calling agent and execution agent are stateful, thus forming an interaction protocol. Each new invocation of one of the execution agent's methods implicitly creates a new session which lasts until the final result was sent back to the invoking agent — no matter if the result is positive or negative. The state on both sides is encapsulated by finite state automata as provided by the JADE framework. As a recommended starting point for simple request/reply agent conversations FIPA has specified the standard *Achieve Rational Effect* protocol [9]. However, we have specified and implemented an extended version of the protocol, named *Achieve Rational Effect ** protocol, since the original protocol is not sufficient with respect to the requirements of service execution interaction. For example, if some client agent requests the execution of some composite OWL-S service from an execution agent, it is useful even necessary for monitoring and re-planning purposes that the requestor gets notified about execution progress. This notification informs about the position of control flow within the composite service, respectively the effects achieved so far. In short, the *Achieve Rational Ef-*

[6] In fact, both atomic and composite OWL-S services can be issued the same way to an execution agent.

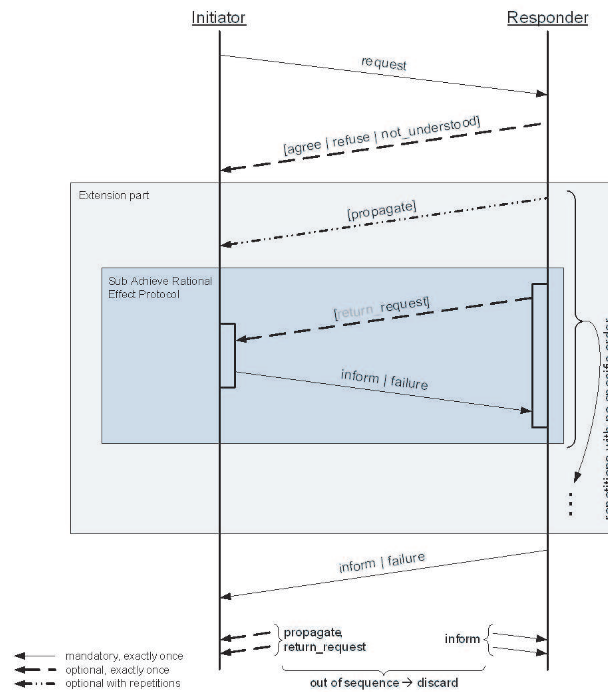

Figure 12.6: Message flow for interactions of a client agent (initiator) with the service execution agent (responder)

*fect ** protocol extends the standard FIPA Achieve Rational Effect protocol with two optional features:

1. The possibility to send any kind of intermediate or feedback ACL messages (information) to the initiator before the final result (inform or failure) is sent to the initiator. Consecutive messages of this kind can be sent, but may alternate with 2.).

2. The possibility to send return requests (ACL messages) back to the initiator to *ask* for additional information which might be required to achieve the original rational effect. A return request must be answered by the initiator until a new return request can be done or a new feedback message can be sent.

Figure 12.6 shows the defined order and cardinalities of the ACL message flow for the *Achieve Rational Effect ** protocol. For the interfacing with the execution agent the extension part is used to provide the initiating agent with up to date information about the current state of the execution. The second part is used to trigger re-planning of composite OWL-S services in case of problems during

execution, for instance, if an atomic service part of the composite service became suddenly unavailable.

12.4.4 Implementation

The distributed service execution is implemented in Java as autonomous agents that can be deployed within the CASCOM agent platform [11] as well as the JADE platform [3]. The implementation derives from and incorporates concepts of the OSIRIS process management system [17]. This system essentially represents a peer-to-peer message oriented middleware with an integrated component framework. The component framework allows to implement and run custom components each delivering a pre-defined service and able to communicate with other (remote) components. The messaging layer basically realizes the publish/subscribe messaging paradigm as well as addressing of specific receivers. Furthermore, it incorporates advanced concepts such as eager and lazy replication, and freshness properties. On top of the middleware, a distributed while decentralized process execution engine (implemented as components) was developed. This means that several peers may be involved in the execution of a composite service whereby no central co-ordinator is required. Its execution strategy can be compared to the one presented in Section 12.4.2. Finally, the internal design of the OSIRIS system is strictly multithreaded, comparable to the SEDA approach [20].

The OSIRIS system was extended to support execution of Semantic Web Services. All its functionality is represented to the outside by agents, whereby for a minimum deployment just one agent is sufficient — of course, such a deployment in fact would be not distributed. Furthermore, all execution agents in a distributed deployment are equal in their functionality accessible by client agents, which means that all of them are available to be used in the same way, thus presenting a true P2P structure. In fact, all the functionality implemented is encapsulated by one so called agent *Behaviour*. Conceptually, a behaviour represents a task that an agent can carry out. Each agent can be added any number of different behaviours at any time, thus adding it any number of tasks that it can do. Consequently, a great flexibility is achieved with respect to who can implement OWL-S service execution functionality: The behaviour can be added to any agent, thus extending the agent to which it is added with OWL-S service execution functionality.

Another aspect that has been incorporated in the implementation is the distinction between call-by-value and call-by-reference semantics. Execution requests by a client agent can be made in either of those types with respect to both the OWL-S service description itself and the actual input data. Using the call-by-value style means to embed the data value itself in a request[7]. On the other hand, call-by-reference style means to embed only a reference[8] to the data value in a request that must be resolved by the execution agent afterwards.

[7]The data format for both the OWL-S service description and input data is serialised XML.

[8]The implementation allows using URLs to reference (external) data that is available somewhere in the Web.

12.5 Summary

In this chapter an agent based framework has been presented that enables the execution of composite (as well as atomic) Semantic Web Services that are described based on OWL-S and WSDL. From a systems point of view, two comprehensive approaches are provided: (i) a centralized solution where a single Execution Agent solely takes over the responsibility for executing a given service and (ii) a distributed approach where a set of P2P-organized Execution Agents co-operate to share the execution task among them. The centralized solution employs context information to determine the appropriate service providers for each situation to distribute and balance the execution work. The distributed approach goes even one step further by providing built-in scalability support not only with respect to the number of service providers but also in terms of an increasing number of client agents. Here, scalability is supported at two levels: First, at a micro level by a multithreaded implementation that allows an Execution Agent to handle multiple execution requests concurrently. Second, at a macro or infrastructure level by the possibility to dynamically migrate execution from one agent to another. This is beneficial when for instance the overall execution time and/or throughput can be optimized, or when the load among the different execution agents can be balanced. This optimization is based on a generic cost model that is open to incorporate different measures like, for instance, service costs, data and communication costs, reliability of computing resources, and resource consumption of services. This means that the assessment whether an ongoing execution would profit from a transfer to another agent is open to allow (i) incorporating domain-specific requirements and (ii) context information. This gives a great flexibility to apply this approach to various environments having different preconditions, requirements, and properties. Furthermore, the migration procedure does not require any centralized supervision and was developed to happen completely self-dependent and decentralized.

The following two subsections discuss alternative approaches to service execution that can be taken into account when starting from different assumptions.

12.5.1 Late Binding of Service Provider Instance during Execution

The overall CASCOM architecture approch considers that Execution Agents take *instantiated* service descriptions as their input, that is, service descriptions that are already grounded to a particular service provider (instance). This means that in the CASCOM setting the Service Composition Planner Agent already decides about which concrete service provider it will use when creating a new composite service. This approach is highly beneficial for services which are created ad-hoc, and are only very infrequently executed, immediately after the service is composed.

However, when services are executed frequently or when the execution of the service is time-consuming, then it can no longer be guaranteed that the groundings are still valid, while the service type may still be appropriate. In this case, it is

beneficial when planning and service execution is further decoupled. A first approach considers only service types as output of the planner, while the groundings for the complete service is done at instantiation time. This approach is particularly useful for short running services such that the groundings remain valid during service execution. Second, the grounding might be done prior to the invocation of an atomic service during the execution of a composite service. Deferring the decision on the service grounding to instantiation or execution time frees the composition planner agent from this additional task and will reduce the composition time. This shift of service provider selection to happen *closer* to the service invocation has the potential to improve the overall system behaviour in very dynamic service environments. The higher the probability that an optimal service provider selection that was done at time t_1 becomes suboptimal at a later point in time t_2, the more important this becomes. An example that illustrates this would be either a long delay between service composition and execution, or a (very) long running composite service consisting of, say, three subsequent atomic service invocations. If service providers are already selected before execution it might happen that the one associated to the third atomic service is not available anymore when the actual service invocation is due, thus, would raise a failure. The task of binding the abstract service type to an instance will be under the responsibility of the execution agent. To accomplish this binding, additional queries to discover service providers must be done by the execution agent.

However, this aspect is of rather minor importance to the CASCOM system: First, service execution immediately follows service composition, i.e., there is almost no delay in between. Second, the envisioned application scenarios are rather characterized by short running composite services compared to the extent that is considered usual in other application domains like scientific workflows. Finally, the dynamic failure handling by composition re-planning provides a method to cope with services that become unavailable during execution.

Nevertheless, the implementations of both the centralized and the distributed approach are prepared for late binding of service instances during execution but need to be extended to closely integrate with the Service Discovery Agent.

12.5.2 Tight Integration of Service Providers and Execution Agents

The discussion of tight integration versus remote coupling of service providers and execution agents has already been raised and discussed in detail in Section 12.4.2. The CASCOM systems assumes a remote coupling of service providers and execution agents because of the status quo in the way Web Services are deployed today and how they can be accessed – not only from a technical point of view but also from a organizational point of view: Service providers are unwilling to deploy additional software layers that would integrate their services with inter-organizational service infrastructures for several reasons. The most important one is that they are afraid of loosing control over their services.

Still, in intra-organizational deployments it would be more easily possible to

use the approach for distributed execution of composite services that has been described in Section 12.4.2. Even then one aspect remains to be further considered, which is the adequacy for ad hoc automated service composition where the resulting composite services are usually executed only once.

References

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

[2] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith and P. Steggles: Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[3] F. Bellifemine and G. Rimassa: Developing multi-agent systems with a FIPA-compliant agent framework. *Software-Practice and Experience*, 31(2):103–128, 2001.

[4] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana: Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[5] W F. Clocksin and C. S. Mellish: *Programming in Prolog.* Springer-Verlag New York, Inc., New York, NY, USA, 1981.

[6] OWL Services Coalition. OWL-S: Semantic Markup for Web Services, 2003.

[7] P. Costa and L. Botelho: Generic context acquisition and management framework. In *Proceedings of the First European Young Researchers Workshop on Service Oriented Computing*, 2005.

[8] E. Denti, A. Omicini and A. Ricci: Multi-paradigm java-prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[9] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. `http://www.fipa.org/specs/fipa00037`, 2000. Specification number SC00037.

[10] H. Helin, M. Klusch, A. Lopes, A. Fernandez, M. Schumacher, H. Schuldt, F. Bergenti, and A. Kinnunen: Context-aware Business Application Service Co-ordination in Mobile Computing Environments. In *Proceedings of the 2005 Workshop on Ambient Intelligence - Agents for Ubiquitous Environments*, Ultrecht, The Netherlands, July 2005.

[11] H. Helin, T. van Pelt, M. Schumacher and A. Syreeni. Efficient Networking for Pervasive eHealth Applications. In GI-Edition, editor, *Proceedings of the European Conference on EHealth (ECEH06)*, volume P-91 of *Lecture Notes in Informatics*, October 2006.

[12] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean: SWRL: A Semantic Web Rule Language combining OWL and RuleML. `http://www.w3.org/Submission/SWRL`, 2004.

[13] A. Lopes and L.M. Botelho: SEA: a Semantic Web Services Context-aware Execution Agent. In *AAAI Fall Symposium on Agents and the Semantic Web*, Arlington, VA, USA, 2005.

[14] D. McDermott: PDDL – the planning domain definition language, 1998.

[15] M. G. Nanda, S. Chandra and V. Sarkar: Decentralizing execution of composite Web Services. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[16] H. Schuldt, G. Alonso, C. Beeri and H.-J. Schek: Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, March 2002.

[17] C. Schuler, H. Schuldt, C. Türker, R. Weber and H.-J. Schek: Peer-to-peer execution of (transactional) processes. *International Journal of Cooperative Information Systems (IJCIS)*, 4(14):377–405, 2005.

[18] C. Schuler, R. Weber, H. Schuldt and H.-J. Schek: Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In *Proceedings of the 2nd International Conference on Web Services (ICWS)*, pages 26–34, San Diego, CA, USA, July 2004. IEEE Computer Society.

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

[20] M. Welsh, D. E. Culler and E. A. Brewer: SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles (SOSP-18)*, pages 230–243, Banff, Canada, 2001.