D2.4.10 Architecture and Execution Semantics for the SWS

Coordinator: Tomas Vitvar
(National University of Ireland, Galway)

with contributions from:

Matthew Moran, Maciej Zaremba (National University of Ireland, Galway),

Adrian Mocan, Mick Kerrigan, and Thomas Hasselwanter (University of Innsbruck, Austria)

Abstract.
EU-IST Network of Excellence IST-2004-507482 Deliverable D2.4.10 version 1

The goal of this deliverable is to design the Semantic Web Services Architecture and to establish grounds for joint work on the Semantic Service Oriented Architecture involving various groups. In this work we define the architecture from several viewpoints allowing to clarify different architecture aspects, its services, processes and technology.

Keyword list: Web Services, Service Oriented Architecture, Semantic Web

Document Identifier	KWEB/2006/D2.4.10/v1
Project	KWEB EU-IST-2004-507482
Version	v1.0
Date	December 30, 2006
State	final
Distribution	public

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) - Coordinator
Institute of Computer Science
Technikerstrasse 13
A-6020 Innsbruck
Austria
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polytechnique Fédérale de Lausanne (EPFL)
Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne
Switzerland
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)
4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)
Takustrasse 9
14195 Berlin
Germany
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)
Piazza Domenicani 3
39100 Bolzano
Italy
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)
ZIRST - 655 avenue de l'Europe -
Montbonnot Saint Martin
38334 Saint-Ismier
France
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas / Informatics and Telematics Institute (ITI-CERTH)
1st km Thermi - Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
30539 Hannover
Germany
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)
National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Contact person: Tomas Vitvar
E-mail address: tomas.vitvar@deri.ie

The Open University (OU)
Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo sn
28660 Boadilla del Monte
Spain
Contact person: Asunción Gómez Pérez
E-mail address: asun@fi.upm.es

University of Karlsruhe (UKARL)
Institut für Angewandte Informatik und Formale
Beschreibungsverfahren - AIFB
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Contact person: Rudi Studer
E-mail address: studer@aifb.uni-karlsruhe.de

University of Liverpool (UniLiv)
Chadwick Building, Peach Street
L697ZF Liverpool
United Kingdom
Contact person: Michael Wooldridge
E-mail address: M.J.Wooldridge@csc.liv.ac.uk

University of Manchester (UoM)
Room 2.32. Kilburn Building, Department of Computer
Science, University of Manchester, Oxford Road
Manchester, M13 9PL
United Kingdom
Contact person: Carole Goble
E-mail address: carole@cs.man.ac.uk

University of Sheffield (USFD)
Regent Court, 211 Portobello street
S14DP Sheffield
United Kingdom
Contact person: Hamish Cunningham
E-mail address: hamish@dcs.shef.ac.uk

University of Trento (UniTn)
Via Sommarive 14
38050 Trento
Italy
Contact person: Fausto Giunchiglia
E-mail address: fausto@dit.unitn.it

Vrije Universiteit Amsterdam (VUA)
De Boelelaan 1081a
1081HV. Amsterdam
The Netherlands
Contact person: Frank van Harmelen
E-mail address: Frank.van.Harmelen@cs.vu.nl

Vrije Universiteit Brussel (VUB)
Pleinlaan 2, Building G10
1050 Brussels
Belgium
Contact person: Robert Meersman
E-mail address: robert.meersman@vub.ac.be

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to writing parts of this document:

École Polytechnique Fédérale de Lausanne
France Telecom
Freie Universität Berlin
National University of Ireland Galway
University of Innsbruck
University of Liverpool
University of Manchester
University of Trento

Changes

Version	Date	Author	Changes
0.2	20.12.06	Tomas Vitvar	First version.
0.3	10.01.07	Tomas Vitvar	Comments from reviews implemented
1.0	08.02.07	Tomas Vitvar	Final changes and alignments

Executive Summary

The architecture for the Semantic Web Services is overarching the work done within the WP2.4 over the duration of the Knowledge Web project. The main goal is to provide a framework which would allow integration of various functionality required for services provisioning while at the same time promoting goal-based invocation of web services which are semantically described. In this deliverable we aim to find the consensus of various research working on the architecture for the Semantic Web Services and establish the solid grounds for joint collaboration within the OASIS Semantic Execution Environment Technical Committee. The work in this deliverable thus reflect the first design stage of the architecture and contains additional concepts build on the top of architectures for Semantic Web Services or Semantically Oriented Architectures from other EU funded projects. In this deliverable we define a number of perspectives through which the architecture is described, namely *global view* identifying a number of layers from the global viewpoint on the architecture, *service view* identifying various types of services and describing these services in detail, *process view* describing processes which are both provided as well as facilitated by the architecture, and *technology view* revealing details of the technology used for implementation of the architecture and its middleware system in particular. The work on the architecture is the continuous and incremental process which involves various aspects specific for each group and project where the architecture is being developed. The work in this deliverable aims to establish grounds which will allows to add additional concepts and functionality to the architecture in the future.

Contents

Chapter 1

Introduction

With regard to new emerging trends in enterprise computing, the adoption of Service Oriented Architectures (SOA) is starting to be of interest to industry. With the goal of enabling dynamics and adaptivity of business processes, SOA builds a service-level view on organizations conforming to principles of well-defined and loosely coupled services services which are reusable, discoverable and composable. Although the idea of SOA targets the need for integration that is more adaptive to changes in business requirements, existing SOA solutions will prove difficult to scale without a proper degree of automation. In addition, todays SOA technologies only provide partial solution to interoperability, mainly by means of unified technological environments. A major characteristic of Web service technology, such as WSDL, SOAP, UDDI, and BPEL, is to provide interoperable solutions at the technological level, while a generic and scalable solution for content and process level interoperability is still in its infancy. Where content-level interoperability is to be solved, ad-hoc solutions are often hard-wired in business processes using traditional XSLT approaches. Process level interoperability is often maintained through manual configuration of workflows which makes interoperability a manual task. In order to address these drawbacks, the extension of SOA with semantics offers a scalable integration, more adaptive to changes that might occur over a software systems lifetime. Semantics for SOA allow the definition of semantically rich and formal service models and ontologies and, by means of logical reasoning, promotes a total or partial automation of service discovery, contracting, negotiation, mediation, composition and invocation. Rather than replacing existing approaches, semantic SOA must build on existing industry standards and technologies being used in industry within existing enterprise infrastructures.

1.1 Goal of the Deliverable

The goal of this deliverables is to provide a conceptual and logical/detail design of the architecture for the semantic web services forming Semantic Service Oriented Architecture (SESOA). This work is compliant with requirements for web service description as described in deliverable D2.4.1 while at the same time it is in line with the conceptual

and formal framework for the Semantic Web Services as described in deliverable D2.4.5. The design of the architecture also integrates some work done in WP2.4, i.e. discovery, interoperation, invocation and mediation of web services (deliverables D2.4.2, D2.4.7, D2.4.12). This work has also been done with respect to selected use case of the WP2.4 from the SWS Challenge (deliverable D2.4.13) (Data and Process Mediation of Services in Enterprise Application Integration).

For the design of the architecture in the context of the Knowledge Web project we follow the standard software engineering approach to development of computer-based and information systems. Thus, we conform to phases of conceptual analysis and design, logical/detail design, implementation, testing, and deployment. The work on the architecture in this deliverable falls into the phases of conceptual and logical/detail design which follows a conceptual analysis from deliverables D2.4.1 and D2.4.5 (i.e. requirements analysis, conceptual framework for Semantic Web Services). Additional phases are partially covered within the SWS challenge efforts of WP2.4.

1.2 Overview of the Deliverable

The architecture is described in the next section 2. We describe the architecture from several perspectives, namely global view, service view, process view and technology view, within each we elaborate on particular details of services, processes, and technology used for implementation of the architecture and its middleware system. In section 3 we summarize the work and outline our future work.

1.3 Semantic Web Services

Web services add a new level of functionality to the Web, stepping towards an open environment of distributed and heterogeneous applications. While current Web service technologies around SOAP, WSDL and UDDI have established the potential of a Web of services, they are hindered by dependence on XML-only descriptions. Although flexible and extensible, XML can only define the structure and syntax of data. Without machine-understandable semantics, services must be located and bound to service requesters at design-time, limiting possibilities for automation. In order to address this drawback, several initiatives exist (WSMO, OWL-S, WSDL-S)[9, 6, 10] to define a new layer on top of the current Web service stack based on semantic mark-up for functional, non-functional and behavioral aspects of service descriptions. The ultimate goal is to enable total or partial automation of service discovery, composition, mediation, invocation, etc.

1.3.1 Web Service Modeling Ontology

A general aim of Semantic Web Services is to define a semantic mark-up for Web services providing the higher expressivity then traditional XML-based descriptions. One of the initiatives in the area is the Web Service Modeling Ontology (WSMO)[11]. WSMO provides a conceptual model describing all relevant aspects of Web services in order to facilitate the automation of service discovery, composition and invocation. The description of WSMO elements is represented using the Web Service Modeling Language (WSML)[11] a family of ontology languages which consists of a number of variants based on different logical formalisms and different levels of logical expressiveness. WSMO also defines the conceptual model for WSMX[8], a Semantic Web Services execution environment. Thus, WSMO, WSML and WSMX form a coherent framework for modeling, describing and executing Semantic Web Services.

WSMO Model

The WSMO top-level conceptual model consists of *Ontologies*, *Web Services*, *Goals*, and *Mediators*.

Ontologies provide the formal definition of the information model for all aspects of WSMO. Two key distinguishing features of ontologies are, the principle of a shared conceptualization and, a formal semantics (defined by WSML in this case). A shared conceptualization is one means of enabling information interoperability across independent Goal and Web service descriptions.

Web Services are defined by the functional capability they offer and one or more interfaces that enable a client of the service to access that capability. The Capability is modeled using preconditions and assumptions, to define the state of the information space and the world outside that space before execution, and postconditions and effects, defining those states after execution. Interfaces are divided into *choreography* and *orchestration*. The choreography defines how to interact with the service while the orchestration defines the decomposition of its capability in terms of other services.

Goals provide the description of objectives a service requester (user) wants to achieve. WSMO goals are described in terms of desired information as well as "state of the world" which must result from the execution of a given service. The WSMO goal is characterized by a requested capability and a requested interface.

Mediators describe elements that aim to overcome structural, semantic or conceptual mismatches that appear between different components within a WSMO environment. Although WSMO Mediators are essential for addressing the requirement of loosely coupled and heterogeneous services, they are out of the scope of our work at this point.

Chapter 2

Architecture: Concepts and Technology

The architecture for the Semantic Web Services is designed to operate on semantic descriptions of various elements of the conceptual model provided by the Web Services Modeling Ontology (WSMO) and described using Web Service Modeling Language (WSML). The design of the architecture is governed by several principles which underpin various components of the architecture . In this section we identify these governing principles and define the architecture from several perspectives, namely *global*, *services*, *processes* and *technology*. Within these views, we identify and describe in detail service and process types which are provided and facilitated by the architecture as well as technology used for building the architecture, its middleware and service infrastructure.

The information provided in this section is relevant to the first phase of the architecture design. The design of the architecture for the Semantic Web Services is done incrementally and within the first phase our aim is to establish the framework for the architecture, its basic services, their interfaces and processes. We discuss this functionality within the following sub sections.

2.1 Governing Principles

Following principles are the main drivers for the overall architecture design. They reflect fundamental requirements for user-centric, service oriented and distributed environment which all together facilitate seamless provisioning of business services.

Service Oriented Principle. Service-orientation represents a distinct approach for analysis, design, and implementation which further introduces particular principles that govern aspects of communication, architecture, and processing logic. This includes service reusability, loose coupling, abstraction, composability, autonomy, statelessness, and discoverability. With respect to the service orientation which enables a service level view on the organization we further distinguish services from several views.

First, we distinguish two types of services from the point of the functionality they pro-

vide within the architecture, namely (1) *Business Services* and (2) *Middleware Services*. Business services are services provided by various service providers, their back-end systems – business services are subject of integration and interoperation within the architecture and can provide a certain value for users (e.g. purchasing a flight). On the other hand, middleware services are the main facilitators for integration and interoperation of business services (e.g. discovery, interoperability, etc.).

Second, we distinguish two types of services from the point of their abstraction in the architecture, namely (1) Web Services, and (2) Services. The Web Service is a general service which might take several forms when it is instantiated (e.g. purchase a flight) whereas the Service is actual instance of the Web Service which is consumed by a user and which provides a concrete value for a user (e.g. purchase a flight from Prague to Bratislava). We use this distinction for Business Services in the architecture.

Semantic Principle. Semantics in general is considered as a rich and formal description of information and behavioral models enabling automation of certain tasks by means of logical reasoning. Combined with service oriented principle, semantics allows to define scalable, semantically rich and formal service models and ontologies allowing to promote total or partial automation of tasks such as service discovery, contracting, negotiation, mediation, composition, invocation, etc. Semantic service oriented approach to modeling and implementation of the organization enables scalable and seamless interoperation, reusability, discovery, composition, etc. of various Business Services.

Distributed Principle. Distributed principle is the process of aggregating the power of several computing entities to collaboratively run a single computational task in a transparent and coherent way, so that they can appear as a single and centralized system. Distributed principle is applied to the architecture middleware system which allows to distribute its components over the network in a transparent way so that the execution process run in the middleware could be scaled across a number of physical servers over the network. Distributed principle is also applied to the Business Services allowing to run a process spanning across several Business Services distributed over the network.

User-centric Principle. The user-centric principle puts the user in the center of the architecture. It refers to concepts like personalizing of Business Services, facilitating service usability, promoting multi-channel access and service delivery, building trust, achieving efficiency, accountability and responsiveness according to users' requirements, enabling seamless implementation of Business Processes across organizational boundaries, etc. The architecture establishes the infrastructure which by means of its services and processes promotes the user centric principle in service provisioning.

2.2 Global View

The global view on the architecture, depicted in figure 2.1, comprises of several layers, namely (1) *Stakeholders* forming several groups of users of the architecture, (2) *Service*

Requesters as client systems of the architecture, (3) *Middleware* providing the intelligence for the integration and interoperation of Business Services, and (4) *Service Providers* exposing the functionality of back-end systems as Business Services.

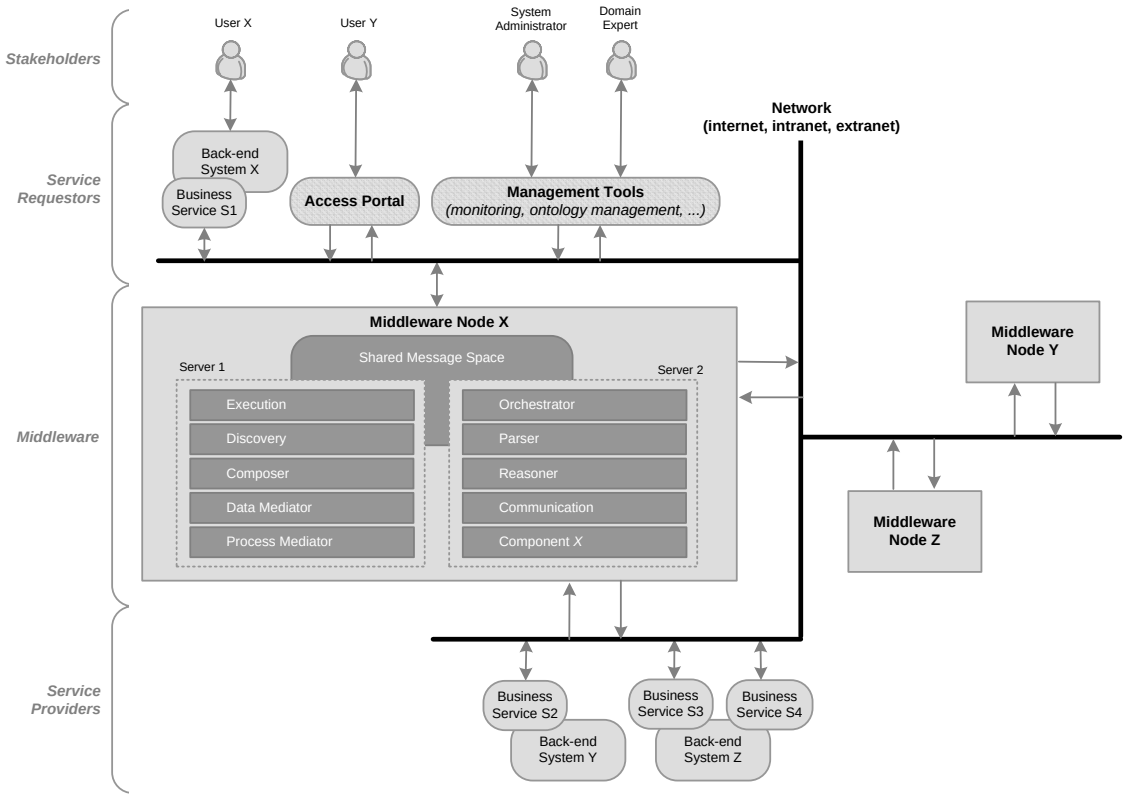

Figure 2.1: Global View

Stakeholders form the group of various users which use the functionality of the architecture for various purposes. Two basic groups of stakeholders are identified: (1) *users*, and (2) *administrators*. Users form the group of those stakeholders to which the architecture provides end-user functionality through *access portals* or through *back-end systems*. For example, users can perform electronic exchange of information to acquire or provide products or services, to place or receive orders or to perform financial transactions. Such operations can be performed through back-end systems, e.g. Enterprise Resource Planning (ERP) integrated with external suppliers (B2B integration), or through various access portals such as e-marketplaces. In general, the goal is to allow users to interact with business processes on-line while at the same time reduce their physical interactions with back-office administrations. On the other hand, the group of administrators form those stakeholders which perform administrative tasks in the architecture. These tasks should support the whole SOA lifecycle including service modeling, creation (assembling), deployment (publishing), and management. Different types of administrators could be involved in this process ranging from domain experts (modeling, creation) to system administrators (deployment, management). In general, all groups perform certain

activities by triggering various *middleware processes* provided by the architecture.

Service Requesters are client systems in the architecture. On one side they provide interface for stakeholders and on the other they are integrated with the middleware through various specialized API. Service requesters include *access portals*, i.e. client applications through which users directly interact with the architecture in order to achieve certain goal, *back-end systems*, i.e. organizational systems, such as Enterprise Resource Planning (ERP) allowing integration of organizational systems through the architecture, and *administration tools* through which administrators model, create, deploy and manage the architecture, its services and processes.

Administration tools provide a specific functionality for domain experts and system administrators. This functionality cover the whole SOA lifecycle including service modeling, creation (assembling), deployment (publishing), and management. In our Semantic Web Services settings based on the WSMO model and WSML language, this functionality is provided as part of the Web Service Modeling Toolkit (WSMT)[1]. WSMT includes the ontology management tools for service and ontology modeling, creation and deployment (e.g. ontology and service editor, ontology and service visualizer, and ontology mapping tool). For management, WSMT allows configuration of the middleware and monitoring of processes run in the middleware.

On the other hand, access portals and back-end systems provide a specialized functionality for architecture stakeholders, namely end-users. They provide a specialized domain specific user interfaces and application functionality through which stakeholders interact with the architecture and its processes. According to some particular deployment of the architecture to the particular environment, this functionality can guide user in providing and getting input and output data respectively during processing as well as provide interactions with the middleware at various levels. From end-user perspectives, such functionality however hides back-end distributed processing within the architecture, its middleware and services. A specialized end-user functionality is subject to design and development in application oriented projects, such as SemanticGov[2]. In this project we develop a specialized functionality for clients to interact with public administration processes facilitated by the middleware system.

Middleware is the core of the architecture providing the main intelligence to integration and interoperation of Business Services. The middleware system consists of a number of components (middleware services) where each component provides a certain functionality within an execution process. Each component exposes its functionality through a number of interfaces, thus the functionality of the component could be consumed by other components through these interfaces. Components could form this way a number of processes called *middleware processes* which could be executed in the middleware. These processes facilitate the business service provisioning and are defined by so called *execution semantics* of the middleware system. A number of execution semantics can

[1] http://wsmt.sourceforge.net
[2] http://www.semantic-gov.org

exist in the middleware system which form the overall behavior of the architecture. Although the design of the middleware system is open, in section 2.3 we define a number execution semantics which specifically target the need for integration and interoperation of the Semantic Web Services.

The components represent middleware services including services for managing the execution (execution semantics, messaging), discovery, selection, data and process mediation, resource management through repositories etc. In addition, the middleware can operate in distributed manner on a number of physical servers connected using a shared message space. Shared spaces provide a messaging abstraction for distributed architecture which reflects requirements of distributed principle and in addition empowers the scalability of the integration process. Moreover, middleware systems can be connected in a way where each instance of the middleware represents a node in a network of middleware systems. Each node of the middleware system in the architecture can provide certain functionality for certain purpose. For example, nodes operating only repository services can build up a distributed repository system connected over P2P network with other nodes on the network. Another example is the interoperability gateway for various "regions" operating on domain semantics which need to interoperate. Interoperability gateway, run as a node in the architecture, can be configured to provide interoperability services for all peers on the network. Such approach can be applied for example to implementation of so called Interoperability Clearinghouse which in the European Union facilitates the interoperation of so called Pan-European E-government Services (PEGS) – services which are defined and executed across a number of EU Member State Administrations. The design of the architecture for interoperation of various Public Administrations in the EU based on the semantic technologies is the subject of our other work in the European Project SemanticGov[3].

Service Providers are various back-end systems. Unlike back-end systems in service requesters layer which act as clients in client-server setting of the architecture, the back-end systems in service providers layer act as servers which provide certain functionality for certain purpose exposed as a business service to the architecture. Depending on particular architecture deployment and integration scenarios, the back-end systems could originate from one organization (one service provider) or multiple organizations (more service providers) interconnected over the network (internet, intranet or extranet). The architecture thus can serve various requirements for Business to Business (B2B), Enterprise Application Integration (EAI) or Application to Application (A2A) integration. In all cases, functionality of back-end systems is exposed as business services which in addition are semantically described.

[3] http://www.semantic-gov.org

2.3 Service View

Services provide certain functionality for certain purpose. In the architecture and with respect to service oriented principle, we distinguish two types of services, namely (1) *Middleware Services*, and (2) *Business Services*. The middleware and business services are the main enablers of a processing logic at both middleware and business levels. In our Semantic Web Services settings, the business services are modeled using WSMO model and WSML language and their descriptions together with the middleware services are deployed to the Web Services Execution Environment (WSMX)[4].

2.3.1 Middleware Services

As described in the architecture global view in section 2.2, middleware services are represented as components in the middleware system. Although our approach to design of the middleware system is open, we define a number of components built specifically to enact semantic web services. In figure 2.2, these components are depicted together with their interfaces. We further define two major groups of the middleware services, namely (1) *Basic Services*, and (2) *Application Services*. Basic services provide the fundamental functionality of the middleware such as reasoning, communication and control of the execution process. Application services are services providing particular specific functionality for particular middleware process such as discovery, selection, choreography, mediation, etc. In this section we define the middleware services relevant to the first phase of the architecture design. In this phase we do not consider composition and orchestration of composite services as it adds additional complexity to execution semantics of the middleware. Composition and orchestration is the subject of future design of the architecture and will be built on top of the functionality available from the first phase of the architecture design.

Basic Services

- **Communication.** The Communication facilitates inbound and outbound communication with the middleware system. In other words, any message sent to or sent from the middleware system is passed through this component. The Communication thus implements a number of external interfaces through which the functionality of the whole middleware system can be consumed. Through invocation of such external interface, the execution process is triggered in the middleware system or it is possible to step into the already running execution process in the middleware system (which facilitates asynchronous interactions with the middleware system).

  Since the middleware system is meant to support the integration of semantic web services, messages which are being handled within execution processes at the mid-

[4] http://www.wsmx.org/

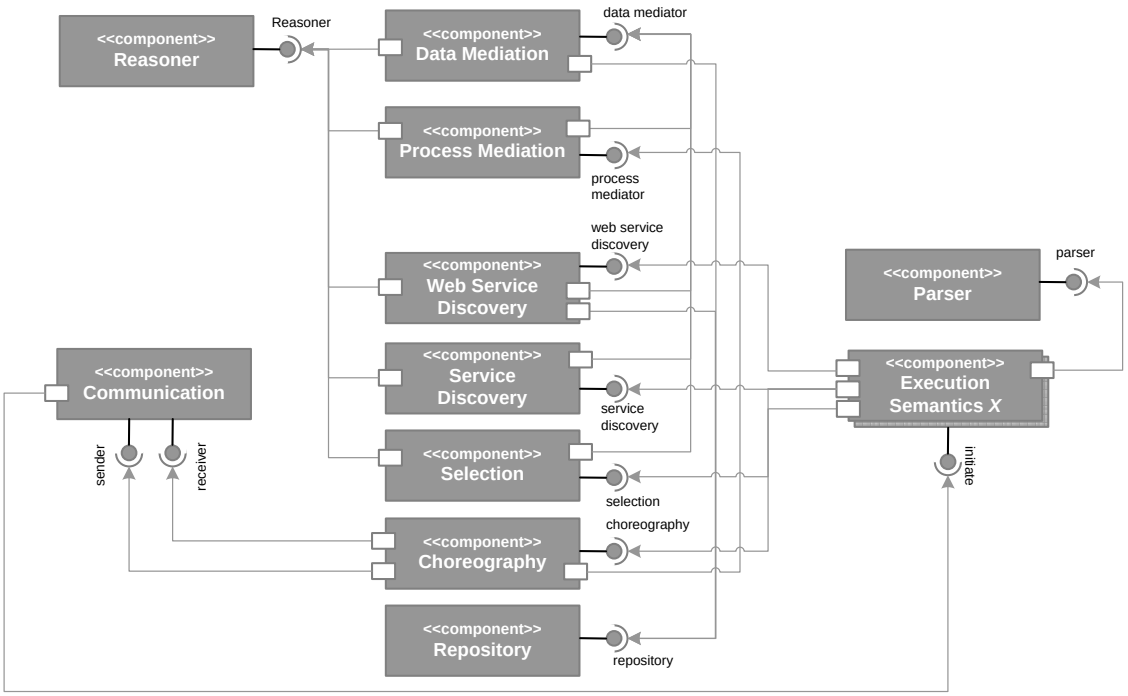

Figure 2.2: Service View – Middleware

dleware system are messages conveying semantic descriptions of data (according to the WSMO model). On the other hand, the mechanism used for invocation of services is based on SOAP and WSDL specifications. Thus, the communication component also implements mechanisms for *grounding* of semantic WSMO level and physical invocation level (see next section 2.3.2).

- **Parser.** The parser component provides the parsing of semantic messages into the object model as defined by WSMO4J[5]. Since all messages are semantic messages captured in WSMO and WSML, the parser component operates on WSMO4J library. All messages are then physically handled within the middleware system as WSMO4J objects.

- **Reasoner.** The Reasoner component provides the reasoning functionality over semantics in description of messages. Reasoning is important functionality required during various execution processes and is used by most of the components such as discovery, data mediation, process mediation, etc. Different requirements apply to reasoning which is based on the variant of the WSML language used for semantic descriptions. Description Logic (DL) based reasoner is used when DL-based variant of WSML is used, Datalog or F-Logic based reasoner is used when WSML-Flight or WSML-Rule variant is used respectively. In general, various reasoners complying to the interface of the resoner component can be used. The use of particular

[5]wsmo4j.sourceforge.net

reasoner depends on requirements for semantic descriptions required for description of services and ontologies and can be dependent on particular requirements of a specific case scenario. Development of reasoners for WSML is ongoing work in the WSML WG[6].

- **Execution Semantics.** Execution semantics controls the interactions of various components which servers particular middleware process for specific purpose. Each execution process is started by invocation of particular external interface (implemented by the communication component) and can be interfaced through other external interfaces during execution allowing asynchronous communication with the middleware. A number of execution semantics can exist in the middleware which can facilitate the design-time processes (modeling, creation, deployment and management) such as getting/storing entity from/to repository and run-time such as conversation with data and process mediation applied where necessary. More details about execution semantics and processes are described in the section 2.4.

- **Repository.** Repository manages the storages of various entities of the middleware including goals, services, ontologies and mediators (mapping rules). All these entities are described using WSML semantic language. The storage mechanism used for storing the entities is the Triple Store mechanism.

Application Services

- **Data Mediation.** Data mediator facilitates run-time mediation during execution process when different ontologies are used in service descriptions involved in the process. Data mediation can be applied during discovery between service requester's goal and potential services which satisfy the goal or during conversation between service requester and service providers when description of services' interfaces can use different ontologies. Such data mediation operates on mapping rules between ontologies which must be published to the architecture before the mediation can happen. These mapping rules are created using design-time data mediation tool which is part of the ontology management tools. Detail description of data mediation for the semantic web services can be found in [7].

- **Process Mediation.** Process mediator facilitates the run-time mediation when different choreography interfaces are used in service descriptions involved in the conversation. Process mediation is applied together with choreography, data mediation, and communication components when service requester and service provider communicate (exchange messages). In addition, process mediation can be used also during discovery for evaluation whether conversation between service requester and potential service provider is possible (in other words if process mediation can be fulfilled). By analysis of choreography descriptions, process mediator decides

[6] http://www.wsmo.org/

to which party the data in a received message belongs – service requester, service providers or both. Through this analysis, the process mediator resolves possible choreography conflicts including stopping a message when the message is not needed for any party, swapping the sequence of messages where messages are to be exchanged in different order by both parties, etc. More information about conceptual definition of process mediator and choreography conflicts can be found in [1].

- **Web Service Discovery.** Web Service discovery is a process of finding services satisfying requesters needs. At this stage, services are matched at abstract level taking into account capability descriptions of services. Several set-theoretical relationships exist between these description such as exact match, plug-in match, subsumption match, intersection match, and disjnontness [4].

- **Service Discovery.** Service discovery is a process of finding concrete services satisfying concreate goals of users. At this stage, services which match at abstract level are matched at instance-level when additional information might be retrieved from the service provider. Such information (e.g. price or product availability) usually has a dynamic character and is not suitable for static capability or ontology descriptions. For this purpose so called meta-interactions within the execution process and service providers might take place in order to retrieve this information through specialized service interfaces. More information about interactions between service providers, service requesters and middleware can be found in the next section 2.4.

- **Selection.** Selection is a process where one service which best satisfies the user preferences is selected from candidate services returned from the service discovery stage. As a selection criteria, various non-functional properties such as Service Level Agreements (SLA), Quality of Services (QoS), etc. can be used expressed as user preferences – non-functional properties of the goal description. Such non-functional descriptions can capture constraints over the functional and behavioral service descriptions. Selection can thus restrict the consumption of service functionality by a specific condition, e.g. quality of service preference may restrict the usage of a service when its satisfiable quality is provided.

- **Choreography.** Choreography is the core component which drives the run-time conversation between service requester and service providers. This step involves the interactions with process mediator (together with data mediator) as well as communication component called each time the message exchange needs to happen between them. This process is described in a little bit more detail in the next section 2.4. The requester-provider conversation is the elementary step within the orchestration of the composite service. At this stage of architecture development, we however do not deal with composite services and their orchestrations. Such conversations add additional level of complexity to the middleware processes will be the subject of design in subsequent versions of the architecture.

2.3.2 Business Services

Business services contain a specific functionality of back-end systems which descriptions conform to WSMO Service specification. Description of business services is exposed to the architecture (these descriptions are published to the middleware repositories) and are handled during execution processes in the middleware in both design-time (service creation) and run-time processes (conversation). The important aspect of service creation from the semantic web services architecture point of view is *semantic modeling of business services*. This process can be seen at following levels (see figure 2.3), namely (1) *conceptual*, (2) *logical*, and (3) *physical*. With respect to the modeling levels, we further distinguish two modeling approaches to semantic business services, namely *top-down* and *bottom-up* approach.

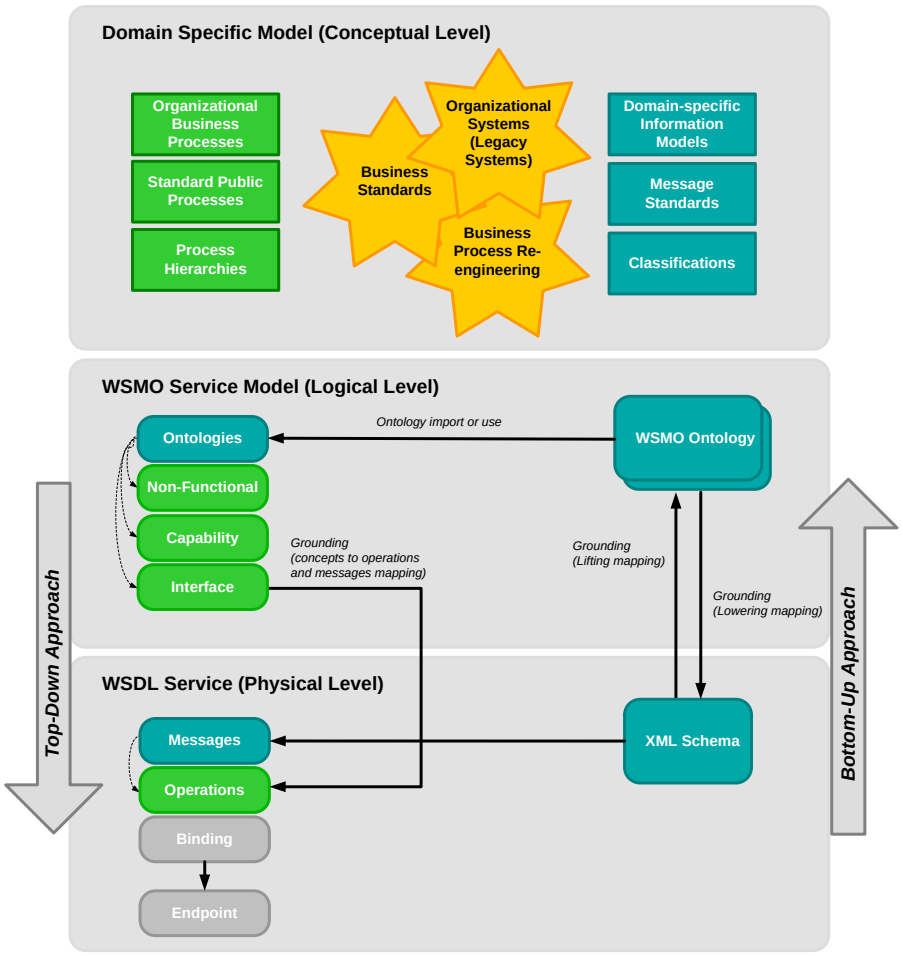

Figure 2.3: Semantic Business Service Modeling Levels

Modeling Levels

- **Conceptual Level.** Conceptual level contains all domain specific information which is relevant for modeling of business services. This information covers various domain-specific information such as database schemata, organizational message standards, standards such as B2B standards (e.g. RosettaNet Partner Interface Processes (PIP) messages[7]), or various classifications such a NAICS[8] (The North American Industry Classification System) for classification of a business or industrial units. In addition, the specification of organizational business processes, standard public process such as RosettaNet PIP processes specifications, and various organizational process hierarchies are used for modeling of business processes. All such information is gained from re-engineering of business processes in the organization, existing standards used by organizational systems or existing specifications of organizational systems (e.g. Enterprise Resource Planning systems).

- **Logical Level.** Logical level represents the semantic model for business services used in various stages of execution process run on middleware. For this purpose we use WSMO service model together with WSML semantic language. WSMO defines service semantics including non-functional properties, functional properties and interfaces (behavioral definition) as well as ontologies that define the information models on which services operate. In addition, grounding from semantic descriptions to underlying WSDL and XML Schema definitions must be defined in order to perform invocation of services. Semantic services described using WSMO follow the paradigm of service decoupling with strong mediation among them. This means that services can be described independently allowing the service semantics to be used for (semi) automated service integration (handling interoperability issues with data and process mediation as well as discovery, selection, etc.).

- **Physical Level.** Physical level represents the physical environment used for service invocation. In our architecture, we use WSDL and SOAP specification. For this purpose, the grounding must be defined between semantic descriptions and WSDL descriptions of services. Definition of such grounding can be placed to WSMO descriptions at the WSMO service interface level or WSDL descriptions using the recent Semantic Annotations for WSDL (SAWSDL) approach[9]. The definition of grounding is dependant on the modeling approach and is discussed in following paragraph.

[7] http://www.rosettanet.org
[8] http://www.naics.com
[9] www.w3.org/2002/ws/sawsdl/

Modeling Approaches and Grounding Definitions

A semantic business service is modeled using WSMO service model and all relevant domain-specific information. As a result, all WSMO service description according to the WSMO service model and all relevant ontologies (used by the WSMO service) are defined. A domain expert can reuse already existing domain ontologies or create the new ontologies based on the information he/she gets from the domain models (databases, standards, etc.). Similarly, the WSMO services is modeled based on domain-specific requirements, specifications of back-end systems etc. The important aspect of the modelling phase is to define grounding from the semantic WSMO service (logical level) to the underlying WSDL description (physical level). This grounding takes two forms in WSMO: (1) grounding defined at the level of WSMO service interface, and (2) grounding defined at the level of ontologies and XML Schemata. The former specifies a reference for each used concept in the interface definition to the input or output messages used in the WSDL. The latter specifies lifting and lowering schema mapping for XML Schema and ontology respectively in order to perform instance transformations during invocation.

Top-Down Approach. In the top-down approach, the underlying representation of the service in WSDL does not exist up-front and thus needs to be created (and service implemented) as part of business service creation/modeling phase. For the first type of grounding, references of used concepts of the service interface are defined to the newly created WSDL operations, its input and output messages. The definition of the second type of grounding is then placed to the implementation of the service itself. That means, that semantic messages passed from the middleware to the service during invocation are serialized to the RDF/XML (WSML can be represented in RDF[10]) and passed to the service where the lowering must be performed. Inversely, the lifting is performed in the service to the ontology (represented in RDF/XML) and passed to the middleware. In the middleware, RDF/XML is transformed to WSML (in the Communication component) and other processing follows according to the execution semantics definition. More information about WSMO grounding as described in this paragraph can be found at [5].

Bottom-Up Approach. In the bottom-up approach, the underlying representation of the service in WSDL already exist (together with the implementation of the service) and thus needs to be taken into account during business service modeling. The grounding definition at the service interface is defined the same way as in the top-down approach. However, the difference exist for the second type of the grounding definition. Since it is not possible to modify the implementation of the service, the schema mapping must be performed and defined externally from the WSDL and service implementation. The schema mapping is thus attached to the WSDL descriptions using SAWSDL[11] specifications (using *loweringSchemaMapping* and *liftingSchemaMapping* extension attributes). The location of these mappings is resolved by the Communication component and executed during the invocation process. On result, the XML schema created from lowering is passed to the

[10] for details see http://www.wsmo.org/TR/d16

[11] Semantic Annotations for WSDL - see http://www.w3.org/ws/sawsdl

service endpoint according to the grounding definition of the service interface. Inversely, created instances of the ontology from lifting is used for data for subsequent execution within the middleware. At the time of writing this article, the WSMO grounding specification using SAWSDL for bottom-up modeling approach is the ongoing work within the WSMO WG.

2.4 Process View

Processes reflect the behavior of the architecture through which stakeholders interact with the middleware and with business services. Similarly as in section 2.3, we distinguish two types of processes, namely (1) *middleware processes* and (2) *business processes*.

2.4.1 Business Processes

Business processes are actual processes provided by the architecture and facilitated by the middleware in concrete business settings. The primary aim of the architecture is to facilitate so called *late-binding* of business services (which results in business processes) and provide the functionality for conversation between business services within a particular business process with data and process mediation applied where necessary. In section 2.4.2, the late-binding and conversation phases is described in detail where only one service is involved in a business process (at this stage there is no composition involved during late-binding phase). As a follow up work when service composition will be included in the late-binding phase, the business process with more business services will be supported by the architecture.

2.4.2 Middleware Processes

Middleware processes are designed to facilitate the integration of business services using middleware services including service discovery, mediation, selection, etc. Middleware processes are described by a set of execution semantics. As described in previous sections, execution semantics defines interactions of various middleware services which servers particular middleware process for specific purpose and on the top provides a particular functionality in a form of business processes to architecture stakeholders. Each middleware process is started by invocation of particular external interface (implemented by the communication component) and can be interfaced through other external interfaces during execution. A number of execution semantics can exist in the middleware which can facilitate the design-time and run-time processes.

For purposes of describing various forms of Execution Semantics, we distinguish two phases of the middleware process, namely *Late-binding Phase*, and *Conversation Phase*. Late-binding phase allows to bind service requester (represented by a goal definition) and

a service provider (represented by a service definition) by means of intelligence of middleware using reasoning mechanisms and with process and data mismatches resolved during binding. In general, late-binding performs a binding of a goal and a service(s) (which includes web service and service discovery, mediation, selection, etc.). Conversation phase allows to perform conversation between previously binded goal and a service by processing of their interfaces and with data and process mediation applied where necessary. Both late-binding and conversation phases follow strong decoupling principle when services are described semantically and independently from requester's goal.

External Integration with the Middleware

From the point of integration of service requesters and service providers with the middleware, we define following aspects for external integration with the middleware.

- **Communication**. The interactions between service requesters and the middleware or the middleware and service providers and vice-versa can happen (1) *synchronously* or (2) *asynchronously*. During synchronous communication, all data is sent in one session when the result/response is sent within the same session. During asynchronous communication, the data is sent in one session whereas the response is sent back in other (newly created) session. Asynchronous communication also allows multiple interactions with the middleware/service requester or provider can happen over time for which one session does not need to be allocated.
- **Entrypoints**. There are two types of entrypoints which can be implemented by the middleware for external communication, (1) *execution entrypoint* entrypoint, and (2) *data entrypoint*. The execution entrypoint identifies each middleware process (execution semantics) which exists in the middleware system. By invoking the execution entrypoint by a service requester, the relevant process starts in the middleware system. The data entrypoint is used by service requester for interfacing the middleware process during its execution in order to provide some data for the execution asynchronously.
- **Interactions**. There are two types of interactions between service requester/provider and the middleware, namely (1) *late-binding interactions*, and (2) *conversation interactions*. Late-binding interactions allow service requester or provider to interact with the middleware in order to get or provide some information for the middleware process during late-binding phase. Conversation interactions allow to exchange information between service requester and service provider in order to facilitate conversation between them. Conversation interactions happen through the middleware (which provide the added value of mediation functionality during conversation).

In order to further illustrate the above described aspects of integration between service requesters/providers and the middleware, following figure shows entrypoints, communication and interactions together with a sample execution semantics. In this figure, the

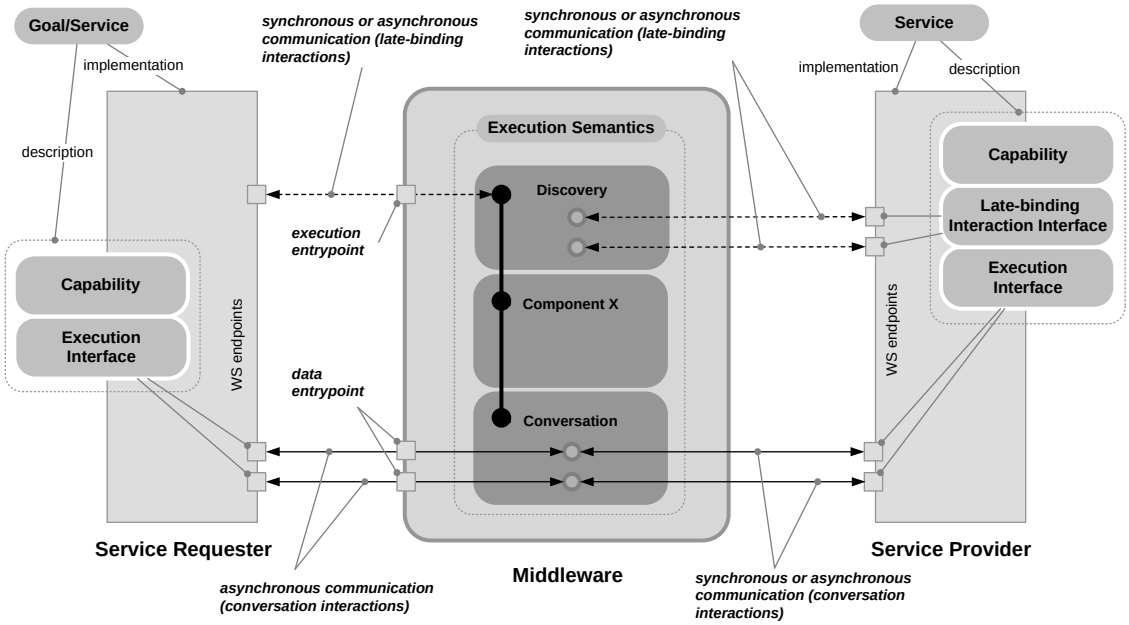

Figure 2.4: Integration Styles on the Middleware

service requester invokes the execution entrypoint of the middleware through which the requester's goal is sent and the execution semantics is started (these interactions are late-binding interactions run in synchronous or asynchronous way). In the middleware, the discovery component tries to find appropriate services from the repository where services have been registered. During the discovery-time, the middleware might interact with potential services in order to retrieve additional information needed to decide on match between requester's goal and the service. Through these interactions, concrete instance data can be be retrieved from the service requester in order to complete the discovery process. Such data could convey information about price or product availability which cannot be directly included in service descriptions (usually from practical reasons). Late-binding interactions may be however used for other purposes than for discovery, such as interactions related to negotiation, contracting or bidding. Such interactions run in synchronous or asynchronous way and in addition must conform to the protocol defined by a component which require or allow such interactions (these interactions are also described using WSMO service interface). After discovery, some other processing may apply during execution in the middleware (such as selection etc. – these are not shown in the figure). Finally, the conversation between service requester and discovered services is facilitated by the middleware (these are conversation interactions run in asynchronous way). During the conversation, other components may apply such as data and process mediation which maintain the interoperability in case different semantics is used by service requester or service providers.

Execution Semantics

In this section we describe a set of execution semantics which allow so called *goal-based invocation of semantic web services*. In particular, as depicted in figure 2.5 we define three basic types of execution semantics, namely *AchieveGoal Execution Semantics*, *RegisterGoal Execution Semantics*, and *Optimized AchieveGoal Execution Semantics*. With respect to the phases of the execution semantics (i.e. late-binding and conversation phases), it also makes sense to break down the execution semantics into the design-time and run-time stages. The late-binding could be then performed during the design-time when approval of the result business process would be made by the domain expert, and run-time when the conversation would be executed. Such break down of the execution semantics is still under development and is not described in this work.

The execution semantics described here are the simplest form for the goal-based invocation which involves both late-binding and conversation phases. However, as already mentioned earlier these execution semantics do not take into account service composition. Service composition adds additional level of complexity to the middleware process which involves all components and will be the subject of work in the next phases of the architecture design.

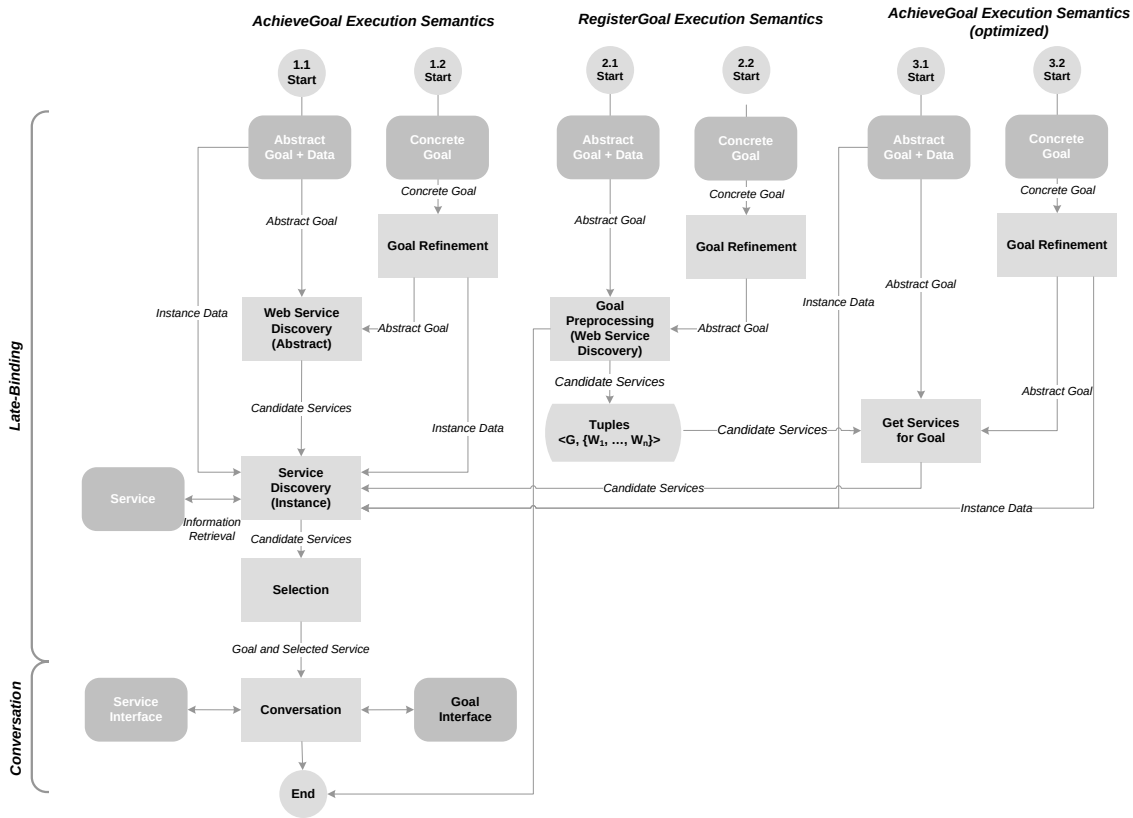

Figure 2.5: Execution Semantics

Each execution semantics is initiated with the WSMO goal provided as the input. We further distinguish two basic variants for each execution semantics. For the first variant, the execution semantics expects the abstract goal and for the second variant the execution semantics expects the concrete goal. The abstract goal contains no instance data in its definition (instance data is provided separately from the goal definition either synchronously or asynchronously) whereas concrete goal contains instance data directly embedded in its definition (directly as part of WSMO capability definition). For example, the WSMO capability of the concrete goal can contain axioms in a form $?x[name\ hasValue\ \text{“}HarryPotter''] \ memberOf\ book$ whereas abstract goal contains axioms in a form $?x\ memberOf\ book$ where instance of the $book$ concept is provided separately from the goal definition. Since the abstract goal and instance data is required for the processing of the goal, the refinement of the goal must be first performed when the concrete goal is supplied (see 1.2, 2.2, and 3.2 branches in the figure 2.5). During the refinement, the reasoning about goal definition is performed with result of the new abstract goal and instance data defined separately which both correspond to the original concrete goal definition. The algorithm for the goal refinement is the subject of work at the time of writing this article.

AchieveGoal Execution Semantics. For the execution semantics *AchieveGoal* (see branches 1.1 and 1.2 in the figure 2.5), (1) web services discovery, (2) service discovery, (3) service selection, and (4) conversation is performed. During the web service discovery, matching of the abstract definition of the goal with abstract definitions of potential services (previously published in repositories) which can fulfill the goal is performed. A number of possible set-theoretic relationships is evaluated between the goal and web services, namely exact match, plug-in match, subsumption match, intersection match, and disjnontness [4]. When the match is found[12], the next step is to check whether the goal and its data also satisfy a concrete form of the abstract service. For this purpose, possible interactions with the service can happen in order to retrieve additional data to complete the discovery process (see late-binding interactions in figure 2.4). Such data cannot be usually included in static service descriptions and needs to be retrieved during discovery-time (e.g. data about price or product availability). On result, a set of candidate services which satisfy the goal is passed to the selection component which, based on additional criteria (e.g. quality of service), selects the best service which best satisfies user preferences (these preferences are included as part of the goal definition in non-functional descriptions). Finally, the conversation is started between the selected service and the goal by processing of goal and service interfaces.

RegisterGoal Execution Semantics. This execution semantics allows to register a goal definition in the middleware when pre-processing in terms of abstract discovery of the goal and potential services is performed off-line separated from the goal-based invocation. This approach reflects the fact that the matching process, which involves reasoning, is time consuming and will hardly scale. From this reason, the abstract goal is matched with possible service candidates from the service repository and the result in a form of tuples

[12]For the match we consider exact match only; the other cases are subject of composition.

$< G, \{W_1, ..., W_n\} >$ is stored in the repository of the middleware. G represents the abstract description of the goal and a set $\{W_1, ..., W_n\}$ represents a list of candidate web services where where each web service W_i match the goal description G.

Optimized AchieveGoal Execution Semantics. This execution semantics performs goal-based invocation of service where goal has been previously registered with the *Register-Goal* execution semantics. In this case, the goal and its candidate services is first found in the goal repository. The result is passed to the instance discovery where further processing is performed as described in the original *AchieveGoal* execution semantics. Such approach can significantly improve the performance of goal-based invocation as the major burden of the processing, namely abstract discovery, is performed off-line during goal registration.

Conversation. The conversation phase follows up the binding of the service and the goal and enables interchange of messages between requester and provider by processing of their choreography interfaces. In addition, those interfaces might follow slightly different protocols as well as can use different ontologies, thus data and process mediation is applied during this processing. In figure 2.6 a control state diagram for the conversation between service requester and service provider is shown. The core functionality for the conversation is provided by the Choreography component which processes the requester's and provider's choreographies according to the control mechanisms implemented in the Process Mediator. The choreographies in WSMO are modeled as Abstract State Machines thus its processing is based on evaluation of rules' heads in the processing memory of the Choreography component. When this evaluation holds, the rule body is executed resulting in *adding*, *updating* or *removing* data in the processing memory. Adding/updating data in the memory means that the actual data needs to be obtained from the service by invoking underling operation of the service from the WSDL description (information which WSDL operation to invoke is part of the grounding definition of the choreography description in WSMO service).

The conversation is initiated by loading requester's and provider's choreography interfaces (these interfaces are taken from goal and service descriptions respectively) (transitions 1.1, 1.2 and state 2). When the choreographies are loaded, the Choreography gets to the control state (state 3) managing the whole conversation process. At this stage, it can wait for the new data, end the conversation or request process mediator to add the new data to a processing memory of a loaded choreography. Requester or provider, by following a protocol described by their choreography interfaces, can either send or expect to receive messages from/to the middleware. Also, there might be already some data available initially which could have been provided as part of goal definition. When data is available from either side, the control is passed to the process mediator (transition 3.3 and state 5). Next, the data is mediated to the provider's ontology if the data originates from the requester or vice-versa (transition 5.1, state 6). After the data mediation is finished, the process mediator decides where to put the new data, either into processing memory of the requester's choreography, processing memory of the provider's choreography or both. This decision is based on the evaluation of rules' heads of each choreography, in

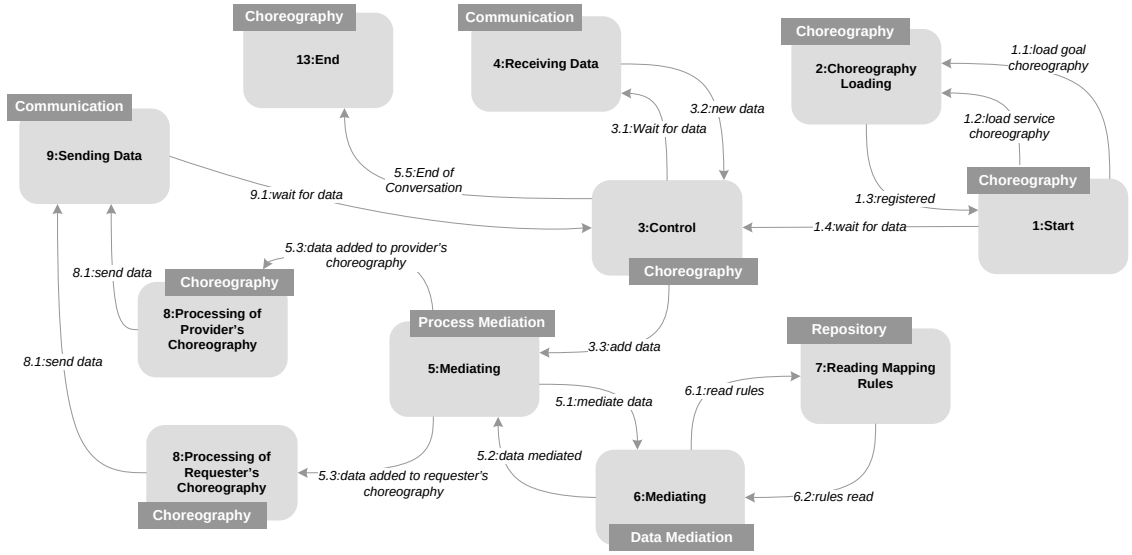

Figure 2.6: Control State Diagram of the Conversation Phase

particular it is evaluated if the data could be potentially used in subsequent processing of respective choreography. Based on this evaluation, the data is added to the particular choreography (transitions 5.3). Next, the updated choreographies are processed meaning, that the next rule of each updated choreography is evaluated. In particular, for a rule which head satisfies the content of the processing memory of the choreography, the body is executed which means that some data should be either added, updated or removed from the memory (state 8). In case of add or update, the actual data needs to be obtained from the service requester or service provider respectively. This is done by processing of the grounding definition of the concept that needs to be added/updated when invocation of underlying operation of the WSDL service is performed (transition 8.1 and state 9). When the new data is added, the conversation gets back to the control state and potentially to receiving data state (transitions 9.1, state 3 or transition 5.5, state 4 respectively). The Choreography might also evaluate that there is no additional rules to be processed and may get to the end of conversation state (transition 5.5, state 13). In figure 2.6.3, the conversation execution semantics described here is depicted from the point of running in the middleware system.

2.5 Technology View

Technology chosen for the implementation of the architecture is WSMO model and WSDL for modeling and semantic description of business services, J2SE for implementation of Web Service Modeling Toolkit, and J2EE for implementation of the middleware as the WSMX prototype. In this section we concentrate on the design of the technology for

the middleware system which address requirements of component management, inter-component messaging and configuration of execution semantics. With this respect, we take the advantages of existing technologies available for middleware platforms and applications servers.

2.6 Middleware Core

The core of the middleware takes the role of component coordination, i.e. it manages interactions between other components through the exchange of messages containing instances of WSMO concepts expressed in WSML and provides the *microkernel* and messaging infrastructure for the middleware.

The middleware Core is responsible for handling following three main functional requirements. The middleware Core thus implements the middleware kernel utilizing Java Management Extensions (JMX) as described by Haselwanter in [3].

- A framework for the management and monitoring to start and stop the system and together with monitoring of its health.
- Enabling communication and coordination between components, i.e. handling messages, routing messages to a suitable target component, etc.
- Support for the lifecycle of execution semantics. Multiple definitions of execution semantics are supported and multiple instances of each execution semantics may run concurrently.

2.6.1 Management

It is common for middleware and distributed computing systems that management of their components becomes a critical issue. In the design of the middleware, we have made a clear separation between operational logic and management logic, treating them as orthogonal concepts. By not separating these two elements, it would become increasingly difficult to maintain the system and keep it flexible. In figure 2.7, an overview of the infrastructure provided by the middleware Core to its components is depicted. This infrastructure primarily allows to manage and monitor the system. In the core of the management lies a management agent which offers several dedicated services. The most important one is the *bootstrap service* responsible for loading and configuring application components. In here, the management agent plays the role of a driver which is directly built into the application. The Core in addition employs *self-management* techniques through scheduled operations, and allows *administration* through a representation-independent management and monitoring interface. Through this interface, a number of

management consoles can be interconnected, each serving different management purposes. In particular, terminal, web browser and eclipse management consoles have been implemented.

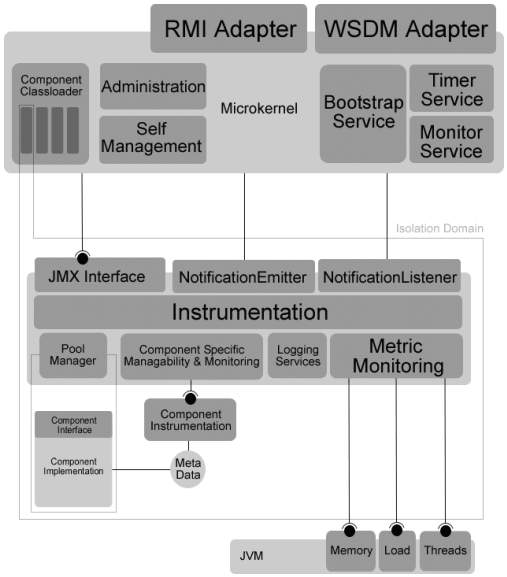

Figure 2.7: Component Management in the Middleware Core

Similarly as in state-of-the art middleware systems, the Core hosts a number of subsystems that provide services to components and enable inter-component communication. In addition, the Core provides a number of services including *pool management* which takes care of handling component instances, and logging, transport and lifecycle services. The Core also exploits the underlying (virtual) machine's instrumentation to monitor performance and system health metrics. Although some general metrics can be captured for all components, the component metric monitoring allows to capture metrics specific to some components which require custom instrumentation. Such customization can be achieved by extending the configuration for the instrumentation of a specific component which is done independently from the implementation of the component itself.

With respect to the distributed principle of the architecture, the Core infrastructure may act as a facade to distributed components. However, the preferred way to distribution is to organize the system as *federations of agents*. Each agent has its own Core component and a particular subset of functional components. In order to hide the complexity of the federation for the management application, a single *agent view* is provided, i.e. single point of access to the management and administration interfaces. This can be achieved by propagating requests within the federation via proxies, broadcasts or directories. A federation thus consists of a number of Cores, each of them operating a kernel per one machine and hosting a number of functional components.

2.6.2 Communication and Coordination

The middleware avoids hard-wired bindings between components when the inter-component communication is based on events. If some functionality is required, an event representing the request is created and published. A component subscribed to this event type can fetch and process the event. The event-based approach naturally allows event-based communication within the middleware. As depicted in 2.8, the exchange of events is performed via Tuple Space which provides a persistent shared space enabling interaction between components without direct exchange of events between them. This interaction is performed using a publish-subscribe mechanism.

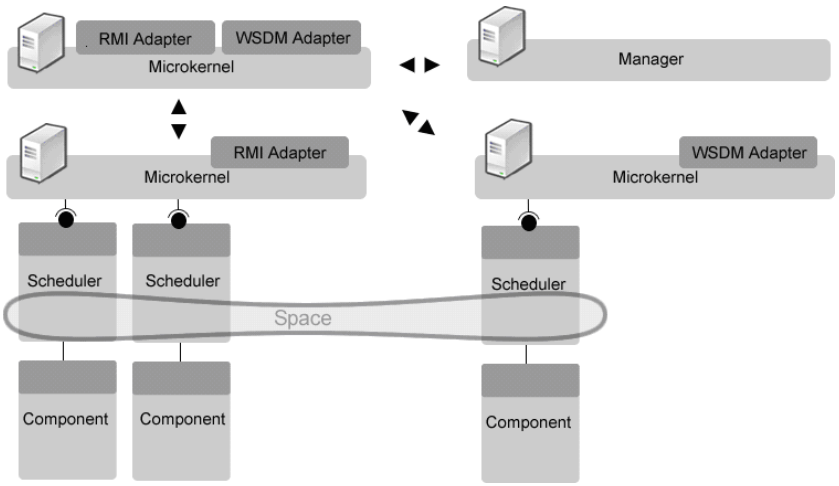

Figure 2.8: Communication and Coordination in the Middleware

The Tuple Space enables communication between distributed components running on both local as well as remote machines while at the same time the components are unaware of this distribution. For this purpose, an additional layer provides components with a mechanism of communication with other components which shields the actual mechanism of local or remote communication. The Tuple Space technology used in the middleware is based on Linda [2] which provides a shared distributed space where components can publish and subscribe to tuples. Subscription is based on templates and their matching with tuples available in the space. The space handles data transfer, synchronization and persistence. The Tuple Space can be in addition composed of many distributed and synchronized Tuple Space repositories. In order to maximize usage of components available within one machine, instances of distributed Tuple Space are running on each machine and newly produced entries are published locally. Before synchronization with other distributed Tuple Spaces, a set of local template rules is executed in order to check if there are any local components subscribed to the newly published event type. It means that by default (if not configured otherwise), local components have priority in receiving locally published entries.

Through the infrastructure provided by the Core, component implementations are sep-

arated from communication. This infrastructure is made available to each component implementation during instantiation of the component carried out by the Core during the *bootstrap* process (a process that occurs when a component is loaded by the system). Through the use of JMX and reflection technology, this can occur both at start-up as well as after the system is up and running. The communication infrastructure has the responsibility to interact with the transport layer (a Tuple Space instance). Through the transport layer, component subscribe to an event-type template. Similar mechanism applies when events are published in the Tuple Space. In order to enable a component to request functionality from another component a proxy mechanism is used. When a component need to invoke other component's functionality, the proxy creates the event for this purpose and publishes it on the Tuple Space. At the same time, the proxy subscribes to the response event and takes care of the correlation. From the perspective of the invoking component, the proxy appears as the component being invoked. This principle is the same as one used by Remote Method Invocations (RMI) in object-oriented distributed systems.

2.6.3 Execution Semantics

Execution Semantics enable a combined execution of functional components as illustrated in figure 2.9. Execution semantics defines the logic of the middleware which realize the middleware behavior. The Core infrastructure provides the implementation that enables a general computation strategy by enforcing execution semantics, operating on transport as well as component interfaces. It takes events from the Tuple Space and invokes the appropriate components while keeping track of the current state of execution. Additional data obtained during execution can be preserved in the particular instance of an execution semantic. The Core provides the framework that allows execution semantics to operate on a set of components without tying itself to a particular set of implementations. In particular, Core component takes care of the exectuion semantics lifecycle, management and monitoring. As described in secion 2.4, the execution semantics form processes run in the middleware (middleware processes).

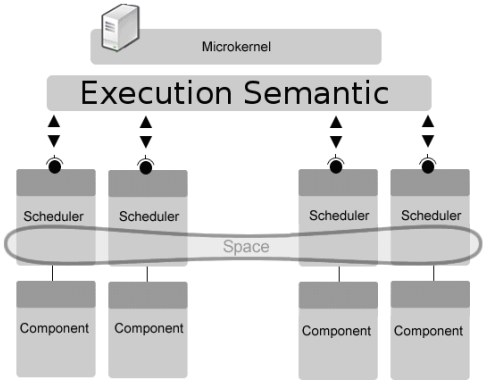

Figure 2.9: Execution Semantics in the Middleware Core

Figure 2.10 depicts how services are decoupled from the process (described in the execution semantics) by means of wrappers. Based on an execution semantics definition,

these wrappers will only be able to consume and produce particular types of events. In a running system dynamic execution semantics are achieved by mapping abstract system behavior into real event infrastructure of the system. The wrappers are generated and managed by the Core in order to separate components from the transport layer for events. One wrapper raises an event with some message content and another wrapper can at some point in time consume this event and react to it. However, component developers do not need to be aware of this mechanism.

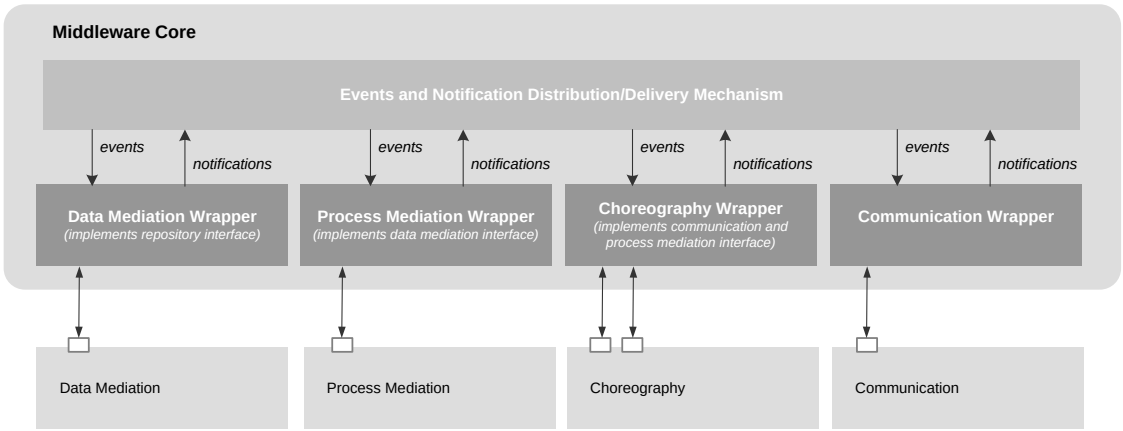

Figure 2.10: Component's Wrappers and Event Messaging

Conceptual definition of various execution semantics and their variants is described in section 2.4.2. In particular, in figure 2.6, the control state diagram of the conversation is depicted. In figure 2.11, the same conversation process is depicted how it is run in the middleware.

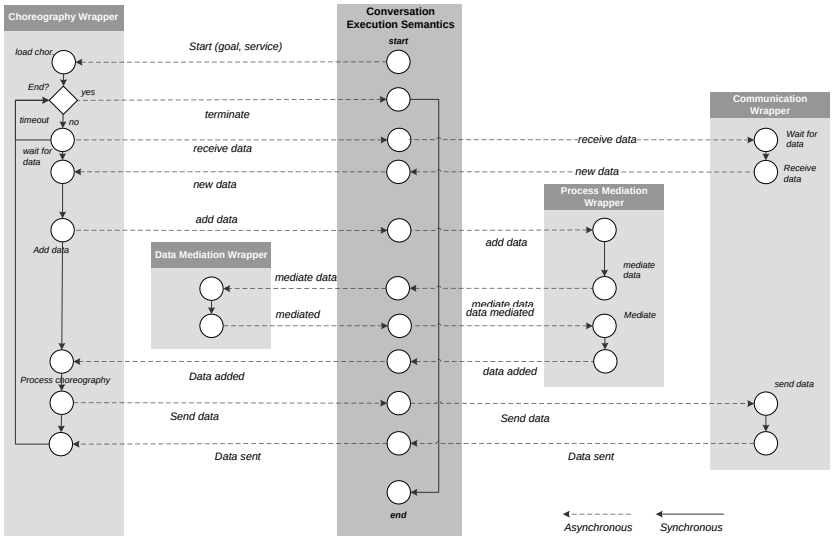

Figure 2.11: Conversation run in the Middleware

Chapter 3

Conclusion and Future Work

This work is the first version of the architecture for the Semantic Web Services which aims at establishing the grounds for joint work on Semantic Service Oriented Architecture (SESOA) and in particular exploited in the OASIS Semantic Execution Environment Technical Committee (OASIS SEE TC). In this work we defined a number of views through which the architecture is described, namely *global view* identifying a number of layers from the global viewpoint on the architecture, *service view* identifying various types of services and describing these services in detail, *process view* describing processes which are both provided as well as facilitated by the architecture, and *technology view* revealing details of the technology used for implementation of the architecture and its middleware system in particular.

In the second version of the architecture we plan to revise this work based on the communication with various groups involved in the architecture within various EU funded projects and strengthen this way the architecture grounds. In addition, we plan to enhance the functionality of the architecture and its middleware system by incorporating composition and orchestration. Such functionality will add additional complexity to the processes run in the middleware. We also plan to build the overlay network above the nodes of middleware system, enable the P2P communication within the architecture and promote the distributed mechanisms of the Semantic Service Oriented Architecture by allowing semantic routing of requests within the architecture.

Bibliography

[1] Emilia Cimpian and Adrian Mocan. D13.7 v0.1 process mediation in wsmx. In *WSMO Working Draft*, 2005.

[2] D Gelernter, N. Carriero, and S. Chang. Parallel Programming in Linda. In *Proceedings of the International Conference on Parallel Processing*, 1985.

[3] Haselwanter, Thomas. *WSMX Core - A JMX Microkernel*. PhD thesis, University of Innsbruck, 2005.

[4] Uwe Keller, Ruben Lara, and Axel Polleres. WSMO Discovery, WSMO Working Draft D5.1v0.1, Available from http://www.wsmo.org/2004/d5/d5.1/v0.1/. Technical report, 2004.

[5] Jacek Kopecký, Dumitru Roman, Matthew Moran, and Dieter Fensel. Semantic web services grounding. In *AICT/ICIW*, page 127, 2006.

[6] David Martin et al. Owl-s: Semantic markup for web services. Member submission, W3C, 2004. Available from: `http://www.w3.org/Submission/OWL-S/`.

[7] Adrian Mocan, Emilia Cimpian, and Mick Kerrigan. Formal model for ontology mapping creation. In *International Semantic Web Conference*, pages 459–472, 2006.

[8] Adrian Mocan, Matthew Moran, Emilia Cimpian, and Michal Zaremba. Filling the gap - extending service oriented architectures with semantics. In *ICEBE*, pages 594–601. IEEE Computer Society, 2006.

[9] E. Motta, J. Domingue, L. Cabral, and M. Gaspari. IRS-II: A Framework and Infrastructure for Semantic Web Services. In *Proc. of the second International Semantic Web Conference Sanibal Island, FL, USA*, pages 306–318, 2003.

[10] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. Semantic Web Services: Meteor-S Web Service Annotation Framework. In *13th International Conference on World Wide Web*, pages 553–562, 2004.

[11] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubn Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontologies*, 1(1):77 – 106, 2005.