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),

Michal Zaremba, Adrian Mocan, Mick Kerrigan, Thomas Hasselwanter

(University of Innsbruck, Austria)

Abstract.
EU-IST Network of Excellence IST-2004-507482 Deliverable D2.4.10 version 2

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/v2
Project	KWEB EU-IST-2004-507482
Version	v2.0
Date	December 30, 2007
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
1.1	20.10.07	Tomas Vitvar	Updates of the deliverable v1.0
1.2	30.10.07	Tomas Vitvar	Comments implemented
2.0	03.12.06	Tomas Vitvar	Final second version.

Executive Summary

The architecture for the Semantic Web Services is the 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 researchers 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 reflects the second version 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. A special focus of this deliverable is on definition of the execution semantics for the execution phase of the service provisioning process. We formally define the algorithm for the interaction of various components within that process.

The major content of this deliverable has been used as the input for the accepter and published jounral articles, namely *SESA: Emerging Technology For Service-Centric Enviornmnets*[15] published in IEEE Software special issue on Service-Centric Software Components by IEEE Press and *Semantically-enabled Service Oriented Architecture: Concepts, Technology and Application*[13] published in Service Oriented Computing and Aplpications journal by Springer.

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 allow to add additional concepts and functionality to the architecture in the future.

Contents

Chapter 1

Introduction

The design of enterprise information systems has gone through a great change in recent years. In order to respond to requirements of business for flexibility and dynamism, traditional monolithic applications are being challenged by smaller composable units of functionality known as services. Information systems thus need to be re-tailored to fit this paradigm, with new applications developed as services and legacy systems to be updated in order to expose service interfaces. The drive is towards a design of information systems which adopt paradigms of Service Oriented Architectures (SOA). 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 change in business requirements, existing SOA solutions will prove difficult to scale without a proper degree of automation. While today's service technologies around WSDL, SOAP, UDDI and BPEL certainly brought a new potential to SOA, they only provide partial solution to interoperability, mainly by means of unified technological environments. Where content and process level interoperability is to be solved, ad-hoc solutions are often hard-wired in manual configuration of services or workflows while at the same time 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 which in turn limits possibilities for automation. In order to address these drawbacks, the extension of SOA with semantics offers scalable integration, that is more adaptive to changes that might occur over a software system's lifetime. Semantics for SOA allow the definition of semantically rich and formal service models where semantics can be used to describe both services offered and capabilities required by potential consumers of those services. Also the data to be exchanged between business partners can be semantically described in an unambiguous manner in terms of ontologies. By means of logical reasoning, semantic SOA thus promotes a total or partial automation of service discovery, mediation, composition and invocation. Semantic SOA does not however mean to replace existing

integration technologies. The goal is to build a new layer on the top of existing service stack while at the same time adopting existing industry standards and technologies used within existing enterprise infrastructures.

1.1 Goal of the Deliverable

The goal of this deliverable is to provide a conceptual and logical/detail design of the architecture for the semantic web services forming Semantic Service Oriented Architecture (SESA). 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

In Section 2 we define the global architecutre and governing principles which drive the architecture research, design and imlpamantation. We also describe the running example we refer to throughout the deliverable. In Section 3 we describe the architectrue from several perspectives including services that architecture offers and supports, processes according to which the architecture behaves and technology implementing the architecture, its services and processes. In Section 4 we describe in detail the algorithm for the execution semantics of the architecture execution processes and in Section 5 we describe the evaluation process and results of the architecture from the SWS Challenge initiative. In Section 6 we conclude the delivarable.

Chapter 2

Global Architecture

In this chapter we define several governing principles for the SESA research, design and implementation as well as underling SESA technology. We also intruduce a running example used to illustrate various aspects of the SESA in this deliverable.

2.1 Governing Principles

The SESA architecture builds on a number of principles which define essential background knowledge governing the architecture research, design and implementation. These principles reflect fundamental aspects for service-oriented and distributed environment which all promote intelligent and seamless integration and provisioning of business services. These principles include:

- **Service Oriented Principle** represents a distinct approach for analysis, design, and implementation which further introduces particular principles of service reusability, loose coupling, abstraction, composability, autonomy, and discoverability.
- **Semantic Principle** allows a rich and formal description of information and behavioral models enabling automation of certain tasks by means of logical reasoning. Combined with the 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.
- **Problem Solving Principle** reflects Problem Solving Methods as one of the fundamental concepts of the artificial intelligence. It underpins the ultimate goal of the architecture which lies in so called *goal-based discovery and invocation of services*. Users (service requester's) describe requests as goals semantically and independently from services while the architecture solves those goals by means of

logical reasoning over goal and service descriptions. Ultimately, users do not need to be aware of processing logic but only care about the result and its desired quality.

- **Distributed Principle** allows to aggregate the power of several computing entities to collaboratively run a task in a transparent and coherent way, so that from a service requester's perspective they can appear as a single and centralized system. This principle allows to execute a process across a number of components/services over the network which in turn can promote scalability and quality of the process.

2.2 Global SESA Architecture

In this section we define the overall SESA architecture, depicted in Figure 2.1, building on governing principles introduced in section 2.1. These layers are: (1) *Stakeholders* forming several groups of users of the architecture, (2) *Problem Solving Layer* building the environment for stakeholders' access to the architecture, (3) *Service Requesters* as client systems of the architecture, (4) *Middleware* providing the intelligence for the integration and interoperation of business services, and (5) *Service Providers* exposing the functionality of back-end systems as Business Services.

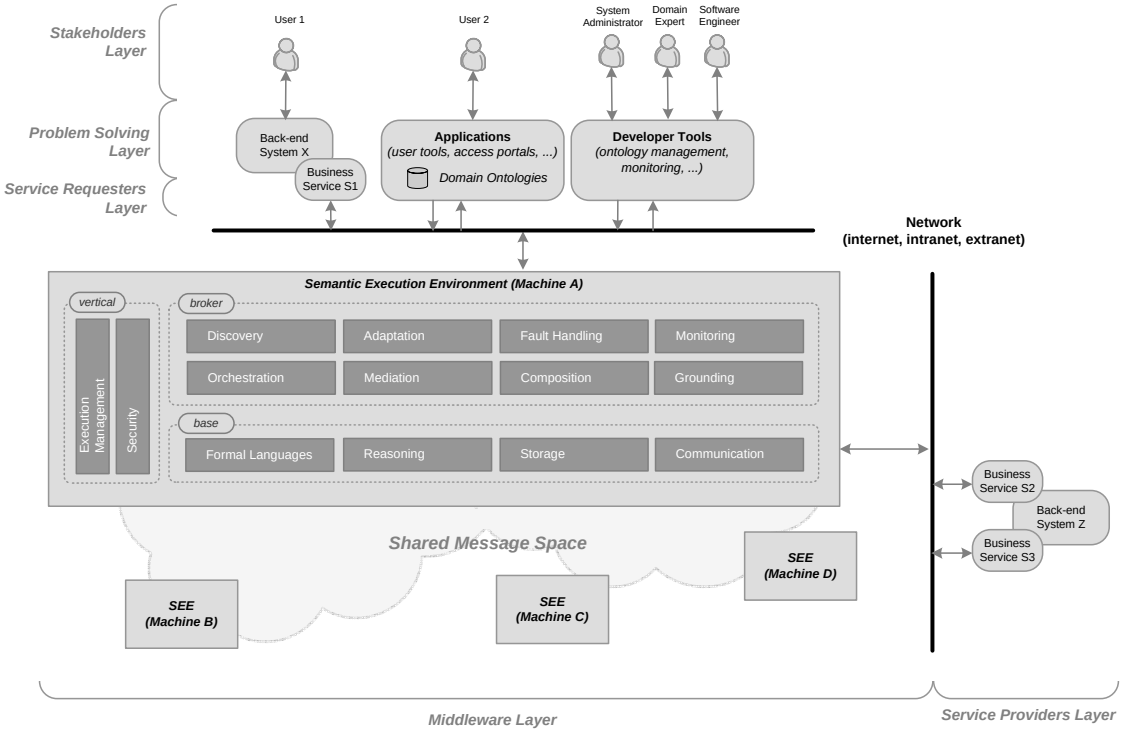

Figure 2.1: Global View

2.2.1 Stakeholders Layer

Stakeholders form the group of various users which use the functionality of the architecture for various purposes. Two basic groups of stakeholders are identified: *users*, and *engineers*. Users form the group of those stakeholders to which the architecture provides end-user functionality through specialized applications. 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. 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 operations. On the other hand, the group of engineers form those stakeholders which perform development and 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 engineers could be involved in this process ranging from domain experts (modeling, creation), system administrators (deployment, management) and software engineers.

2.2.2 Problem Solving Layer

The problem solving layer contains applications and tools which support stakeholders during formulation of problems/requests and generates descriptions of such requests in the form of user goals. Through the problem solving layer, a user will be able to solve his/her problems, i.e. formulate a problem, interact with the architecture during processing and get his/her desired results. This layer contains *back-end systems* which directly interface the middleware within business processes, specialized *applications* built for specific purpose in a particular domain which also provide specific domain ontologies, and *developer tools* providing functionality for development and administrative tasks within the architecture.

Developer tools provide a specific functionality for engineers, i.e. domain experts, system administrators and software engineers. The functionality of developer tools cover the whole SOA lifecycle including service modeling, creation (assembling), deployment (publishing), and management. The vision is to have an Integrated Development Environment (IDE) for management of the architecture. The IDE aids developers through the development process including engineering of semantic descriptions (services, goals, and ontologies), creation of mediation mappings, interfacing with architecture middleware and external systems. By combining this functionality, a developer will be allowed to create and manage ontologies, Web services, goals and mediators, create ontology to ontology mediation mappings and deploy these mappings to the middleware.

Applications provide a specialized functionality for architecture end-users. They provide a specialized domain specific ontologies, user interfaces and application functionality through which stakeholders interact with the architecture and its processes. Through specialized applications in a particular application settings, the technology and its func-

tionality can be also validated and evaluated. A specialized end-user functionality is subject to design and development in application oriented projects, such as SemanticGov[1]. In this project we develop a specialized functionality for clients to interact with public administration processes facilitated by the middleware system.

2.2.3 Service Requesters Layer

Service requesters act as client systems in a client-server settings of the architecture. With respect to the problem sovling principle, they are represented by goals created through problem/request formulation by which they describe requests as well as interfaces through which they wish to perform conversation with potential services. Service requesters are present for all applications and tools from problem solving layer and are bound to specific service semantics specification.

2.2.4 Middleware Layer

Middleware is the core of the architecture providing the main intelligence for the integration and interoperation of Business Services. For the purposes of the SESA, we call this middleware Semantic Execution Environment (SEE) (the SEE conceptual architecture is depicted in figure 2.1). The SEE defines the necessary conceptual functionality that is imposed on the architecture through the underlying principles defined in section 2.1. Each such functionality could be realized (totally or partially) by a number of so called *middleware services* (in section 3.1.1 we further define middleware services that realize these conceptual functionalities). We further distinguish this functionality into the following layers: *base layer*, *broker layer*, and *vertical layer*. The SEE middleware system is being specified within the OASIS Semantic Execution Environment Technical Committee (OASIS SEE TC)[2] with reference implementations of WSMX[3] and IRS-III[4].

Vertical Layer The Vertical layer defines the middleware framework functionality that is used across the Broker and Base Layers but which remains invisible to them. This technique is best understood through so called "Hollywood Principle" that basically means "Don't call us, We'll call you". With this respect, framework functionality always consumes functionality of Broker and Base Layers, coordinating and managing overall execution processes in the middleware. For example, Discovery or Data Mediation is not aware of the overall coordination and distributed mechanism of the Execution Management.

[1] http://www.semantic-gov.org
[2] http://www.oasis-open.org/committees/semantic-ex
[3] http://www.wsmx.org
[4] http://kmi.open.ac.uk/projects/irs

- *Execution Management* defines a control of various execution scenarios (called execution semantics) and handles distributed execution of middleware services.
- *Security* defines a secure communication, i.e. authentication, authorization, confidentiality, data encryption, traceability or non-repudiation support applied within execution scenarios in the architecture.

Broker Layer The Broker layer defines the functionality which is directly required for a goal based invocation of Semantic Web Services. The Broker Layer includes:

- *Discovery* defines tasks for identifying and locating business services which can achieve a requester's goal.
- *Orchestration* defines the execution of a composite process (business process) together with a conversation between a service requester and a service provider within that process.
- *Monitoring* defines a monitoring of the execution of end point services, this monitoring may be used for gathering information on invoked services e.g. QoS related or for identifying faults during execution.
- *Fault Handling* defines a handling of faults occurring within execution of end point Web services.
- *Adaptation* defines an adaptation within particular execution scenario according to users preferences (e.g. service selection, negotiation, contracting).
- *Mediation* defines an interoperability at the functional, data and process levels.
- *Composition* defines a composition of services into an executable workflow (business process).
- *Grounding* defines a link between semantic level (WSMO) and a non-semantic level (e.g. WSDL) used for service invocation.

Base Layer The Base layer defines functionality that is not directly required in a goal based invocation of business services however they are required by the Broker Layer for successful operation. Base layer includes:

- *Formal Languages* defines syntactical operations (e.g. parsing) with semantic languages used for semantic description of services, goals and ontologies.
- *Reasoning* defines reasoning functionality over semantic descriptions.
- *Storage* defines persistence mechanism for various elements (e.g. services, ontologies).

- *Communication* defines inbound and outbound communication of the middleware.

The SEE middleware can operate in a distributed manner when a number of middleware systems connected using a shared message space operate within a network of middleware systems which empoweres this way a scalability of integration processes.

2.2.5 Service Providers Layer

Service providers represent 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 semantically described business services .

2.3 Underlying Concepts and Technology

In this section we describe a concrete semantic service model and technology we choose for realization of the SESA architecture described in section 2.2. In general, a semantic service model builds the additional layer on the top of the current Web service stack by introducing a semantic mark-up for functional, non-functional and behavioral aspects of service descriptions. Today, several initiatives exists in this area such as Web Service Modeling Ontology (WSMO)[11], OWL-S[8] and WSDL-S[10].

With respect to requirements imposed on the architecture through the governing principles, we choose the WSMO model for our work. The reason why we choose WSMO and not other specifications (e.g. OWL-S) is that WSMO has a well defined focus which is in solving of integration problems by clear separation of requester and provider side (i.e. between goals and services) and thus fully adopts the semantic, problem solving and service orientation principles (see section 2.1). In addition, WSMO is being developed as a complete framework including: the conceptual model describing all relevant aspects of Web services – ontologies, goals, web services and mediators, Web Service Modeling Language (WSML)[11] – a family of ontology languages based on different logical formalisms and different levels of logical expressiveness (including both Description Logic and Logic Programming representation formalisms), Web Service Execution Environment (WSMX) – a reference implementation for the middleware system, and the Web

Service Modeling Toolkit (WSMT)[5] – an IDE used for engineering of WSMO descriptions (services, goals, and ontologies), creation of mediation mappings, and interfacing with architecture middleware and external systems. WSMO and its counterparts thus provides grounds for semantic modeling of services and semantic technology which could be well adopted to particular domain requirements (e.g. by choosing appropriate WSML variant according to particular modelling requirements, by extending the functionality of WSMX, WSMT, etc.).

WSMO conceptual model. In this paragraph we describe the WSMO top-level conceptual model which defines the ontology used for modeling of SESA business services. 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.

2.4 Running Example

In this section we introduce a running example which we will use throughout the deliverable to demonstrate various aspects of the SESA. This scenario and its implementation is

[5] http://wsmt.sourceforge.net

based on scenarios from the SWS Challenge[6], an initiative which provides a standard set of increasingly difficult problems, based on industrial specifications and requirements.

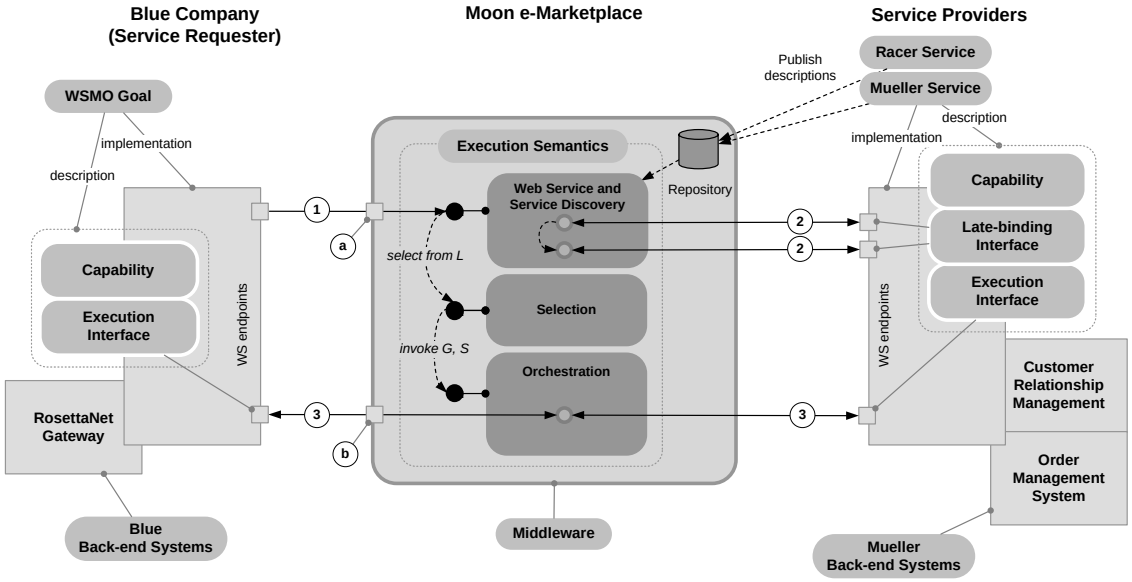

Figure 2.2: Running Example

As depicted in figure 2.2, the scenario introduces various service providers (such as Racer and Mueller) offering various purchasing and shipment options for various products through e-marketplace called Moon. On the other hand, there is a service requester called Blue who intends to buy and ship a certain product for the best possible price. The Moon operates the e-marketplace on the middleware system of the SESA. Following are the prerequisites of the scenario.

- Service requesters and providers are using various back-end systems for handling interactions in their environment. In particular, Mueller uses a Customer Relationship Management system (CRM) and an Order Management System (OMS) while Blue uses a standard RosettaNet[7] system.
- Engineers representing service requesters and service providers respectively model services and requests using the WSMO model while at the same time different ontologies as well as different descriptions of choreographies are used by service requester and provider. In particular, Blue sends its request and expects to receive a response according to the RosettaNet PIP3A4 Purchase Order (PO) specification. On the other hand, Mueller in order to process the request must perform more interactions with its back-end systems such as identify a customer in the CRM, open the

[6]http://www.sws-challenge.org

[7]RosettaNet (http://www.rosettanet.org) is the B2B integration standard defining standard components called Partner Interface Processes (PIPs), which include standard intercompany choreographies (e.g. PIP 3A4 Purchase Order), and structure and semantics for business messages.

order in the OMS, add all line items to the OMS from the request and close the order in the OMS. Thus, data and process interoperability issues exists between Blue and Mueller – Blue uses information model and choreography defined by the PIP3A4 and Mueller uses information model and choreography defined by the CRM/OMS systems.

- Service requesters and service providers maintain the integration with their respective back-end systems through the implementation of necessary web services (adapters for their back-end systems). Both Blue and Mueller are responsible for maintaining these adapters and their integration with the middleware through semantic descriptions and/or through interfaces with the middleware.
- Engineers representing service providers and service requesters respectively publish ontologies they use for WSMO goal and service descriptions in the middleware repositories. In addition, they publish mapping rules between their ontologies and existing ontologies in the middleware system.

The scenario runs as follows: all business partners first model their business services using WSMO (see section 3.1.3). After that, Blue sends the purchase order request captured in WSMO goal to the middleware system which on receipt of the goal executes the *achieveGoal* execution semantics including: (1) *discovery*, (2) *selection* and (2) *orchestration*. During discovery, the matching is performed for the goal and potential services at abstract level as well as instance levels (abstract-level discovery allows to narrow down number of possible Web services matching a given Goal while instance-level discovery carries out detailed matchmaking considering instance data of the service and the goal). During selection, the best service is selected (in our case Mueller service) based on preferences provided by Blue as part of the the WSMO goal description. Finally during orchestration, the execution and conversation of Blue and Mueller services is performed by processing the choreography descriptions from Blue's goal and Mueller's service. We will refer to this scenario and the figure 2.2 throughout the deliverable to illustrate various aspects of the SESA.

Chapter 3

Architecture Views

In this chapter we define the SESA 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.

3.1 Service View

The SEE consists of several decoupled services allowing independent refinement of these services - each of which can have its own structure without hindering the overall SEE architecture. Following the SOA design principles, the SEE architecture separates concerns of individual middleware services thereby separating service descriptions and their interfaces from the implementation. This adds flexibility and scalability for upgrading or replacing the implementation of middleware services which adhere to required interfaces.

Services provide certain functionality for certain purpose. With respect to the service orientation which enables a service level view on the organization we further distinguish services from several perspectives. From the view of services' *functionality*, we distinguish two types of 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. In the SESA architecture, business services are exposed by service providers, their back-end systems, as semantic descriptions conforming to the WSMO service model (as defined in section 2.3). Business services are published to the middleware repositories.

- **Middleware Services** are the main facilitators for integration and interoperation of business services. Middleware services are deployed to the middleware system.

From the view of business services' *abstraction*, we further distinguish following two types of services.

- **Web Services** are general services which might take several forms when they are instantiated (e.g. purchase a flight).
- **Services** are actual instances of Web Services which are consumed by users and which provide a concrete value for users (e.g. purchase a flight from Prague to Bratislava).

3.1.1 Middleware Services

In this section we describe a number of middleware services which realize the total or partial conceptual functionality of the middleware described in section 2.2.4. These services also reflect the current WSMX implementation – the reference implementation of the SEE middleware. In figure 3.1, middleware services are depicted together with their interfaces. Middleware services can be further combined into so called *middleware processes* which provide a certain functionality of the middleware system to its users. Middleware processes are described in section 3.2.2.

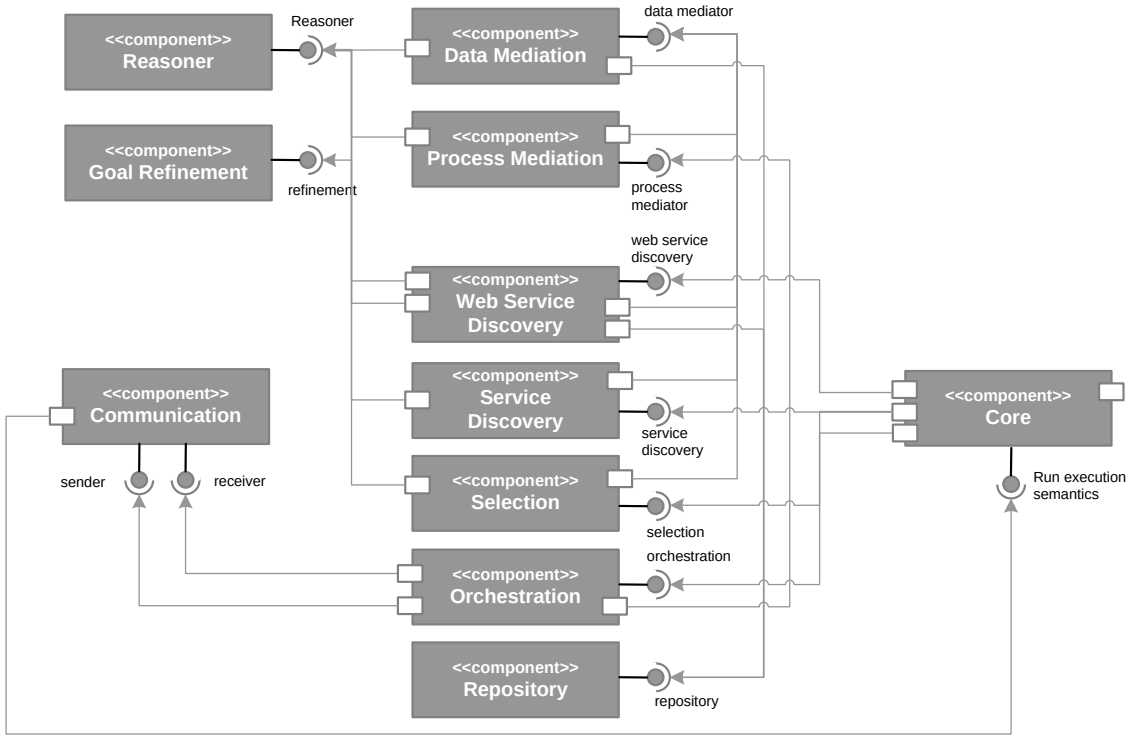

Figure 3.1: Service View – Middleware

Core. Core realizes the Execution Management, Monitoring, Fault Handling and Formal Languages conceptual functionalities of the middleware. The core manages the coordination and communication within middleware processes as defined by so called *Execution Semantics*. Execution semantics define the interactions of various middleware services which serves particular middleware process for specific purpose. Each middleware process is started by invocation of particular external interface (implemented by the Communication service) and can be interfaced through other external interfaces during execution by means 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 (see section 3.2). The middleware core also implements the formal language support, i.e. parser. The parser parses the semantic messages into the object model as defined by WSMO4J[1]. In addition, the core defines the distributed operation of the middleware which can operate on a number of physical machines connected using a shared message space. Shared spaces provide a messaging abstraction for distributed architecture which empowers the scalability of integration processes.

Communication. The Communication service realizes Communication and Grounding conceptual functionalities of the middleware. It facilitates inbound and outbound communication within 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 middleware 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.

Reasoner. The Reasoner service realizes the Reasoner functionality of the middleware. It provides reasoning support over the semantic descriptions of resources. Reasoning is an important functionality required during various execution processes and it 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 needed

[1] wsmo4j.sourceforge.net

when DL-based variant of WSML is used, and a Datalog or F-Logic based reasoner when WSML-Flight or WSML-Rule variant is used respectively. The use of the particular reasoner is application dependent as well as in tight connection with the formalism (e.g. DL vs. logic programming) used in modeling of semantic descriptions. The reasoner component offers a universal layer on top of several existing reasoners (called WSML2Reasoner [2]) that cover the above mentioned requirements. As a consequence, depending on the WSML variant used, the queries are passed to the proper underlying reasoning engine in a completely transparent manner. Development of reasoners for WSML is ongoing work in the WSML WG[3].

Repository. Repository realizes Storage conceptual functionality of the middleware. It manages the persistent mechanisms for various entities including goals, services, ontologies and mediators (mapping rules). All these entities are described using WSML semantic language.

Data Mediation. Data mediation realizes Mediation conceptual functionality of the middleware. It 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 WSMT ontology management tools. Detail description of data mediation for the semantic web services can be found in [9].

Process Mediation. Process mediaton realizes Mediation conceptual functionality of the middleware. It 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). By analysis of choreography descriptions, process mediator decides 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].

[2] http://tools.deri.org/wsml2reasoner
[3] http://www.wsmo.org/wsml

Goal Refinement. Goal Refinement realizes Discovery conceptual functionality of the middleware. Goal refinement is a process of creating an abstract goal from a concrete goal. The refinement of the goal must be first performed when the concrete goal is supplied for the web service discovery. 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 $?x[name\ hasValue\ "HarryPotter"]\ memberOf\ book$ whereas abstract goal contains axioms in a form $?x\ memberOf\ book$ where instance of the $book$ concept is provided separately from the goal definition.

Web Service Discovery. Web Service discovery realizes Discovery conceptual functionality of the middleware. 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*. More detailed information about web service discovery in WSMO can be found in [5].

Service Discovery. Service discovery realizes Discovery conceptual functionality of the middleware. Service discovery is a process of finding concrete services satisfying concrete 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 *late-binding interactions* (see section 3.2) within the execution process and service providers might take place in order to retrieve this information through specialized service interfaces. In WP2.4 we have elaborated on the service discovery in our ESWC2007 paper[14].

Selection. Selection realizes Adaptation conceptual functionality of the middleware. Selection is a process where one service which best satisfies 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. More detailed information about service selection in WSMO can be found in [16].

Orchestration. Orchestration realizes Orchestration conceptual functionality of the middleware. The Orchestration implements a run-time conversation between a service requester and a service provider by processing their choreography interfaces. This also involves interactions with process mediator (together with data mediator) as well as Communication service called each time the message exchange needs to happen between a service requester and a service provider.

3.1.2 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 (late-binding and execution of services). The important aspect of service creation phase is *semantic modeling of business services* which we define in following levels (see figure 3.2).

- **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 PIP messages), or various classifications such a NAICS[4] (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 the 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.

- **Physical Level.** Physical level represents the physical environment used for service invocation. In our architecture, we use WSDL and SOAP specification. For this

[4] http://www.naics.com

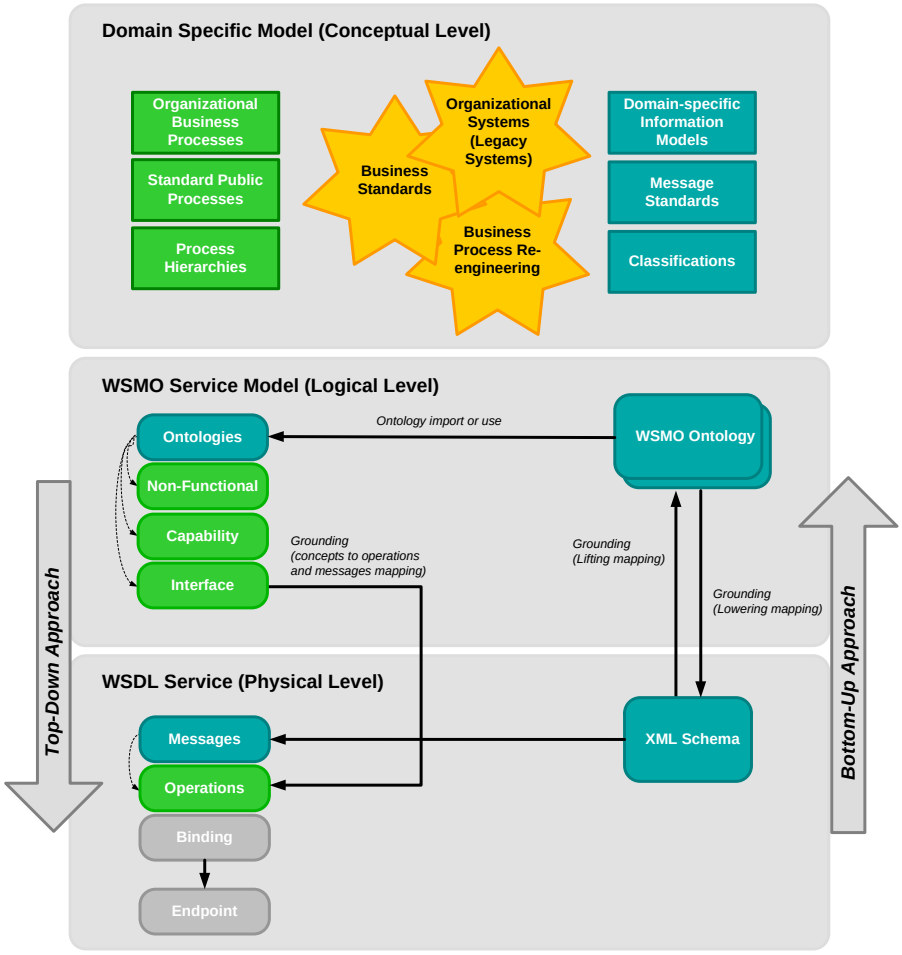

Figure 3.2: Semantic Business Service Modeling Levels

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[5]. The definition of grounding is dependent on the modeling approach and is discussed in following paragraph.

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,

[5] www.w3.org/2002/ws/sawsdl/

standards, etc.). Similarly, the WSMO services is modeled based on domain-specific requirements, specifications of back-end systems etc. The important aspect of the modeling phase is to define grounding from the semantic WSMO service (logical level) to the underlying WSDL description (physical level). In WSMO, grounding is defined for:

1. **WSMO service interface and WSDL messages.** This type of grounding specifies a reference for each used concept in the service interface to the input or output messages used in the WSDL.
2. **WSMO Ontologies and XML Schemata.** This type of grounding specifies so called *lifting* and *lowering* schema mapping for XML Schema and ontology respectively in order to perform instance transformations during invocation.

With respect to the modeling levels, we further distinguish two modeling approaches to semantic business services, namely *top-down approach* and *bottom-up approach*.

- **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. In this case services are implemented the way so that they can process semantic descriptions of ontologies and services. 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 are directly passed to the service where the lowering is performed[6]. Inversely, the lifting is performed in the service to the ontology and passed to the middleware where other processing follows according to the execution semantics definition. More information about WSMO grounding as described in this paragraph can be found in [6]. In section 3.1.3 we further show how top-down modeling approach is applied in our example.
- **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 specifications (using *loweringSchemaMapping* and *liftingSchemaMapping* extension attributes). The location of these mappings is resolved by the Communication service and executed during the invocation process.

[6]This could be done for example by serializing the WSML message to RDF/XML according to the schema defined in the WSDL

On result, the XML schema created from lowering is passed to the 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. More information about WSMO grounding using SAWSDL can be found in WSMO Grounding Working Draft[7].

3.1.3 Example of Business Services Modeling

In this section we show how the Blue and Mueller systems from the example in section 2.4 can be modeled using WSMO and WSML formalisms. In this example, we use the top-down approach to modeling of services (described in section 3.1.2), thus the modeling involves (1) *Web Service Creation* when underlying services as Web services with WSDL descriptions are created, and (2) *Semantic Web Service and Goals Creation* when semantic service and goal descriptions are created using WSMO. For our scenario, we create two services, namely PIP3A4 and CRM/OMS service (we model both systems as one business service).

- *Web Services Creation.* This step involves creation of Web services as adapters to existing systems, i.e. WSDL descriptions for these adapters including XML schema for messages, as well as binding the WSDL to implemented adapter services. In our scenario, we use two adapters: (1) PIP3A4 adapter and (2) CRM/OMS adapter. Apart from connecting Blue's system and Mueller's CRM and OMS systems to the middleware, and possibly resolving communication interoperability issues, they also incorporate lifting and lowering functionality for XML schema and ontologies as well as grounding definitions of WSMO services.

- *Semantic Web Services and Goals Creation.* In order to create Semantic Web services and Goals, the ontologies must be created (or reused) together with non-functional, functional and interface description of services. In addition, a grounding must be defined from the semantic (WSMO) descriptions to the syntactic (WSDL) descriptions. Semantic Web Services and Goals are described according to WSMO Service and WSMO Goal definitions respectively. We create a WSMO Goal as PIP3A4 service and WSMO Service as CRM/OMS service. Please note that WSMO Goal and WSMO Service have the same structural definition but differ in what they represent. The difference is in the use of defined capability and interface – WSMO Goal describes a capability and an interface required by a service requester whereas WSMO service describes a capability and an interface provided by a service provider. In our scenario, this task is performed by domain experts (ontology engineers) using WSMT. In this section we further elaborate on this step.

[7] http://www.wsmo.org/TR/d24/d24.2/

Creation of Ontologies and Grounding. One possible approach towards creation of ontologies would be to define and maintain one local domain ontology for Moon's B2B integration. This approach would further allow handling message level interoperability through the domain ontology when lifting and lowering operations would be defined from underlying message schemata to the domain ontology. Another option is the definition of independent ontologies by each vendor and its systems. In our case, these are different ontologies for RosettaNet and ontologies for CRM/OMS systems. The message level interoperability is then reached through mappings between used ontologies which are defined during design-time and executed during runtime. Although both approaches have their advantages and limitations, we will use the latter approach in our scenario. The main reason is to demonstrate mediators' aspects to integration of services which are available as independent and heterogeneous services.

We assume that all ontologies are not available up-front and they need to be created by an ontology engineer. The engineer takes as a basis the existing standards and systems, namely RosettaNet PIP3A4 and CRM/OMS schemata, and creates $PIP3A4$ and CRM/OMS ontologies respectively. When creating ontologies, the engineer describes the information semantically, i.e. with richer expressivity as opposed to the expressivity of underlying XML schema. In addition, the engineer captures the logic of getting from XML schema level to semantics introduced by ontologies by lifting and lowering rules executed on non-semantic XML schema and ontologies respectively. These rules are part of grounding definition between WSMO and WSDL descriptions and physically reside within adapters. In listing 3.1, example of the lifting rules and resulting WSML instance is shown for extract of a RosettaNet PIP3A4 message.

```
/* Lifting rules from XML message to WSML */
...
instance PurchaseOrderUID memberOf por#purchaseOrder
        por#globalPurchaseOrderTypeCode hasValue "<xsl:value-of select="dict:
            GlobalPurchaseOrderTypeCode"/>"
        por#isDropShip hasValue
          IsDropShipPo<xsl:for-each select="po:ProductLineItem">
              por#productLineItem hasValue ProductLineItem<xsl:value-of select="position()"/>
          </xsl:for-each>
        <xsl:for-each select="core:requestedEvent">
            por#requestedEvent hasValue RequestedEventPo
        </xsl:for-each>
        <xsl:for-each select="core:shipTo">
            por#shipTo hasValue ShipToPo
        </xsl:for-each>
        <xsl:for-each select="core:totalAmount">
            por#totalAmount hasValue TotalAmountPo
        </xsl:for-each>
...

/* message in WSML after transformation */
...
instance PurchaseOrderUID memberOf por#purchaseOrder
        por#globalPurchaseOrderTypeCode hasValue "Packaged product"
        por#isDropShip hasValue IsDropShipPo
        por#productLineItem hasValue ProductLineItem1
        por#productLineItem hasValue ProductLineItem2
        por#requestedEvent hasValue RequestedEventPo
        por#shipTo hasValue ShipToPo
```

```
        por#totalAmount hasValue TotalAmountPo
...
```

Listing 3.1: Lifting from XML to WSML

Creation of Functional and Non-Functional Descriptions. WSMO functional description (modeled as WSMO service capability) contains the formal specification of functionality that the service can provide, which is definition of conditions on service "inputs" and "outputs" which must hold before and after the service execution respectively. Functional description for our back-end systems contains conditions that input purchase order data must be of specific type and contain various information such as customer id, items to be ordered, etc. (this information is modeled as preconditions of the service). In addition, the service defines its output as purchase order confirmation as well as the fact that the order has been dispatched. Functional description of service is used for discovery purposes in order to find a service which satisfies the user's request. Non-functional properties contain descriptive information about a service, such as author, version or information about Service Level Agreements (SLA), Quality of Services (QoS), etc. In our example, we use the non-functional properties to describe user preference for service selection. In our case, the Blue company wants to buy and get shipped a product for the cheapest possible price which is encoded in the WSMO goal description.

Creation of Interfaces and Grounding. Interfaces describe service behavior, modeled in WSMO as (1) *choreography* describing how service functionality can be consumed by service requester and (2) *orchestration* describing how the same functionality is aggregated out of other services (in our example we only model choreography interfaces as we currently do not use WSMO service orchestration). The interfaces in WSMO are described using Abstract State Machines (ASM) defining rules modeling interactions performed by the service including grounding definition for invocation of underlying WSDL operations. In our architecture and with respect to types of interactions between service requester/provider and the middleware (see section 3.2.2), we distinguish two types of choreography definitions, namely *late-binding choreography* and *execution choreography*. Listing 3.2 shows a fragment depicting these two choreographies for the CRM/OMS service. The first choreography, marked as $DiscoveryLateBindingChoreography$, defines the rule how to get the quote for the desired product from purchase order request (in here, the concept $PurchaseQuoteReq$ must be mapped to corresponding information conveyed by the purchase order request sent by the Blue). This rule is processed during the service discovery and the quote information obtained is used to determine whether a concrete service satisfies the request (e.g. if the requested product is available which is determined through quote response). The second choreography, marked as $ExecutionChoreography$, defines how to get information about customer from the CRM system. Decision on which choreography should be used at which stage of execution (i.e. service discovery or conversation) is determined by the choreography namespace

(in the listing this namespace is identified using prefixes $dlb\#$ for discovery late-binding and $exc\#$ for execution respectively). In general, choreographies are described from the service point of view. For example, the rule in line 21 says that in order to send $SearchCustomerResponse$ message, the $SearchCustomerRequest$ message must be available. By executing the action of the rule ($add(SearchCustomerResponse)$), the underlying operation with corresponding message is invoked according to the grounding definition of the message which in turn results in receiving instance data from the Web service.

```
/* late−binding choreography for service discovery stage */
choreography dlb#DiscoveryLateBindingChoreography
    stateSignature
      in mu#purchaseQuoteReq withGrounding { ... }
      out mu#PurchaseQuoteResp withGrounding { ... }

        forall {?purchaseQuoteReq} with (
                ?purchaseRequest memberOf mu#PurchaseQuoteReq
                ) do
                        add( # memberOf mu#PurchaseQuoteResp)
          endForall
...

/* execution choeography for service execution stage */
choreography exc#ExecutionChoreography
    stateSignature
      in mu#SearchCustomerRequest withGrounding { ... }
      out mu#SearchCustomerResponse withGrounding { ... }

transitionRules MoonChoreographyRules
        forall {?request} with (
                        ?request memberOf mu#SearchCustomerRequest
                ) do
                        add(_# memberOf mu#SearchCustomerResponse)
        endForall
...
```

Listing 3.2: CRM/OMS Choreography

Creation of Ontology Mappings. Mappings between used ontologies must be defined and stored in the middleware repositories before execution. In listing 3.3, the mapping of $searchString$ concept of the CRM/OMS ontology to concepts $cusomterId$ of the $PIP3A4$ ontology is shown. The construct $mediated(X, C)$ represents the identifier of the newly created target instance, where X is the source instance that is transformed, and C is the target concept we map to [9]. Such format of mapping rules is generated from the ontology mapping process by the WSMT ontology mapping tool.

```
axiom mapping001 definedBy
  mediated(X, o2#searchString) memberOf o2#searchString :−
  X memberOf o1#customerId.
```

Listing 3.3: Mapping Rules in WSML

3.2 Process View

Processes reflect the behavior of the architecture through which stakeholders interact with the middleware and with business services. Similarly as in section 3.1, we distinguish two types of processes, namely (1) *middleware processes* and (2) *business processes*.

3.2.1 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 execution and conversation of services within a particular business process with data and process mediation applied where necessary. In section 3.2.2, the late-binding and execution phases is described in detail.

3.2.2 Middleware Processes

Middleware processes are designed to facilitate the integration of business services using middleware services. Middleware processes are described by a set of execution semantics. As described in previous sections, execution semantics define interactions of various middleware services which establish particular middleware processes for specific purposes, and which provide a particular functionality in a form of business processes to 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 operations.

For purposes of describing various forms of execution semantics, we distinguish following two phases of the middleware process.

1. **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.).
2. **Execution Phase** allows to execute and perform conversation of previously binded goal and a service by processing of their interfaces and with data and process mediation applied where necessary.

Both late-binding and execution phases follow a strong decoupling principle where services are described semantically and independently from requester's goal. In some cases

which are dependent on particular application domain, the validation of results from the late-binding phase needs to be performed. We discuss this validation later in section 3.2.4.

Middleware process which runs in the middleware system are initiated by a service requester and can be interfaced with the service requester/provider during run-time. Thus, the external integration between service requester/provider and the middleware and its middleware processes must be solved. For this purpose we define *communication styles*, *entrypoint types* and *interaction types* through which this external integration can be built.

- **Communication Styles**. 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.

- **Entrypoint Types**. 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.

- **Interaction Types**. There are two types of interactions between service requester/provider and the middleware, namely (1) *late-binding interactions*, and (2) *execution 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. Execution interactions allow to exchange information between service requester and service provider in order to facilitate conversation between them. Execution interactions happen through the middleware (which provide the added value of mediation functionality during service execution).

3.2.3 Example of External Integration

In figure 2.2, the above aspects of integration between service providers and service requesters are illustrated. The service requester (Blue) initiates the middleware process through invocation of the execution entrypoint (marked as a in the figure) and sending a goal representing its request (these is late-binding interaction marked as 1 in the figure).

In the middleware, the discovery component tries to find appropriate services from the service repository. During the discovery-time, the middleware might interact with potential services in order to retrieve additional information in order to decide on a match between requester's goal and the service (these are late-binding interactions marked as 2 in the figure). 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). These interactions must conform to the late-binding interface of respective WSMO service (see listing 3.2 for example of late-binding interface in section 3.1.3). Finally, the execution and conversation between service requester and discovered services is performed (these are execution interactions marked as 3 in the figure).

3.2.4 Execution Semantics

In this section we define a set of execution semantics which facilitate so called *goal-based invocation of services*. In particular, as depicted in figure 3.3 we define three basic types of execution semantics, called *AchieveGoal* execution semantics, *RegisterGoal* execution semantics, and *Optimized AchieveGoal* execution semantics. The execution semantics described here involve both late-binding and execution phases.

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). 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 3.3). 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.

AchieveGoal Execution Semantics. For the execution semantics *AchieveGoal* (see branches 1.1 and 1.2 in the figure 3.3), (1) web services discovery, (2) service discovery, (3) service selection, (4) validation and (5) execution 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

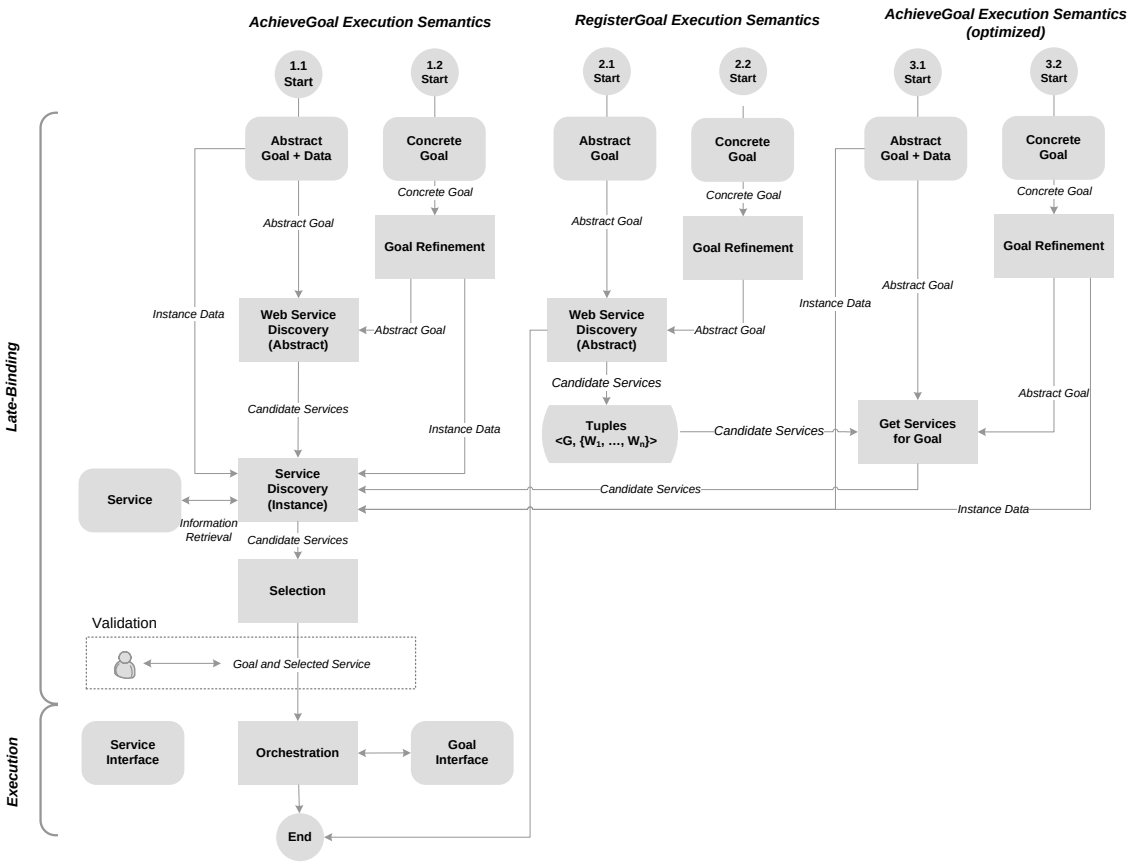

Figure 3.3: Execution Semantics

disjnontness [5]. When the match is found[8], 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 2.2. 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). The process of web service discovery, followed by service discovery may or may not lead to the discovery of service that can fully meet the users functional and non-functional requirements, when it comes to selecting a service on behalf of the user. For this purpose, it is important to ensure that user can validate the results from the late-binding phase, e.g. approve the selected service, relax his/her requirements, etc. This process is part of the general validation of the late-binding phase as described in the next paragraph. After that, the execution is started between the selected service and the goal

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

by processing of goal and service interfaces.

Validation Late-binding phase enables to increase automation in service provisioning processes. Services used to deliver overall functionality are not known prior to the execution time however it is important to ensure that results from the late-binding phase could be verified before their adoption. In SESA, the validity of the late-binding phase can be ensured using combination of *user approval* and (2) *service analysis* methods. The service analysis allows to provide information based on the service usage and quality answering questions such as "how often does the service fail?", "how often given service is used in the given context?", "how often the service was aborted by the user?" etc. With help of this information, late-binding phase can be approved or refined by a user. The user is notified about the late-binding results, e.g. what he/she is searching for cannot be fully satisfied when he/she has the possibility to relax some of his/her functional or non-functional requirements. This can be especially important when many services are available but with a lower quality than desired by the user. By selecting less functionality the user may find services of higher quality and on the other hand by relaxing the non-functional quality the user may find services with more functionality. Choosing a strategy for validation of the late-binding phase strongly depends on application domain where SESA is deployed. In highly-sensitive domains such as eHealth where results of late-binding phase are crucial with their real-world effects, the manual validation of the late-binding phase is necessary. Ultimately, the late-binding might not necessarily be validated by user (e.g. in domains where the risk margin can be higher such as eTourism) when analysis of late-binding results can be done automatically based on advertised functionality and QoS properties of services.

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 $< G, \{W_1, ..., W_n\} >$ 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 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 *RegisterGoal* 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.

Execution. AchieveGoal execution semantics described above ends with orchestration between bound service requester and service provider. In chapter 4 we define this phase in detail.

3.3 Technology View

In this section we describe the technology for the middleware system, in particular the technology and implementation of *execution management service* of the middleware Core. The execution management service is responsible for the management of a platform and for orchestrating the individual functionality of middleware services according to defined execution semantics through which it facilitates the overall operation of the middleware. In here, middleware services are implemented as functional components (also called application components) of the middleware system.

The execution management service takes the role of component management and coordination, inter-component messaging and configuration of execution semantics. In particular, 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 execution management service implements the middleware kernel utilizing Java Management Extensions (JMX) as described in [4]. It is responsible for handling following three main functional requirements: (1) *Management*, (2) *Communication and Coordination*, and (3) *Execution Semantics*.

3.3.1 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 3.4, an overview of the infrastructure provided by the management execution service 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 functional component. In here, the management agent plays the role of a driver which is directly

built into the application. The execution management service 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.

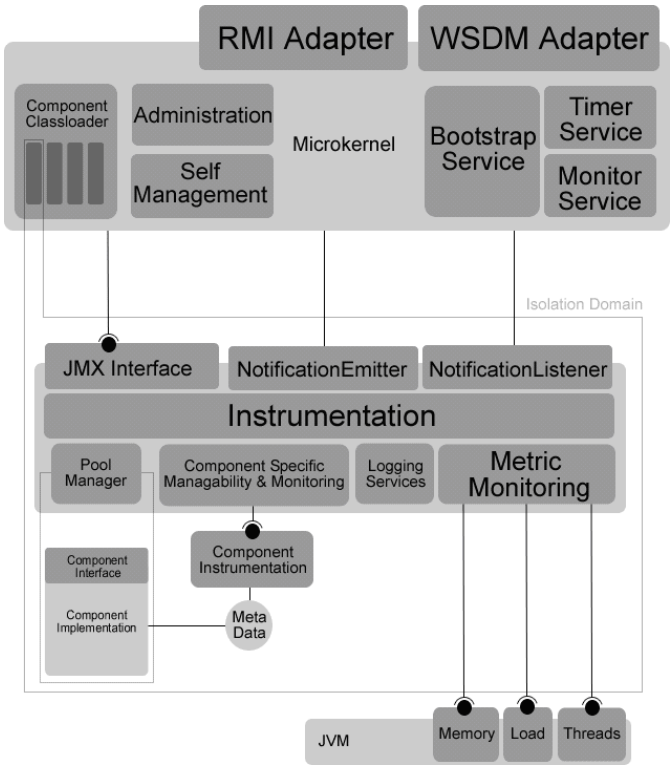

Figure 3.4: Component Management in the Middleware Execution Management Service

Similarly as in state-of-the art middleware systems, the execution management service hosts a number of subsystems that provide services to components and enable inter-component communication. In addition, the service provides a number of other services including *pool management* which takes care of handling component instances, and logging, transport and lifecycle services. The execution management service 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 execution management service 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 execution management service 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 is achieved by propagating requests within the federation via proxies, broadcasts or directories. A federation thus consists of a number of execution management services, each of them operating a kernel per one machine and hosting a number of functional components.

3.3.2 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 3.5, 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.

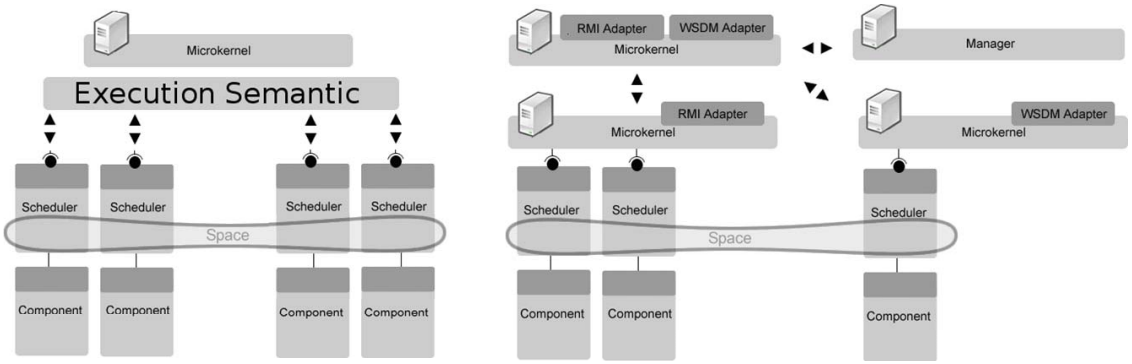

Figure 3.5: Execution Semantics, Communication and Coordination in the Middleware

The Tuple Space enables communication between distributed components running on both local as well as remote machines while at the same time 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[3] 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 execution management, components implementations are separated from communication. This infrastructure is made available to each component implementation during instantiation of the component carried out by the execution management service during the bootstrap process. 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.

3.3.3 Execution Semantics

Execution Semantics enable a combined execution of functional components as illustrated in figure 3.5. Execution semantics defines the logic of the middleware which realize the middleware behavior. The execution management service 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 execution semantics. The execution management service 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, execution management service takes care of the execution semantics lifecycle, management and monitoring.

Figure 3.6 depicts how components 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. The wrappers are generated and managed by the execution management service 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.

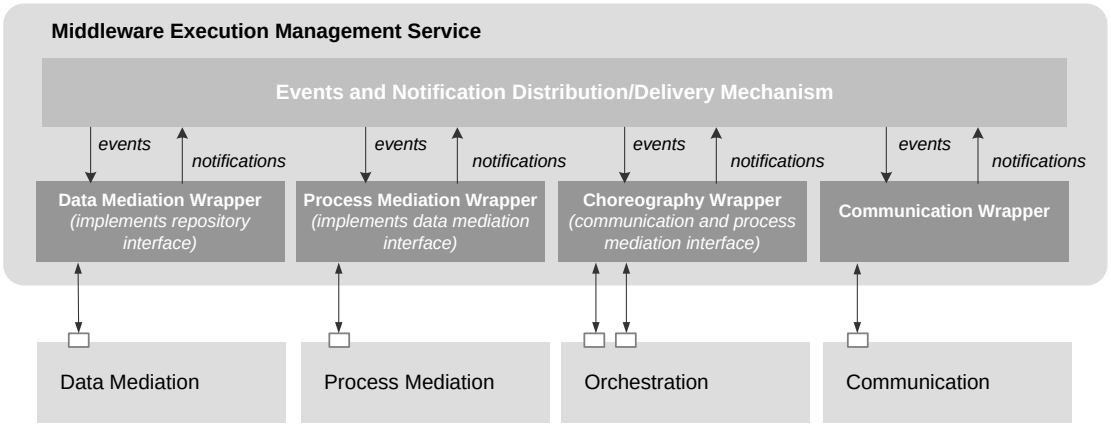

Figure 3.6: Component's Wrappers and Event Messaging

Chapter 4

Execution Semantics for Execution Phase

In this section we define the exacution semantics for the execution phase of the SESA. This execution semantics is triggered as a result of the late-binding phase as described in section 3.2.2.

4.1 Definitions

4.1.1 Information Semantics

Information Semantics is the formal definition of some domain knowledge used by the service in its *input* and *output* messages. We describe the information semantics as an ontology defining the terminology of the domain together with a knowledge base as the instantiation of the ontology. Formally, the information semantics is a structure

$$O = (C, R, E, I) \tag{4.1}$$

with a set of classes (unary predicates) C, a set of relations (binary and higher-arity predicates) R, a set of explicit instances of C and R called E (extensional definition), and a set of axioms called I (intensional definition) that describe how new instances are inferred.

4.1.2 Behavioral Semantics

Behavioral Semantics is a description of the public and the private behavior of a service. For our work we only use the public behavior (called choreography[1]) as a description of a protocol which must be followed by a client in order to invoke the service. We describe a choreography as a public process, i.e. from the service point view, all the messages are sent in to the service from the network and all the messages are sent from the service out to the network. We define the choreography X (read: chi) of the service using the Abstract State Machine (ASM) as[12]

$$X = (\Sigma, L), \tag{4.2}$$

where $\Sigma \subseteq (\{x\} \cup C \cup R \cup E)$ is the signature of symbols, i.e. variable names $\{x\}$ or identifiers of elements from C, R, E of some information semantics O; and L is a set of rules. Further, we distinguish dynamic symbols denoted as Σ_I (input), and Σ_O (output) and static symbols denoted as Σ_S. While the static symbols cannot be changed by the service invocation, the dynamic symbols correspond to input and output data of the service which can be changed by the invocation. Each rule $r \in L$ defines a state transition $r : r^{cond} \rightarrow r^{\mathit{eff}}$ where $cond$ is defined as an expression in logic $\mathcal{L}(\Sigma_I \cup \Sigma_S)$ which must hold in a state before the transition is executed; eff is defined as an expression in logic $\mathcal{L}(\Sigma_I \cup \Sigma_O \cup \Sigma_S)$ describing how the state changes when the transition is executed.

4.1.3 Grounding

Grounding defines a link between semantic descriptions of services and the underlying technology used for the services invocation (such as how and where the service can be accessed). Although the semantic descriptions are independent on the underlying technology, we use grounding to WSDL for on-the-wire message serialization (WSDL binding), physical Web service access (WSDL service and endpoint) and communication (SOAP).

For purposes of grounding definition for a WSDL description we denote the WSDL schema as S and the WSDL interface as N. Further, we denote $\{x\}_S$ as a set of all element declarations and type definitions of S, and $\{o\}_N$ as a set of all operations of N. Each operation $o \in \{o\}_N$ may have one input message element $m \in \{x\}_S$ and one output message element $n \in \{x\}_S$.

There are two types of grounding used for information and behavioral semantics. The first type of grounding specifies references between input/output symbols of a choreography $X = (\Sigma, L)$ and input/output messages of respective WSDL operations $\{o\}_N$ with schema S. We define this grounding as

[1]Please note, that our notion of the choreography is different from the one used by the Web Service Choreography Description Language (WS-CDL) `http://www.w3.org/TR/ws-cdl-10/`

$$ref(c, m) \tag{4.3}$$

where $m \in \{x\}_S$, $c \in \Sigma$ and ref is a binary relation between m and c. Further, m is the input message of operations in $\{o\}_N$ if $c \in \Sigma_I$ or m is the output message of operations in $\{o\}_N$ if $c \in \Sigma_O$.

The second type of grounding specifies transformations of data from schema S to ontology $O = (C, R, E, I)$ called lifting and vice-versa called lowering. We define this grounding as

$$lower(c_1) = m \qquad \text{and} \qquad lift(n) = c_2, \tag{4.4}$$

where $m, n \in \{x\}_S$, $c_1, c_2 \in (C \cup R)$, $lower$ is a *lowering transformation function* transforming the semantic description c_1 to the message m, and $lift$ is a *lifting transformation function* transforming the message n to the semantic description c_2. Please note that both definitions in Eq. 4.3 and Eq. 4.4 are associated either with WSDL or semantic descriptions. For example, [6] defines the grounding associated with WSMO semantic service model and [12] describes the grounding associated with WSDL descriptions using the Semantic Annotations for WSDL and XML Schema (SAWSDL) specifications[7].

Both types of grounding definitions are used when processing the choreography rules and performing the communication with the service (see Section 4.2) while following the underlying definition of WSDL operations and their Message Exchange Patterns (MEPs). Table 4.1 shows basic choreography rules for four basic WSDL 2.0 MEPs[2], (*in-out*, *in-only*, *out-only*, *out-in*) and corresponding WSDL operations. In here, the symbols $c_1, \ldots, c_6$ refer to identifiers of semantic descriptions defined as part of input or output state signature Σ_I or Σ_O of some choreography X (see Eq. 4.2), the symbols `msg1`, ..., `msg6` refer to some XML Schema elements used for input/output messages of operations, $ref(c, m)$ denotes the existence of grounding definition between a semantic description c and a message m (see Eq. 4.3), and `w` is the prefix for URI `http://www.w3.org/ns/wsdl`. Please note that a complex rule may exist in the choreography covering more then one invocation and thus combining multiple MEPs in one rule.

4.1.4 Data Mediation

When the information semantics of the two services is different, i.e. different ontologies are used, the communication cannot take place and the data mediation needs to be performed. The data mediation transforms every incoming message from the terms of the sender's information semantics (the source) into the terms of the receiver's information semantics (the target).

[2] `http://www.w3.org/TR/wsdl20-adjuncts/#meps`

MEP and Rule	WSDL Operation

in-out:
if c_1 then $add(c_2)$
$c_1 \in \Sigma_I$, $ref(c_1, \texttt{msg1})$
$c_2 \in \Sigma_O$, $ref(c_2, \texttt{msg2})$
	`<operation name="oper1" pattern="w:in-out">`
	`  <input messageLabel="In" element="msg1"/>`
	`  <output messageLabel="Out" element="msg2"/>`
	`</operation>`

in-only:
if c_3 then *no action*
$c_3 \in \Sigma_I$, $ref(c_3, \texttt{msg3})$
	`<operation name="oper2" pattern="w:in-only">`
	`  <input messageLabel="In" element="msg3"/>`
	`</operation>`

out-only:
if $true$ then $add(c_4)$
$c_4 \in \Sigma_O$, $ref(c_4, \texttt{msg4})$
	`<operation name="oper3" pattern="w:out-only">`
	`  <output messageLabel="Out" element="msg4"/>`
	`</operation>`

out-in:
if $true$ then $add(c_5)$
if $c_5 \wedge c_6$ then *no action*
$c_5 \in \Sigma_O$, $ref(c_5, \texttt{msg5})$
$c_6 \in \Sigma_I$, $ref(c_6, \texttt{msg6})$
	`<operation name="oper4" pattern="w:out-in">`
	`  <output messageLabel="Out" element="msg5"/>`
	`  <inpput messageLabel="In" element="msg6"/>`
	`</operation>`

Table 4.1: MEPs, Rules and WSDL operations

The agent performing data mediation has to automatically perform the transformation of the exchanged messages. Since the interoperability problems can greatly vary in their nature and severity, automatic solution for the detection and solving of data mismatches are not feasible in a business scenario due to the lower-then-100% precision and recall of the existing methods[3]. As a consequence, alignments between heterogenous ontologies have to be created at design-time and used by the data mediation engine at run-time.

An alignment consists of a set of mappings expressing the semantic relationships that exist between the two ontologies. Technically, the mappings are expressed as rules which concretely specify the semantics of mappings present in alignments. In particular, a mapping can specify that classes from two ontologies are equivalent while corresponding rules use logical expressions to unambiguously define how the data encapsulated in an instance of one class can be encapsulated in instances of the second class. Formally, we define an alignment A between two ontologies $O_s = (C_s, R_s, E_s, I_s)$ and $O_t = (C_t, R_t, E_t, I_t)$ as

$$A_{s,t} = (O_s, O_t, \Phi_{s,t}) \tag{4.5}$$

where $\Phi_{s,t}$ is the set of mappings m of the form

$$m =< \varepsilon_s, \varepsilon_t, \gamma_{\varepsilon_s}, \gamma_{\varepsilon_t} > \tag{4.6}$$

[3]The "*Ontology Alignment Evaluation Initiative 2006*" [2] shows that the best five systems' scores vary between 61% and 81% for precision and between 65% and 71% for recall.

where ε_s, ε_t represent the mapped entities from the two ontologies while γ_{ε_s}, γ_{ε_t} represent restrictions (i.e. conditions) on these entities such as $\varepsilon_s \in C_s \cup R_s$, $\varepsilon_t \in C_t \cup R_t$ while γ_{ε_s} and γ_{ε_t} are expressions in $\mathcal{L}(C_s \cup R_s \cup E_s)$ and $\mathcal{L}(C_t \cup R_t \cup E_t)$, respectively.

Please note, that in order to execute the mappings, they need to be grounded to executable rules expressed in a logical language for which a reasoning support is available. Using this grounding, the reasoner becomes the execution engine of these rules. We implement this grounding using the WSML language. Consequently, the set of rules $\rho_{s,t} = \Phi^G_{s,t}$ is obtained by applying the grounding G to the set of mappings Φ. Every mapping rule $mr \in \rho_{s,t}$ has the following form:

$$mr : \bigwedge_{i=1..n}^{\{x\}} mr_i^{head} \leftarrow \bigwedge_{i=1..n}^{\{x\}} mr_i^{body} \tag{4.7}$$

where

$$\begin{aligned} mr^{head} \in\ & \{x' \textbf{ instanceOf } \varepsilon \mid \varepsilon \in C_t \textbf{ and } x' \in \{x\}\} \cup \\ & \{\varepsilon(x', x'') \mid \varepsilon \in R_t \textbf{ and } \varepsilon(x', x'') \in E_t \textbf{ and } x', x'' \in \{x\}\} \end{aligned} \tag{4.8}$$

$$\begin{aligned} mr^{body} \in\ & \{x' \textbf{ instanceOf } \varepsilon \mid \varepsilon \in C_s \textbf{ and } x' \in \{x\}\} \cup \\ & \{\varepsilon(x', x'') \mid \varepsilon \in R_s \textbf{ and } \varepsilon(x', x'') \in E_s \textbf{ and } x', x'' \in \{x\}\} \cup \\ & \{\gamma_s \mid \gamma_s \in \mathcal{L}(C_s \cup R_s \cup E_s \cup \{x\})\} \cup \\ & \{\gamma_t \mid \gamma_t \in \mathcal{L}(C_t \cup R_t \cup E_t \cup \{x\})\} \end{aligned} \tag{4.9}$$

A mapping rule is formed of a head and a body. The head is a conjunction of logical expressions over the target elements and it constructs the instances of the target ontology which represent the result of the mediation. The body is formed of a set of logical expressions over the source entities which represent the data to be mediated, plus a set of logical expressions representing conditions over both the source and the target data. In the above definitions, $\{x\}$ stands for the set of variable used by the mapping rule and x' and x'' are two particular variables. It is important to mention that the variables are used in such a way to assure that there are no unsafe rules generated (i.e. in the head there are no variables that do not appear in the body).

4.1.5 Process Mediation

Process Mediation handles the interoperability issues which occur in descriptions of choreographies of the two services. In [1], Cimpian defines five process mediation patterns:

a. Stopping an unexpected message: when one service sends a message which is not expected by the other service, the mediator stops the message.

b. Inversing the order of messages: when one service sends messages in a different order than the the other service expects them to receive, the mediator ensures that messages are supplied in proper order.

c. Splitting a message: when a service sends a message which the other service expects to receive in multiple different messages, the mediator splits the message and ensures that all messages are supplied to the service.

d. Combining messages: when a service expects to receive a message which is sent by the other service in multiple different messages, the mediator combines those messages and ensures that the combined message is supplied to the service.

e. Generating a message: when one service expects to receive a message which is not supplied by the other service, mediator generates the message and supplies the message to the service.

The patterns are implemented using an algorithm by processing the both choreographies, i.e. evaluating choreography rules and the information semantics of the both services. In sections 4.2 and 4.3 we show how the algorithm fulfils the patterns (a) – (d). In order to fulfill the pattern (e), the algorithm should be aware of the intention of the messages. For example, if the algorithm is able to distinguish control interactions (e.g. acknowledgements) among all the interactions happening between the both services, it could generate an acknowledgment message (assuming the algorithm would be able to assess that a message to be acknowledged was successfully received by the other service). Since we do not give semantics to message interactions, we currently do not address the pattern (e) in our work.

4.2 Algorithm

The algorithm for the execution model manages the conversation between two services with applied data and process mediation. Each such a service contains description of information and behavioral semantics, WSDL definition and the grounding according to the definitions in the previous section. These services are usually supplied as a result from the late-binding phase. The Figure 4.1 depicts the main steps of the execution phase. The algorithm requires inputs and uses internal structures as follows:

Input:

- Service W_1 and service W_2. Each such a service W contains the ontology (information semantics) $W.O$ (Eq. 4.1), the choreography $W.X$ (Eq. 4.2) with set of rules $W.X.L$, WSDL description and grounding (Eq. 4.3, 4.4). In addition, for a rule $r \in W.X.L$, the condition r^{cond} is a logical expression with set of semantic descriptions $\{c\}$, and the effect r^{eff} is a logical expression with set of actions $\{a\}$.

For each element a we denote its action name as $a.action$ with values $delete$ or add and a semantic description as $a.c$.

- Mappings Φ between $W_1.O$ and $W_2.O$.

Uses:

- Symbols M_1 and M_2 corresponding to the processing memory of the choreography $W_1.X$ and $W_2.X$ respectively (a memory M is a populated ontology $W.O$ with instance data). The content of each memory M determines at some point in time a state in which a choreography $W.X$ is. In addition, each memory has methods $M.add$ and $M.remove$ allowing to add or remove data to/from M and a flag $M.modified$ indicating whether the memory was modified. The flag $M.modified$ is set to $true$ whenever the method $M.add$ or $M.removed$ is used.

- Symbols D_1 and D_2 corresponding to the set of data to be added to the memory M_1 and M_2 after the choreography is processed. Each D has a method $D.add$ for adding new data to the set.

- A symbol A corresponding to all actions to be executed while processing the choreography. Each element of A has the same definition as the element of the rule effect r^{eff}. A has methods $A.add$ and $A.remove$ for adding and removing actions to/from the set.

- A symbol o corresponding to a WSDL operation of a service and symbols m, n corresponding to some XML data of the message (input or output) of the operation o.

States 1, 2, 7: Initialize, Control, End

1: $M_1 \leftarrow \emptyset$; $M_2 \leftarrow \emptyset$
2: **repeat**
3: $M_1.modified \leftarrow false$; $M_2.modified \leftarrow false$
4: $D_1 \leftarrow processChoreography(W_1, M_1)$
5: $D_2 \leftarrow processChoreography(W_2, M_2)$
6: **if** $D_1 \neq \emptyset$ **then**
7: $D_m \leftarrow mediateData(D_1, W_1.O, W_2.O, \Phi)$
8: $M_1.add(D_1)$; $M_2.add(D_m)$
9: **end if**
10: **if** $D_2 \neq \emptyset$ **then**
11: $D_m \leftarrow mediateData(D_2, W_2.O, W_1.O)$
12: $M_1.add(D_m)$; $M_2.add(D_2)$
13: **end if**
14: **until not** $M_1.modified$ **and not** $M_2.modified$

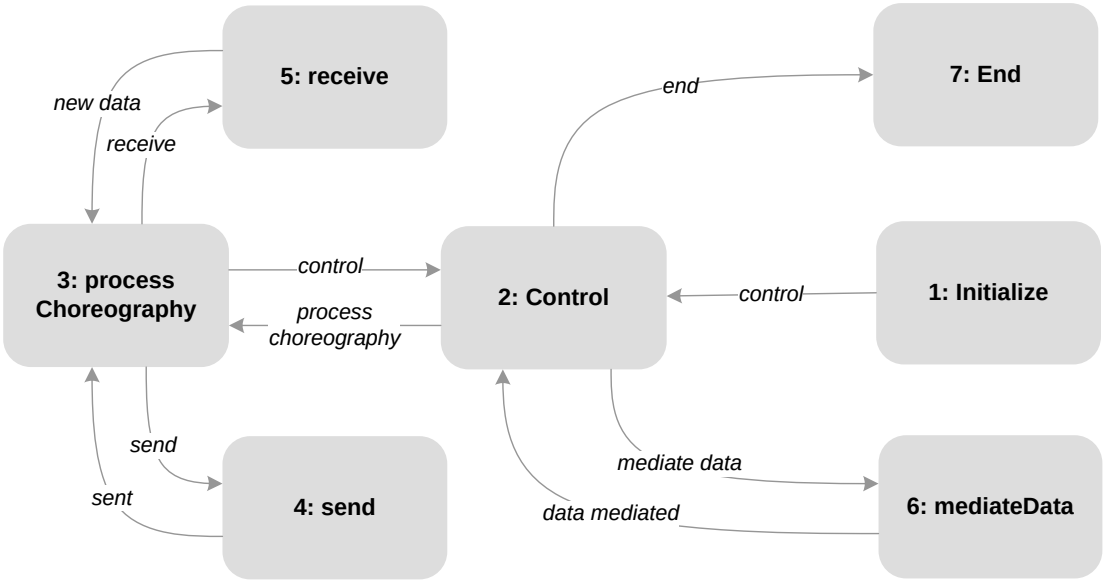

Figure 4.1: Control State Diagram for the Execution Model

After the initialization of the processing memory M_1 and M_2 (line 1), the execution gets to the control state when the whole process is managed. It can process choreographies (state 3), mediate the data (state 6) or end the execution (state 7). The execution ends when no modifications of the processing memories M_1 or M_2 has occurred.

State 3: $D = processChoreography(W, M)$

1: $A \leftarrow \emptyset$; $D \leftarrow \emptyset$
2: {*Performing rule's conditions and sending data*}
3: **for all** r in $W.X.L : holds(r^{cond}, M)$ **do**
4: $A.add(r^{eff})$
5: **for all** c in $r^{cond} : c \in W.X.\Sigma_I$ **do**
6: $send(c, W)$
7: **end for**
8: **end for**
9: {*Performing delete actions*}
10: **for all** a in $A : a.action = delete$ **do**
11: $M.remove(a.c)$
12: $A.remove(a)$
13: **end for**
14: {*Receiving data and performing add actions*}
15: **while** $A \neq \emptyset$ **do**
16: $c \leftarrow receive(W)$
17: **if** $c \neq null$ **then**

18: **for all** a in A: $(a.action = add$ and $a.c = c)$ **do**
19: $D.add(c)$
20: $A.remove(a)$
21: **end for**
22: **end if**
23: **end while**
24: **return** D

The algorithm executes each rule of the choreography which condition holds in the memory by processing its condition and effect, i.e. the algorithm collects all data to be added to the memory or removes existing data from the memory. The whole process is divided into three major steps as follows.

- **Performing rule's conditions and sending data (lines 2-8):** the effect of the rule which condition satisfies the content of the memory (line 3) is added to the set of effects A (line 4). Then, for each input symbol of the rule's condition (line 5), the algorithm sends the data to the service (line 6, see State 4).
- **Performing delete actions (lines 9-13):** all effects with $delete$ action are performed, the data of the effect is removed from the memory (line 11) while such effects are removed from A (line 12).
- **Receiving data and performing add actions (lines 14-24):** When there are effects to be processed in A and the new data is received from the service (line 16), it is checked if the new data corresponds to some of the add effect from A. In this case, the data is added to the set D (line 19) and the effect is removed from A (line 20).

The result of the algorithm is the set D containing all new data to be added to the memory M. The actual modification of the memory M with the new data is performed in State 2. The algorithm assumes that definition of the choreography rules are consistent with WSDL operations and their MEPs while at the same time no failures occur in services. In lines 14-23 the algorithm waits for every message to be received from the service for every add action of the rule's effect. If the definition of the rules was not consistent with WSDL description, the algorithm would either ignore the received message which could in turn affect the correct processing of the choreography (in case of missing add action) or wait infinitely (in case of extra add action or a failure in a service). For the latter, the simplest solution would be to introduce a timeout in the loop (lines 14-23), however, we do not currently handle these issues in the algorithm. They will be the subject of our future work.

State 4: $send(c, W)$

1: $m \leftarrow lower(c)$
2: **for all** o of which m is the input message **do**

3: send m to W
4: **end for**

In order to send the data c the algorithm first retrieves a corresponding message definition according to the grounding and transforms c to the message m using the lowering transformation function (line 1). Then, through each operation of which the message m is the input message, the algorithm sends the m to the service W.

State 5: $c = receive(W)$

1: **if** receive m from W **then**
2: $c \leftarrow lift(m)$
3: **return** c
4: **else**
5: **return** $null$
6: **end if**

When there is a new data from the service W, the data (message m in XML) is lifted to the semantic representation using lifting transformation function associated with the message (line 2), and the result is returned. In the opposite case, the $null$ is returned.

State 6: $c_m = mediateData(c, O_s, O_t, \Phi)$

1: $\varepsilon \leftarrow getTypeOf(c)$
2: $\varepsilon_m \leftarrow null$
3: **for all** $m =< \varepsilon_s, \varepsilon_t, \gamma_{\varepsilon_s}, \gamma_{\varepsilon_t} > \in \Phi$ where $\varepsilon = \varepsilon_s$ **do**
4: **if** $isMoreGeneral(\varepsilon_t, \varepsilon_m)$ **then**
5: $\varepsilon_m \leftarrow \varepsilon_t$
6: **end if**
7: $\rho \leftarrow \rho \cup \{m_G\}$
8: **end for**
9: **if** $\varepsilon_m = null$ **then**
10: **return** $null$
11: **end if**
12: $c_m \leftarrow getDataForType(\varepsilon_m, \rho)$
13: **return** c_m

The first step in the mediation process is to determine the type of the data to be mediated. If this data is a concept instance the algorithm determines its concept. After that, the set of mappings is navigated in order to determine the type of the target, mediated data. Since there could be more mappings from a given source entity to the several other target entities, it is necessary to determine the most general entity to mediate the source data to. Also while traversing the set of mappings, each of them is grounded to WSML and transformed in a set of logical mapping rules. Finally, by using a reasoner engine all the data of the selected target type is retrieved based on the source data and the set of

mapping rules.

From the implementation point of view, several optimizations could be applied to this algorithm. First, the mappings rules could be cached in order to avoid their regeneration every time when a new request for data mediation is coming. Second only the mappings and mapping rules that refer to the input source data could be processed in order to reduce the volume of rules that need to be evaluated.

4.3 Discussion on Data and Process Mediation

The data mediation ensures that all new data coming from one service is translated to the other's service ontology. Thus, no matter from where the data originates the data is always ready to use for the both services. From the process mediation point view, the data mediation also handles the splitting of messages (pattern c) and combining the messages (pattern d). Since the mediated data is always added to the both memories (see State 2, lines 8, 12 and the next paragraph for additional discussion) the patterns a) and b) are handled automatically through processing of the choreography rules. In particular, the fact that a message will be stopped (pattern a) means that the message will never be used by the choreography because no rule will use it (the message remains in the memory until the end of the algorithm). In addition, the order of messages will be inverted (pattern b) as defined by the choreography rules and the order of ASM states in which conditions of rules hold. This means that the algorithm automatically handles the process mediation with help of data mediation through rich description of choreographies when no central workflow is necessary for that purpose.

In our algorithm we always add all the data to the both choreographies and not only the data which could be of *potential use*, i.e. the data could be used when evaluating a rule's condition. However, since we use the language which allows for the intentional definitions (axioms) which are part of the information semantics and the memory, the data might affect the evaluation of the rule indirectly through such axioms. The evaluation of the potential use of data would thus require a logical reasoning and would influence the scalability and the processing time. On the other hand, we do not expect a significant overhead when storing such additional data, however, we leave the evaluation for the future work.

Chapter 5

Evaluation

In order to evaluate the architecture and its implementation we focus on major aspects of semantic-based systems which lie in flexibility and adaptivity of integration processes of heterogeneous services. With this respect, the presented in this article and its implementation together with implementation of the example from section 2.4 has been evaluated in the SWS Challenge[1] series of workshops. The SWS Challenge is an initiative led by a Semantic Web Services providing a standard set of increasingly difficult problems, based on industrial specifications and requirements. Entrants to the SWS Challenge are peer-evaluated to determine if semantically-enabled integration approaches reduce costs of establishing and maintaining the integration between independent systems. In each SWS challenge workshop, the entrants first address introduced initial scenario of particular problem (e.g. mediation, discovery) in a testing environment prepared by the SWS Challenge organizers. The organizers then introduce some changes to back-end systems of the testing environment when the adaptivity of solutions is evaluated – solutions should handle introduced changes by modification of declarative descriptions rather than code changes. This evaluation is done by a methodology, developed by the SWS Challenge organizers and participants, which identifies following so called *success levels*.

- *Success level 0* indicates a minimal satisfiability level, where messages between middleware and backend systems are properly exchanged in the initial scenario.
- *Success level 1* is assigned when changes introduced in the scenario require code modifications and recompilation.
- *Success level 2* indicates that introduced changes did not entail any code modifications but only declarative parts had to be modified.
- *Success level 3* is assigned when changes did not require either modifications to code or the declarative parts, and the system was able to automatically adapt to the new conditions.

[1] http://www.sws-challenge.org

Our implementation of the scenario from section 2.4 and its evaluation has been done in two stages separating (1) mediation problem (data and process mediation) and (2) discovery problem of the example described in section 2.4. The implementation of the mediation problem has been evaluated in Budva, Montenegro[2] where it has been shown how it adapts to data and process changes in the back-end systems. For data mediation we had to make some changes in code due to forced limitations of existing data mediation tools (success level 1). For process mediation we needed to change only description of service interfaces (choreographies) according to the changes in back-end systems (success level 2). The implementation of the discovery problem has been evaluated in Athens, USA[3] where it has been shown how it successfully addresses the discovery of services based on location, weigth and price, scoring success level 2. All this information had to be gathered from the service requester during discovery-time through late-binding discovery service interface and integrated with the discovery process. No changes in code or in the descriptions of the services were required only the Goal request had to be changed .

[2]http://sws-challenge.org/wiki/index.php/Workshop_Budva
[3]http://sws-challenge.org/wiki/index.php/Workshop_Athens

Chapter 6

Conclusion and Future Work

This work is the second version of the architecture for the Semantic Web Services which aims at establishing the grounds for joint work on Semantic Service Oriented Architecture (SESA) and in particular exploited in the OASIS Semantic Execution Environment Technical Committee (OASIS SEE TC). In the second deliverable we have revised the first version of the architecture based on the communication with various groups involved in the architecture within various EU funded projects and strengthen this way the architecture grounds. 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. We

In our future work 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 work on the lightweight descriptions of services with focus on SAWSDL W3C recommendation as well as augmantation of service descriptions for REST services.

Bibliography

[1] Emilia Cimpian and Adrian Mocan. Wsmx process mediation based on choreographies. In *Business Process Management Workshops*, pages 130–143, 2005.

[2] Jérôme Euzenat, Malgorzata Mochol, Pavel Shvaiko, Heiner Stuckenschmidt, Ondřej Šváb, Vojtěch Svátek, Willem Robert van Hage, and Mikalai Yatskevich. Results of the Ontology Alignment Evaluation Initiative 2006. In *Proceeding of International Workshop on Ontology Matching (OM-2006)*, volume 225, pages 73–95, Athens, Georgia, USA, November 2006. CEUR Workshop Proceedings.

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

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

[5] Uwe Keller, Rubén Lara, Holger Lausen, Axel Polleres, and Dieter Fensel. Automatic location of services. In *ESWC*, pages 1–16, 2005.

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

[7] Jacek Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, 11(6), 2007.

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

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

[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.

[12] Tomas Vitvar, Jacek Kopecky, and Dieter Fensel. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web. In *ECOWS*, 2007.

[13] Tomas Vitvar, Adrian Mocan, Mick Kerrigan, Michal Zaremba, Maciej Zaremba, Matthew Moran, Emilia Cimpian, Thomas Haselwanter, and Dieter Fensel. Semantically-enabled service oriented architecture: Concetps, technology and application. *Service Oriented Computing and Applications*, 2(1):129–154, 2007.

[14] Tomas Vitvar, Maciej Zaremba, and Matthew Moran. Dynamic discovery through meta-interactions with service providers. In *ESWC*, 2007.

[15] Tomas Vitvar, Michal Zaremba, Matthew Moran, Maciej Zaremba, and Dieter Fensel. Sesa: Emerging technology for service centric environments. *IEEE Software*, 24(6):56–67, 2007.

[16] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A qos-aware selection model for semantic web services. In *ICSOC*, pages 390–401, 2006.