Service Oriented Computing and Applications May 2007, DOI: 10.1007/s11761-007-0009-9
The editor's copy of the article, published version available at http://dx.doi.org/10.1007/s11761-007-0009-9

Semantically-enabled Service Oriented Architecture : Concepts, Technology and Application

Tomas Vitvar[1], Adrian Mocan[2], Mick Kerrigan[2], Michal Zaremba[1,2], Maciej Zaremba[1], Matthew Moran[1], Emilia Cimpian[2], Thomas Haselwanter[2], Dieter Fensel[2]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
e-mail: `firstname.lastname@deri.org`
[2] Digital Enterprise Research Institute, University of Innsbruck, Austria
e-mail: `firstname.lastname@deri.org`

Received: 28 February 2007 / Revised version: 11 April 2007

Abstract Semantically-enabled Service Oriented Architecture focused around principles of service orientation, semantic modeling, intelligent and automated integration defines grounds for a cutting-edge technology which enables new means to integration of services, more adaptive to changes in business requirements which occur over systems' lifetime. We define the architecture starting from a global perspective and through Web Service Modeling Ontology as its semantic service model we narrow down to its services, processes and technology we use for the reference implementation. On a B2B integration scenario we demonstrate several aspects of the architecture and further describe the evaluation of the implementation according to a community-agreed standard evaluation methodology for semantic-based systems.

Key words SOA, Semantic web, Web services, Semantic Web Services, B2B Integration

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 systems 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.

In this article, we describe the Semantically-enabled Service Oriented Architecture (SESA), building on a number of governing principles which underpins the analysis, design and implementation of architecture, its middleware, services and processing logic. We first define the architecture from the global perspective and through underlying service model we narrow down to its *services*, *processes* and *technology*. We 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. We also illustrate various parts of the architecture on a running example from the Business-to-Business (B2B) integration domain and finally describe evaluation of the architecture implementation with respect to this example.

2 Abbreviations

This sections provides the list and explanation of abbreviations we use in the article.

- **A2A** - Application to Application Integration.
- **B2B** - Business to Business.
- **EAI** - Enterprise Application Integration.
- **CRM** - Customer Relationship Management.
- **IDE** - Integrated Development Environment.
- **OMS** - Order Management System.
- **OWL** - Web Ontology Language.
- **QoS** - Quality of Service.
- **SAWSDL** - Semantic Annotations for Web Service Description Language.
- **SEE** - Semantic Execution Environment.
- **SESA** - Semantically-enabled Service Oriented Architecture .
- **SOA** - Service Oriented Architecture.
- **SWS** - Semantic Web Services.
- **PIP** - Partner Interface Process.
- **PO** - Purchase Order.
- **RMI** - Remote Method Invocation.
- **WG** - Working Group.
- **WSDL** - Web Service Description Language.
- **WSML** - Web Service Modeling Language.

- **WSDM** - Web Service Distributed Management.
- **WSMO** - Web Service Modeling Ontology.
- **WSMT** - Web Service Modeling Toolkit.
- **WSMX** - Web Service Execution Environment.

3 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.

4 Global SESA Architecture

In this section we define the overall SESA architecture, depicted in Figure 1, building on governing principles introduced in section 3. 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.

4.1 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

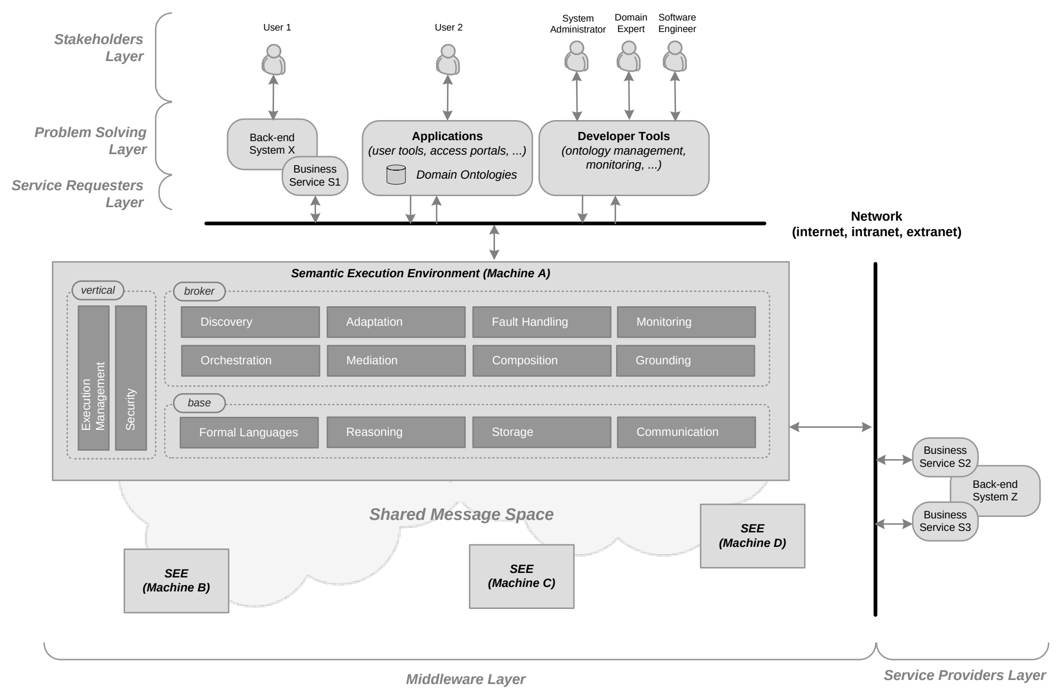

Figure 1 Global View

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.

4.2 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 functionality 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.

4.3 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.

4.4 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 1). The SEE defines the necessary conceptual functionality that is imposed on the architecture through the underlying principles defined in section 3. Each such functionality could be realized (totally or partially) by a number of so called *middleware services* (in section 7.1 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.

4.5 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.

5 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 4. 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)[1], OWL-S[2] and WSDL-S[3].

With respect to requirements imposed on the architecture through the governing principles, we choose the WSMO model for our work[5]. 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 3). 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)[1] – 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)[6] – 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.

[5] A comparison of WSMO with its main competitor OWL-S and other approaches is given in section 11.

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

6 Running Example

In this section we introduce a running example which we will use throughout the paper to demonstrate various aspects of the Semantically-enabled Service Oriented Architecture . This scenario and its implementation is based on scenarios from the SWS Challenge[7], an initiative which provides a standard set of increasingly difficult problems, based on industrial specifications and requirements.

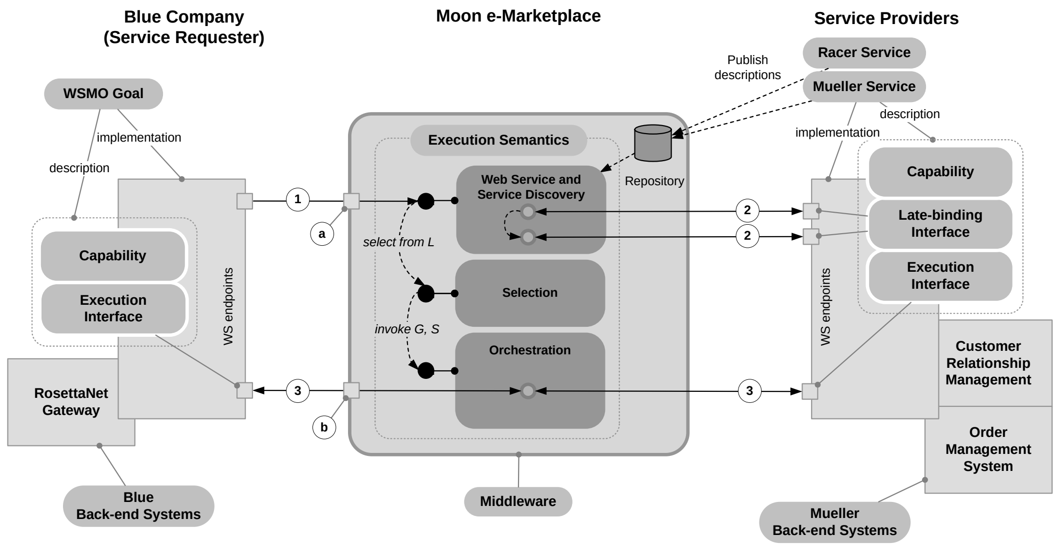

Figure 2 Running Example

As depicted in figure 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 Semantically-enabled Service Oriented Architecture . 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[8] 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

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

[8] 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.

more interactions with its back-end systems such as identify a customer in the CRM, open the 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 7.3). 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 (see section 8.5). We will refer to this scenario and the figure 2 throughout the paper to illustrate various aspects of the Semantically-enabled Service Oriented Architecture .

7 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 5). 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).

7.1 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 4.4. These services also reflect the current WSMX implementation – the reference implementation of the SEE middleware. In figure 3, 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 8.2.

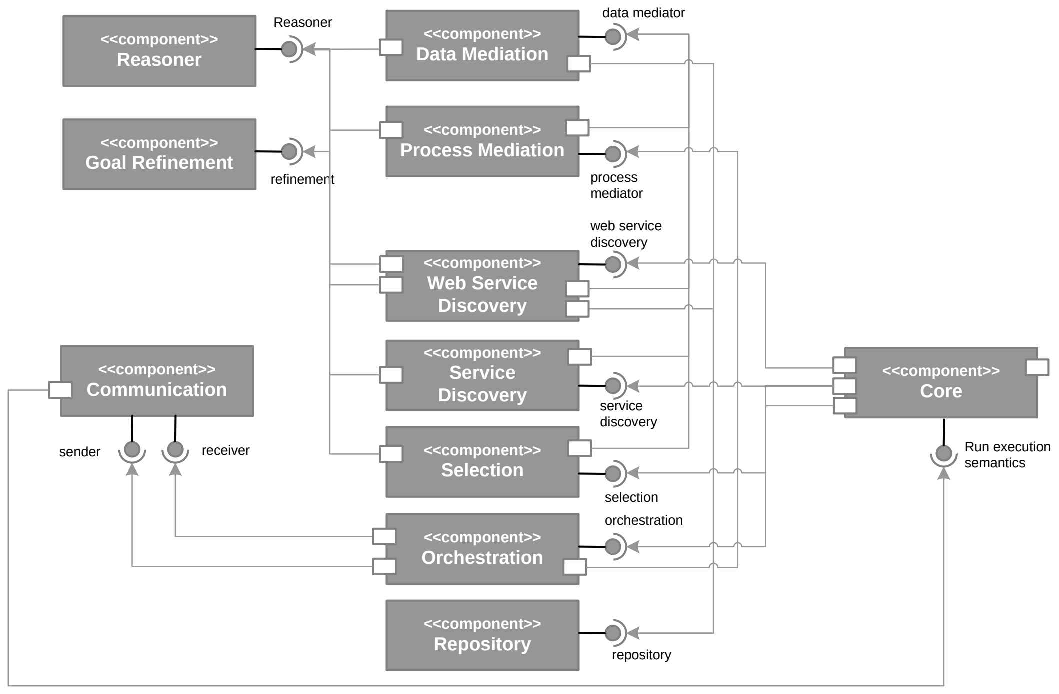

Figure 3 Service View – Middleware

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 8). 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[9]. 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 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[10]) 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[11].

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.

[9] wsmo4j.sourceforge.net
[10] http://tools.deri.org/wsml2reasoner
[11] http://www.wsmo.org/wsml

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 [4].

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 [5].

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 $?x[name\ hasValue\ \text{“}HarryPotter''] \ memberOf\ book$ whereas abstract goal contains axioms in a form $?x\ memberOf\ book$ where instance of the $book$ concept is provided separately from the goal definition.

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 [6].

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 8) within the execution process and service providers might take place in order to retrieve this information through specialized service interfaces. More detailed information about service discovery in WSMO can be found in [7].

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 [8].

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.

7.2 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 4).

- **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[12] (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 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

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

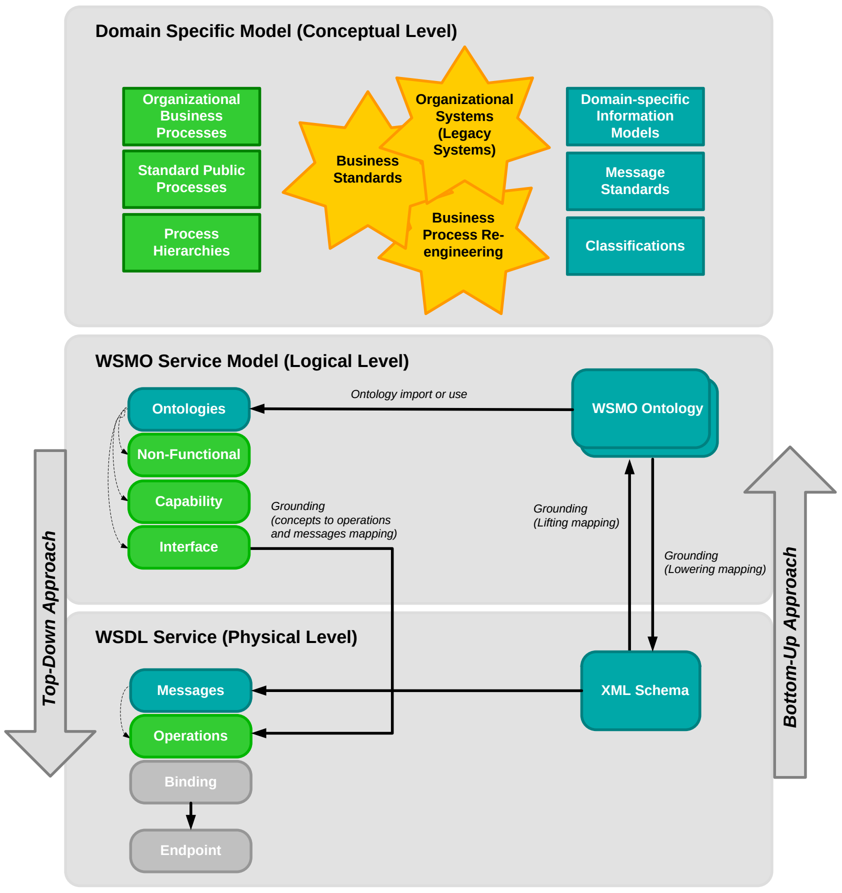
Domain Specific Model (Conceptual Level)
Organizational Business Processes
Standard Public Processes
Process Hierarchies
Business Standards
Organizational Systems (Legacy Systems)
Business Process Re-engineering
Domain-specific Information Models
Message Standards
Classifications
WSMO Service Model (Logical Level)
Ontologies
Non-Functional
Capability
Interface
Ontology import or use
WSMO Ontology
Grounding (concepts to operations and messages mapping)
Grounding (Lifting mapping)
Grounding (Lowering mapping)
WSDL Service (Physical Level)
Messages
Operations
Binding
Endpoint
XML Schema
Top-Down Approach
Bottom-Up Approach

Figure 4 Semantic Business Service Modeling Levels

WSDL descriptions using the recent Semantic Annotations for WSDL (SAWSDL) approach[13]. 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, 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.

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

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[14]. 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 [9]. In section 7.3 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. 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[15].

7.3 Example of Business Services Modeling

In this section we show how the Blue and Mueller systems from the example in section 6 can be modeled using WSMO and WSML formalisms. In this example, we use the top-down approach to modeling of services (described in section 7.2), 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

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

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

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.

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 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 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 8.2), we distinguish two types of choreography definitions, namely *late-binding choreography* and *execution choreography*. Listing 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.

```
1   /* late−binding choreography for service discovery stage */
2   choreography dlb#DiscoveryLateBindingChoreography
3       stateSignature
4         in mu#purchaseQuoteReq withGrounding { ... }
5         out mu#PurchaseQuoteResp withGrounding { ... }
6
7     forall {?purchaseQuoteReq} with (
8       ?purchaseRequest memberOf mu#PurchaseQuoteReq
9       ) do
10        add( # memberOf mu#PurchaseQuoteResp)
11      endForall
12  ...
13
14  /* execution choeography for service execution stage */
15  choreography exc#ExecutionChoreography
16      stateSignature
17        in mu#SearchCustomerRequest withGrounding { ... }
18        out mu#SearchCustomerResponse withGrounding { ... }
19
20  transitionRules MoonChoreographyRules
21    forall {?request} with (
22        ?request memberOf mu#SearchCustomerRequest
23      ) do
24        add(_# memberOf mu#SearchCustomerResponse)
25    endForall
26  ...
```

Listing 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, 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 [4]. 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 Mapping Rules in WSML

8 Process View

Processes reflect the behavior of the architecture through which stakeholders interact with the middleware and with business services. Similarly as in section 7, we distinguish two types of processes, namely (1) *middleware processes* and (2) *business processes*.

8.1 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 8.2, the late-binding and execution phases is described in detail.

8.2 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 8.4.

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).

8.3 Example of External Integration

In figure 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 2 for example of late-binding interface in section 7.3). Finally, the execution and conversation between service requester and discovered services is performed (these are execution interactions marked as 3 in the figure).

8.4 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 5 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

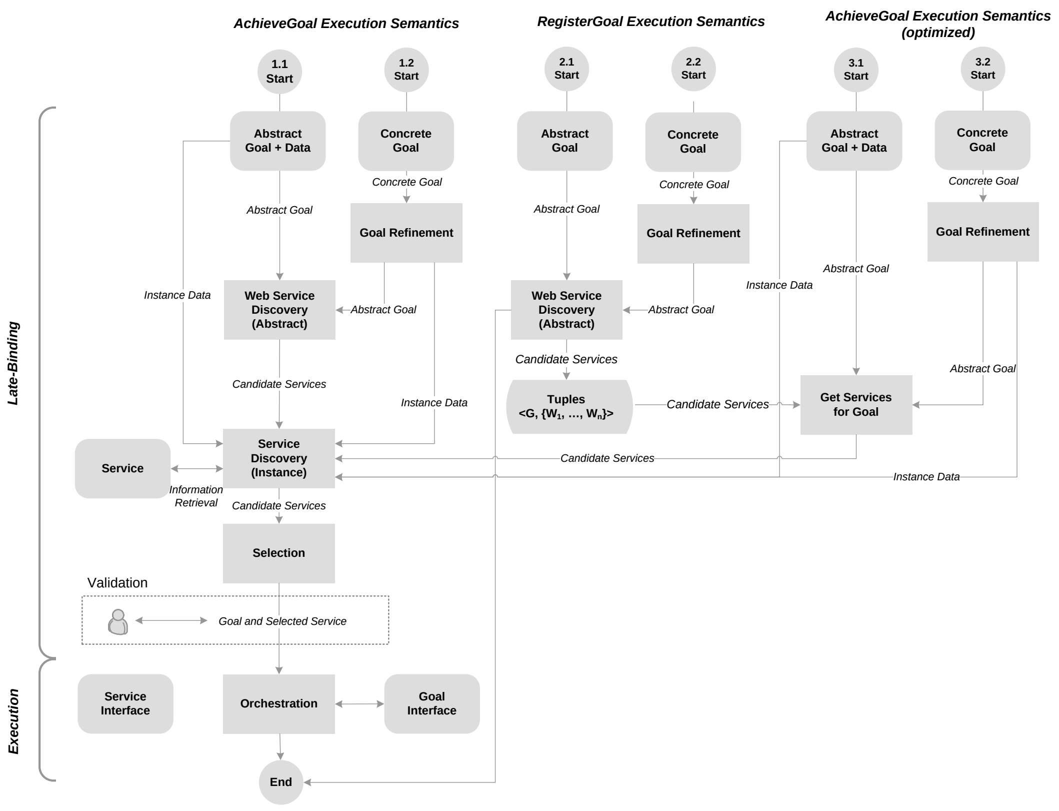

Figure 5 Execution Semantics

3.2 branches in the figure 5). During the refinement, the reasoning about goal definition is performed with result of the new abstract goal and instance data defined separately which both correspond to the original concrete goal definition.

AchieveGoal Execution Semantics. For the execution semantics *AchieveGoal* (see branches 1.1 and 1.2 in the figure 5), (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 disjnontness [6]. When the match is found[16], 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. 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

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

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 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. Orchestration enables interchange of messages between requester and provider by processing of their choreography interfaces. In addition, those interfaces might follow slightly different protocols as well as can use different ontologies, thus data and process mediation is applied during this processing. In figure 6 a control state diagram for the execution between service requester and service provider is shown. The core functionality for the execution is provided by the Orchestration component which processes the requester's and provider's choreographies according to the control mechanisms implemented in the Process Mediation. The choreographies in WSMO are modeled as Abstract State Machines thus its processing is based on evaluation of rules' heads in the processing memory of the Orchestration component. When this evaluation holds, the rule body is executed resulting in *adding*, *updating* or *removing* data in the processing memory. Adding/updating data in the memory means that the actual data needs to be obtained from the service by invoking underling operation of the service from the WSDL description (information which WSDL operation to invoke is part of the grounding definition of the choreography description in WSMO service).

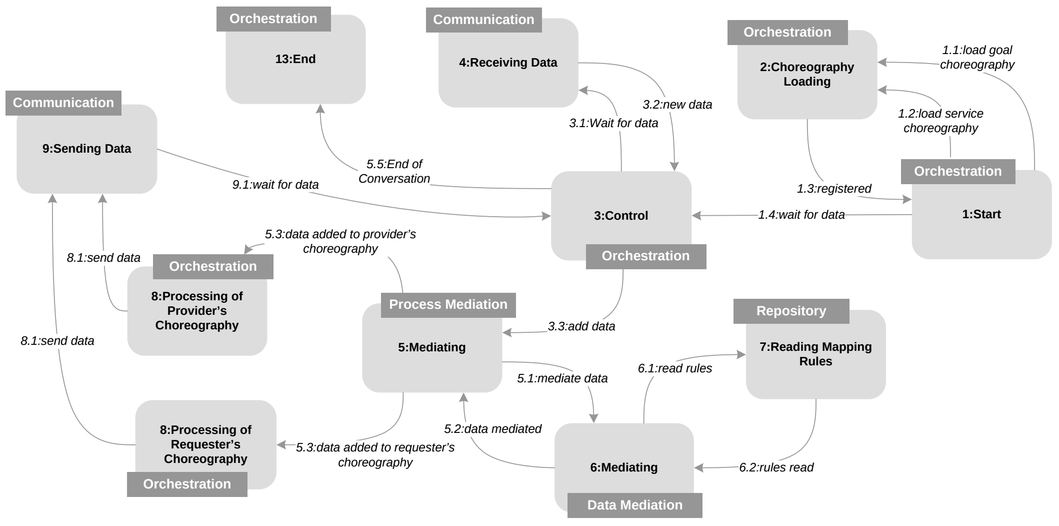

Figure 6 Control State Diagram of the Execution Phase

The execution is initiated by loading requester's and provider's choreography interfaces (these interfaces are taken from goal and service descriptions respectively) (transitions 1.1, 1.2 and state 2). When the choreographies are loaded, the Orchestration gets to the control state (state 3) managing the whole execution process. At this stage, it can wait for the new data, end the execution or request process mediator to add the new data to a processing memory of a loaded choreography. Requester or provider, by following a protocol described by their choreography interfaces, can either send or expect to receive messages from/to the middleware. Also, there might be already some data available initially which could have been provided as part of goal definition. When data is available from either side, the control is passed to the process mediator (transition 3.3 and state 5). Next, the data is mediated to the provider's ontology if the data originates from the requester or vice-versa (transition 5.1, state 6). After the data mediation is finished, the process mediator decides where to put the new data, either into processing memory of the requester's choreography, processing memory of the provider's choreography or both. This decision is based on the evaluation of rules' heads of each choreography. In particular it is evaluated if the data could be potentially used in subsequent processing of respective choreography. Based on this evaluation, the data is added to the particular choreography (transitions 5.3). Next, the updated choreographies are processed, meaning that the next rule of each updated choreography is evaluated. In particular, for a rule whose head satisfies the content of the processing memory of the choreography, the body is executed which means that some data should be either added, updated or removed from the memory (state 8). In case of add or update, the actual data needs to be obtained from the service requester or service provider respectively. This is done by processing of the grounding definition of the concept that needs to be added/updated when invocation of underlying operation of the WSDL service is performed (transition 8.1 and state 9). When the new data is added, the execution gets back to the control state and potentially to receiving data state (transitions 9.1, state 3 or transition 3.1, state 4 respectively). The Orchestration might also evaluate that there is no additional rules to be processed and may get to the end of execution state (transition 5.5, state 13). In section 8.5 we further describe this process on the running example.

8.5 Example of Service Execution.

In this section we describe the execution phase run in the middleware for our example described in section 6. This phase is depicted in the figure 7 and implements the *AchieveGoal* execution semantics as described in section 8.4 (branch 1.1 in figure 5. We further divide this phase into following stages to which we refer using numbers before each paragraph: (1) *Sending Request*, (2) *Late-Binding – Discovery, Selection and Conversation Set-up*, (3) *Execution – Conversation with Service Provider*, (4) *Execution – Conversation with Service Requester*. The execution phase described herein follows the business service modeling phase for the same example as described in section 7.3.

1: Sending Request A RosettaNet PIP3A4 PO message is sent from the Blue Company to the entry point of the PIP3A4 adapter. In the PIP3A4 adapter, the PO XML message is lifted to WSML according to the PIP3A4 ontology and rules for lifting using XSLT. Finally, a WSMO Goal is created from the PO message, including the definition of the desired capability and choreography as well as instance data (this goal is created as abstract goal which contains separately defined data (8.4)). The capability of the requester (Blue company) is used during the discovery process whereas the Goal choreography describes how the requester wishes to interact with the environment. After the Goal is created, it is sent together with the data to middleware by invoking the *AchieveGoal* execution entrypoint.

2: Late-Binding – Discovery, Selection, and Conversation Set-up . The *AchieveGoal* execution entrypoint is implemented by the Communication service, which facilitates inbound and outbound communication with the middleware. After receipt of the Goal, the Communication Service initiates the AchieveGoal execution semantics and starts the middleware process by passing the WSMO goal to it. The first step of the middleware process is to parse the message and perform the discovery. Discovery is first run at abstract level (web service discovery) when matching of capability descriptions of the goal and each service from the repository is processed. When match is found at abstract level, the match is further performed at instance level. For this purpose, fetching some additional data from service requester is done through processing of discovery late-binding interface of each candidate service (if interface exists). Such interface for Mueller service is shown in listing 2. Through this interface, some additional data is obtained which is then used for fine grained querying of the knowledge base containing all instance data of the goal and the candidate service. During this process, interactions with data mediation as well as communication services are also performed (for brevity these are not shown in the figure 7). On result, the list of services all matching at abstract

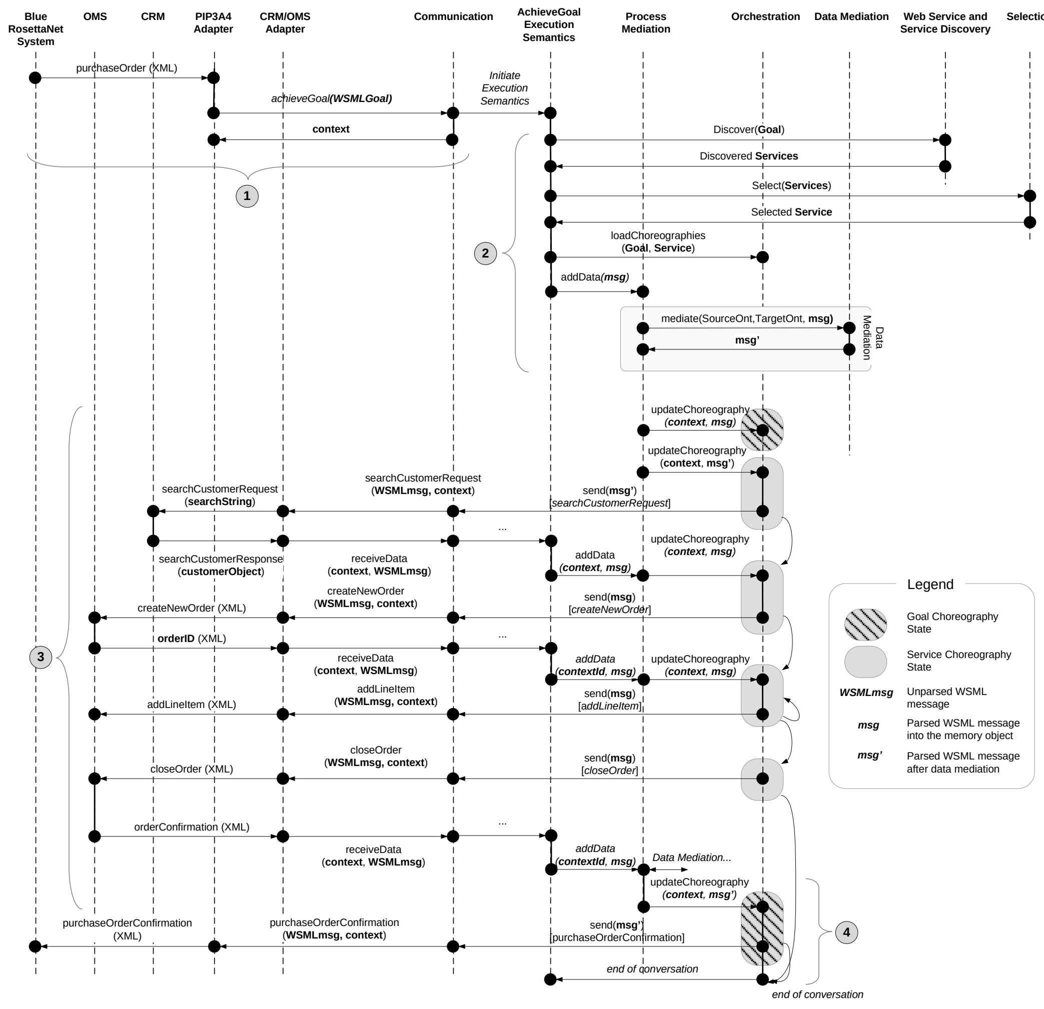
Blue RosettaNet System
OMS
CRM
PIP3A4 Adapter
CRM/OMS Adapter
Communication
AchieveGoal Execution Semantics
Process Mediation
Orchestration
Data Mediation
Web Service and Service Discovery
Selection
purchaseOrder (XML)
achieveGoal(WSMLGoal)
Initiate Execution Semantics
context
Discover(Goal)
Discovered Services
Select(Services)
Selected Service
loadChoreographies (Goal, Service)
addData(msg)
mediate(SourceOnt,TargetOnt, msg)
msg'
Data Mediation
updateChoreography (context, msg)
updateChoreography (context, msg')
searchCustomerRequest (searchString)
searchCustomerRequest (WSMLmsg, context)
send(msg') [searchCustomerRequest]
searchCustomerResponse (customerObject)
receiveData (context, WSMLmsg)
addData (context, msg)
createNewOrder (XML)
createNewOrder (WSMLmsg, context)
send(msg) [createNewOrder]
orderID (XML)
addData (contextId, msg)
addLineItem (XML)
addLineItem (WSMLmsg, context)
send(msg) [addLineItem]
closeOrder (XML)
closeOrder (WSMLmsg, context)
send(msg) [closeOrder]
orderConfirmation (XML)
Data Mediation...
purchaseOrderConfirmation (XML)
purchaseOrderConfirmation (WSMLmsg, context)
send(msg') [purchaseOrderConfirmation]
end of conversation
1
2
3
4
Legend
Goal Choreography State
Service Choreography State
WSMLmsg Unparsed WSML message
msg Parsed WSML message into the memory object
msg' Parsed WSML message after data mediation

Figure 7 Sequence Diagram

and instance levels are returned. The next step is to perform the selection of the best service which satisfies service requester's preference. For this purpose, the preference on price is defined as part of the goal non-functional properties definition which is used for ranking of candidate services in the list of discovered services. In other words, the list is sorted by the property defined in the preference and the first service is selected from the list. In our case, the Mueller service is selected. The final step of this stage is to set-up the conversation between the service requester and the selected service. For this purpose, loading of both the requester's (Goal) and the provider's (selected service) choreography in the Orchestration service is performed (these choreographies are part of the goal and service descriptions). Both choreographies are then set to a state where they wait for incoming messages that could fire a transition rule. Next, the existing data (available thorough the goal description) is passed to the Process Mediation service to decide which data will be added to requester's or provider's choreography (this decision is based on analysis of both choreographies and concepts used by these choreographies). Process Mediation service first updates the memory of the requester's choreography with the information that the PO request has been sent. The Process Mediation then evaluates how data should be added to the memory of the provider's choreography – data must be first mediated to the ontology used by the provider. For this purpose, the source ontology of the requester, target ontology of the provider and the instance data are passed to the Data Mediation service. Data mediation is performed by execution of mapping rules between both ontologies (these mapping rules are stored within middleware repositories and have been created during the Business Service Modeling phase). The mediated data is added to the provider's choreography which finally completes the conversation setup.

3: Execution – Conversation with Service Provider Once the requester's and provider's choreographies have been updated, the Orchestration service processes each to evaluate if any transition rules could be fired. The requester's choreography remains in the waiting state as no rule can be evaluated at this stage. For the provider's choreography, the Orchestration service finds the rule shown in listing 2 (lines 8-12). Here, the Orchestration service matches the data in the memory with the the antecedent of the rule and performs the action of the rule's consequent. The rule says that the message *SearchCustomerRequest* with data *searchString* should be sent to the service provider (this data has been previously added to the processing memory after the mediation - here, *searchString* corresponds to the *customerId* from the requester's ontology). The Orchestration service then waits for the *SearchCustomerResponse* message to be sent as a response from the provider. Sending the message to the service provider is initiated by Orchestration service passing the message to the Communication service which, according to the grounding defined in the choreography, passes the message to the *searchCustomer* entrypoint of the CRM/OMS Adapter. In the adapter, lowering of the WSML message to XML is performed using the lowering rules for the CRM/OMS ontology and the CRM XML Schema. After that, the actual service of the CRM system behind the adapter is invoked, passing the parameter of the *searchString*. The CRM system returns back to the CRM/OMS Adapter a resulting *customerObject* captured in XML. The XML data is lifted to the CRM/OMS ontology, passed to middleware, evaluated by the Process Mediaton service and added to the provider's choreography memory. Once the memory of the provider's choreography is updated, the next rule is evaluated resulting in sending a *createNewOrder* message to the Mueller service (OMS system). This process is analogous to one described before. As a result, the *orderID* sent out from the OMS system is again added to the memory of the provider's choreography. After the order is created (opened) in the OMS system, the individual items to be ordered need to be added to that order. These items were previously sent in one message as part of order request from Blue's RosettaNet system (i.e. a collection of *ProductLineItem*) which must be now sent to the OMS system individually. As part of the data mediation in step 2, the collection of items from the RosettaNet order request were split into individual items whose format is described by the provider's ontology. At that stage, the Process Mediation service also added these items into the provider's choreography. The next rule to be evaluated now is the rule of sending the *addLineItem* message with data of one *lineItem* from the processing memory. Since there is more then one line item in the memory, this rule will be evaluated several times until all line items from the ontology have been sent to the OMS system. When all line items have been sent, the next rule is to close the order in the OMS system. The *closeOrder* message is sent out from the middleware to the OMS system and since no additional rules from the provider's choreography can be evaluated, the choreography comes to the end of conversation state.

4: Execution – Conversation with Service Requester When the order in the OMS system is closed, it replies with *orderConfirmation*. After lifting and parsing of the message, the Process Mediation service first calls for the mediation of the data to the requester's ontology and then adds the data to the memory of the requester's choreography. The next rule of the requester's choreography can be then evaluated which says that the *purchaseOrderConfirmation* message needs to be sent to the RosettaNet system. After the message is sent, no additional rules can be evaluated from the requester's choreography, thus the choreography comes to the end of conversation state. Since both requester's and provider's choreography are in the end-of-conversation state, the Orchestration service closes the conversation.

9 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 [10]. It is responsible for handling following three main functional requirements: (1) *Management*, (2) *Communication and Coordination*, and (3) *Execution Semantics*.

9.1 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 8, 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.

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

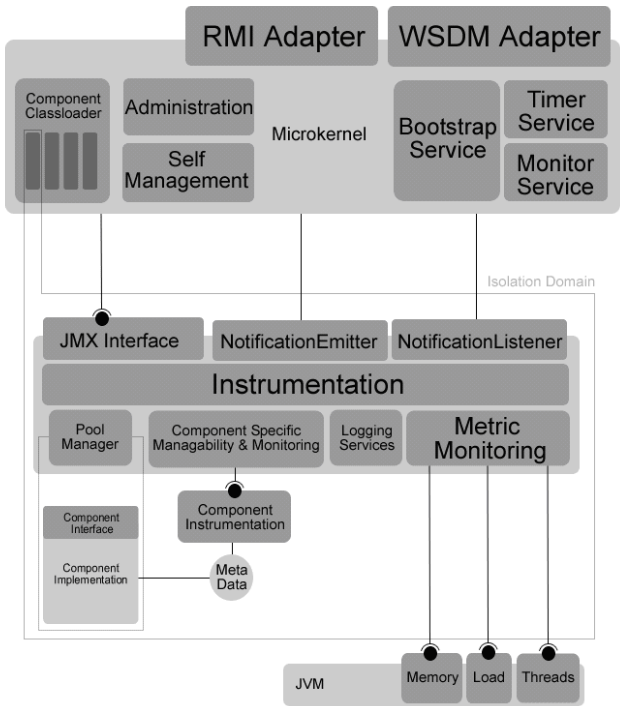

Figure 8 Component Management in the Middleware Execution Management Service

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.

9.2 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 9, 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.

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.

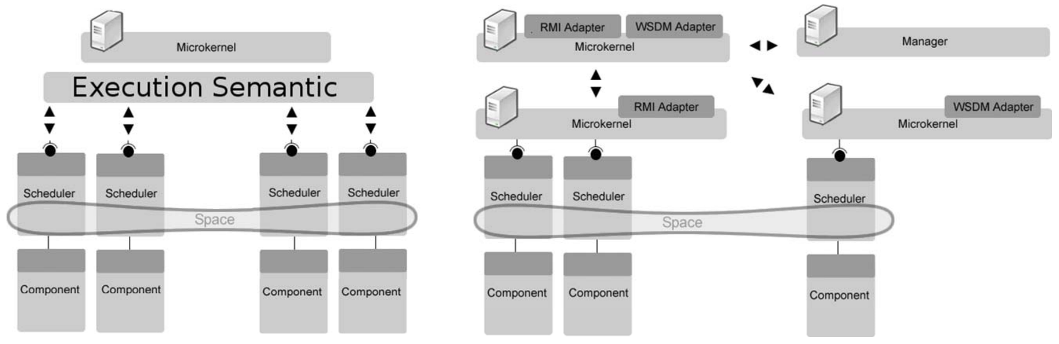

Figure 9 Execution Semantics, Communication and Coordination in the Middleware

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[11] 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.

9.3 Execution Semantics

Execution Semantics enable a combined execution of functional components as illustrated in figure 9. 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 10 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.

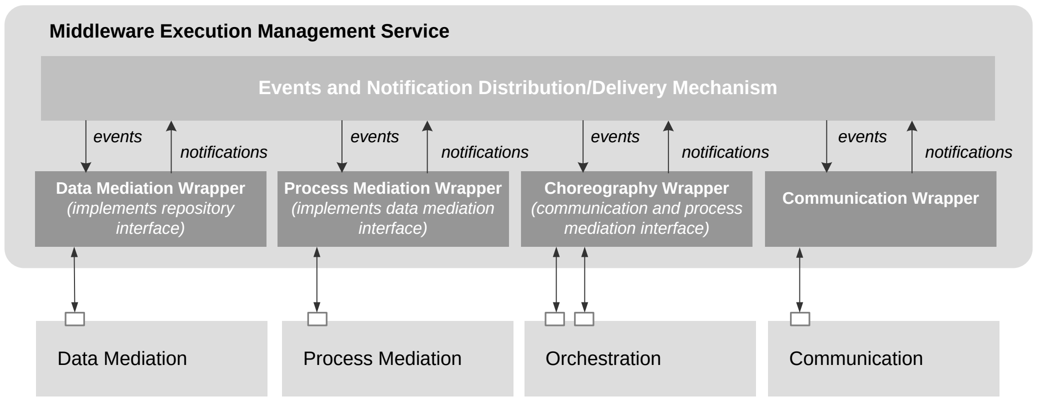
Middleware Execution Management Service
Events and Notification Distribution/Delivery Mechanism
events
notifications
events
notifications
events
notifications
events
notifications
Data Mediation Wrapper
(implements repository interface)
Process Mediation Wrapper
(implements data mediation interface)
Choreography Wrapper
(communication and process mediation interface)
Communication Wrapper
Data Mediation
Process Mediation
Orchestration
Communication

Figure 10 Component's Wrappers and Event Messaging

10 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 Semantically-enabled Service Oriented Architecture presented in this article and its implementation together with implementation of the example from section 6 has been evaluated in the SWS Challenge[17] 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*.

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

- *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.

Our implementation of the scenario from section 6 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 6. The implementation of the mediation problem has been evaluated in Budva, Montenegro[18] 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[19] 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.

11 Related Work

In this section we describe a related work of *Semantic Service Models* which provides background on state-of-the art for Semantic Web Services, and relevant *Service Architectures and Technologies* which describes existing architectures based on the use of the Semantic Service Models.

11.1 Semantic Service Models

OWL-S OWL-S [2] is divided into three sub-ontologies – ServiceProfile, ServiceModel and Grounding. The ServiceProfile describes what a Web Service does and provides the means by which the service can be advertised. In contrast to WSMO Goal and Web service concepts, there is no distinction in the conceptual model of OWL-S between the viewpoints of service requesters and providers. The ServiceProfile is aimed equally at advertising services offered by providers as well as those sought by requesters. Similarly to WSMO, OWL-S defines the capability a service offers as a state transition in terms of pre- and postconditions. The WSMO model of SESA explictly considers the data and process heterogeneity problems in an open Web environment. WSMO defines the concept of mediator to tackle this. OWL-S does not model this problem explicitly, tending to treat it as more of an architectural issue.

The ServiceModel is used to define the behavioural aspect of the Web Service. The ServiceModel allows for the description of different types of services: Atomic, Abstract and Composite. Atomic processes correspond to a single interaction with the service e.g. a single operation in a WSDL document. Composite processes have multiple steps, each of which is an atomic process, connected

[18] http://sws-challenge.org/wiki/index.php/Workshop_Budva
[19] http://sws-challenge.org/wiki/index.php/Workshop_Athens

by control and data flow. Simple processes are abstractions to allow multiple views on the same process. These can be used for the purposes of planning or reasoning. Simple processes are not invocable but are described as being conceived as representing single step interactions. A Simple process can be realized by an Atomic process or expanded to a Composite process. WSMO goes further than OWL-S by modelling orchestrations describing what other Web services have to be used, or Goals to be fulfilled, to perform a specific task. Additionally, WSMO allows the definition of multiple interfaces, and corresponding choreographies, for a Web service while OWL-S allows only a single service model.

The final part of the conceptual model is the ServiceGrounding, providing a link between the ServiceModel and the description of the concrete realization for a Web Service provided by WSDL. Atomic processes are mapped to WSDL operations, where the process inputs and outputs, described using OWL, are mapped to the operations inputs and outputs, described using XML-Schema.

The use of OWL-DL as the ontology language for OWL-S had some unwanted side-effects noted in detail in [12]. Included amongst these was that OWL-S does not comply with the OWL-DL specification, which places constraints on how OWL-S ontologies can be reasoned over. A second problem is that variables are not supported within OWL but are necessary when combining data from multiple co-operating processes. Additionally, a significant problem, is that OWL-DL is not well suited to describing processes - an important aspect for Semantic Web Service descriptions. The lack of clear separation between language layers in OWL-S and the open interpretation of OWL-S semantics if expressions in SWRL[13] or KIF[14] are combined, was a strong motivation for the MOF-style consistent layering of the WSML family of languages.

SWSF and SWSO The establishment of the Semantic Web Services Framework (SWSF) [15] was devised to provide a full conceptual model and language expressive enough to describe the process model of Web Services, and to address the shortcomings of OWL-S in this regard. The first order logic axiomatisation of SWSO is called FLOWS (First Order Logic Ontology for Web Services) and is based on the Process Specification Language (PSL) [16], an ISO international standard process ontology.

FLOWS is expressed in a language called SWSL-FOL (Semantic Web Services Language for First Order Logic). To enable logic-programming based implementations and reasoning for SWSO, a second ontology available called ROWS (Rules Ontology for Web Services), expressed in SWSL-Rules. ROWS is derived from FLOWS by a partial translation. The intent of the axiomatisation of ROWS is the same as that of FLOWS but in some cases is weakened because of the lover expressivity of the SWSL-Rules language.

Service is the primary concept in SWSO with three top level elements, derived from the three parts of the OWL-S ontology. These are Service Descriptors, Process Model and Grounding. *Service Descriptors* provide a set of non-functional properties that a service may have. The FLOWS specification includes examples of simple properties such as the name, author, textual description. The set is freely extensible. Metadata specifications for online documents including Dublin Core are also easily incorporated. Each property is modelled as a relation linking the property to the service. The *Process Model* extends the PSL generic ontology for processes with two fundamental elements, especially to cater for Web Services (i) the structured notion of atomic processes as found in OWL-S and (ii) infrastructure for allowing various forms of data flow. The Process Model of FLOWS is organised as layered extension of the PSL-OuterCore ontology. The SWSO approach to *grounding* follows very closely that of OWL-S v1.1 to WSDL.

Like SWSL Rules, WSMO's rule language WSML-Rule is largely based on F-Logic. The major difference between the SWSO and WSMO efforts is the focus of the former on providing a higly expressive first-order logic ontology for describing process models. WSMO uses guarded transition rules, considered as abstract state machines, to define its process model but does not have the detailed semantics of SWSO. On the other hand SWSO adopts the OWL-S model for describing other aspects of Web services and so does not explicitly model the notion of goals or of mediators.

WSDL-S WSDL-S [17] is a lightweight approach for augmenting WSDL descriptions of Web Services with semantic annotations. It is a refinement of the work carried on by the Meteor-S group at the LSDIS Lab, Athens, Georgia to enable semantic descriptions of inputs, outputs, preconditions and effects of Web Service operations, by taking advantage of the extension mechanism of WSDL. WSDL-S is agnostic to the ontology language and model used for the annotations of WSDL and does not introduce a detailed conceptual model for Web service. Recently, the WSDL-S specification has been migrated to SAWSDL as part of a W3C standards activity.

SAWSDL The Semantic Annotations for WSDL (SAWSDL) W3C Candidate Recommendation, is an incremental bottom-up approach to Semantic Web Services where elements in WSDL documents are provided with semantic annotations through attributes provided using standard valid extensions to WSDL 2.0. The approach is agnostic to the ontological model used to define the semantics of annotated WSDL elements. From SAWSDLs perspective, the annotations are valued by URIs. SAWSDL defines its conceptual model using XML Schema and confines itself to attributes of modelReference, and two specialisations of schemaMapping, namely, liftingSchemaMapping and loweringSchemaMapping. The modelReference attribute can be used to annotate XSD complex type definitions, simple type definitions, element declarations, and attribute declarations as well as WSDL interfaces, operations, and faults. The liftingSchemaMapping can be applied to XML Schema element declaration, complexType definitions and simpleType definitions.

11.2 Service Architectures and Technologies

W3C Web Service Architecture W3C Web Service Architecture[18] describes the main concepts, relationships and models behind a Web service-based architecture. It identifies four architectural models: the message oriented model (focuses on messages, message structure, message transport, etc.), the Service Oriented Model (focuses on aspects of service, action, etc.), the Resource Oriented Model (focuses on resources that exist and have owners) and the Policy Model focuses on constraints on the behavior of agents and services). Based on this conceptual model the stakeholder's perspective is described showing how the architecture meets the goals and the requirements. The stakeholder's perspective description includes the main architectural properties and design principles of Service Oriented Architectures, an overview of the Web Services Technologies stack and the role of the Web Service Semantics. Our work can be seen as complementary, since it respects the same design principles, uses the same underlying technologies and above all recognizes the outmost importance of the explicitly describing the Web Service Semantics. That is, SESA adds semantics on top of existing technologies stack (e.g. SOAP, WSDL, etc.) and show how process such as Discovery, Composition, Choreography, etc. can be enhanced by semantics. While [18] defines the generic principles behind Web Services Architecture, our approach realizes such an architecture (in the lines of the principles) and furthermore adds a semantic layer on top of existing standards and technologies in order to enable dynamic and (semi-) automated discovery, composition, mediation and invocation of semantically described Web services.

OWL-S Virtual Machine The OWL-S Virtual Machine (OWL-S VM) [19] provides a general purpose Web service client for the invocation of a Web service based on the process model of its OWL-S description. The architecture consists of components for executing the OWL-S process model, the grounding, and for making the Web service invocation. The OWL-S VM provides a first implementation, and therefore proof-of-value, of the OWL-S process model but only addresses a subset of the functionality offered by the SESA described in this article. SESA focuses on providing an environment where multiple semantically described services can interact. Data and process mediators are first class citizens, based on the assumption that independent services will exhibit, not only data, but also behavioural independence. Both types of mediation in SESA are based at the conceptual level on mappings between ontologies. In the OWL-S VM, data heterogeneity is handled syntactically, based on the use of XSLT. The SESA prototype, WSMX, is an open-source project where the formal execution semantics and component-based architecture descriptions are public. A detailed architecture and runtime execution semantics for OWL-S VM are not publicly available.

IRS-III IRS-III [20] is an execution environment for Semantic Web services that also uses WSMO as the underlying conceptual model. To facilitate its implementation of capability-based service invocation, IRS-III extends the definition of the WSMO Goal slightly to introduce input and output roles as well as soap bindings for these roles. Conceptually, both SESA and IRS-III have common roots in the UPML framework of Fensel et al. [21] Although there are implementation differences, both the SESA prototype, WSMX, and IRS-III implement a common system-level API to facilitate interoperability.

METEOR-S The METEOR-S [3, 22] project of the LSDIS Lab proposes the application of semantics to existing Web service technologies. In particular the project endeavours to define and support the complete lifecycle of Semantic Web service processes. Their work includes extending WSDL to support the development of Semantic Web services using semantic annotation from additional type systems such as WSMO and OWL ontologies. A similar approach to data mediation is taken by both WSMX and METEOR-S, based on using a common data representation format. METEOR-S proposes an enhancement of UDDI to facilitate semantic discovery as well as a framework for SWS composition. METEOR-S allows for (i) the creation of WSDL-S descriptions from annotated source code, (ii) the automatic publishing of WSDL-S descriptions in enhanced UDDI registries and (iii) the generation of OWL-S descriptions, from WSDL-S, for grounding.

The publication and discovery (MWSDI) module provides support for semantic publication and discovery of Web services across a federation of registries as well as a semantic publication and discovery layer over UDDI. The composition module consists of two main sub-modules - the constraint analysis and optimization sub-module and the execution environment. The constraint analysis and optimization sub-modules deal with correctness and optimization of the process on the basis of quality of service constraints. The execution environment provides proxy based dynamic binding support to an execution engine for BPEL4WS.

SWS Challenge The implementation of the SESA described in this article is one of the contributions to the SWS Challenge series of workshops where several others have been presented. In particular, the entry from DEI and CEFRIEL, Milano [23], was evaluated to be the most complete at the workshop in Budva, June 2006. The DEI/CEFRIEL approach was to use the WebML language to specify their solution to the mediation problem presented by the Challenge. WebML uses entity-relations diagrams extended with Object Query Language constraints to create the data model. The authors then use an extension of WebML that permits interactions with Web services and a further extension that allows process models, specified using the Business Process Modelling Notation, to be translated into executable WebML. Process mediation is catered for at design-time through the use of GUI-based BPMN modelling tools. Data mediation is carried out by XSLT transformation between SOAP message and the internal WebML data model. The system was shown to solve the problems presented by the challenge and changes to the problem specification were able to be addressed relatively easily. However, this was only possible by changing and redeploying the BPMN process model or recoding the XSLT transformations by hand. This is in contrast to the approach of SESA where changing requirements are catered for by the specification of the relevant declarative elements (ontologies, service or goal descriptions, inter-ontology mappings).

12 Conclusion

The Semantically-enabled Service Oriented Architecture presented in this article follows a new approach to integration and interoperation of services by means of semantic languages and semantic service models. Building on the Web Service Modeling Ontology and taking into account governing principles of service orientation, semantic modeling and problem solving methods, the architecture provides a means to total or partial automation of tasks such as discovery, mediation, selection and execution of semantic web services. An important aspect which underpins this integration lies in so called goal-based discovery and invocation of semantic services when users describe requests as goals semantically and independently from services while architecture solves those goals by means of logical reasoning over semantic descriptions. Ultimately, users do not need to be aware of processing logic but only care about the result and its desired quality. With respect to underlying principles, the architecture is defined from several perspectives, presenting global view, service view, process view and technology view. In the global view, several architecture layers are identified including stakeholders, problem solving layer, service requesters, middleware and service providers. The core of the architecture lies in the middleware layer for which several conceptual functionalities are defined. From the service perspective, the architecture defines in detail its service types as middleware and business services and further for each type provides detail specification of how middleware services implement the desired middleware functionality and how business services can be modeled based on WSMO semantic model. The process view than presents how combinations of middleware services into middleware processes can provide certain functionality to the stakeholders and facilitate certain execution scenarios implemented using the architecture while at the same time promoting the seamless provisioning of business services. Within those processes, independently modeled requests and services are bound and executed in the architecture with additional functionality applied where necessary during this processing such as data and process mediation to resolve heterogeneity issues, etc. The technology view finally presents the technology for the middleware system implementation which follows the distributed principle of the architecture and provides the support for distributed management, communication and coordination of middleware processes.

One of the major aspects of the architecture is to facilitate the flexible integration of services which is more adaptive to changes in business requirements. With this respect, the architecture and its implementation has been evaluated according to semantic web services community-agreed evaluation methodology and case scenarios for semantic-based systems defined as part of the SWS Challenge initiative. It has been evaluated how the implementation is capable of adapting to changes in the back-end systems by only modifying the semantic descriptions of services. The ultimate goal for the architecture for our future work is to get to the level where the architecture can adapt without changes in code and configuration – the adaptation will be purely based on reasoning over semantic descriptions of services, their information models and interfaces. We also plan to expand our solution to cover more B2B standards and to integrate key enterprise infrastructure systems such as policy management, service quality assurance, etc. We also plan to target additional scenarios from B2B integration focusing on dynamic composition of services as well as extend the functionality of the middleware services along the lines of fault handling, security, orchestration, etc.

References

1. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontologies **1**(1) (2005) 77 – 106
2. Martin, D., et al.: Owl-s: Semantic markup for web services, version 1.1 available at http://www.daml.org/services/owl-s/1.1/. Member submission, W3C (2004) Available from: `http://www.w3.org/Submission/OWL-S/`.
3. Patil, A., Oundhakar, S., Sheth, A., Verma, K.: Semantic Web Services: Meteor-S Web Service Annotation Framework. In: 13th International Conference on World Wide Web. (2004) 553–562
4. Mocan, A., Cimpian, E., Kerrigan, M.: Formal model for ontology mapping creation. In: International Semantic Web Conference. (2006) 459–472
5. Cimpian, E., Mocan, A.: Wsmx process mediation based on choreographies. In: Business Process Management Workshops. (2005) 130–143
6. Keller, U., Lara, R., Lausen, H., Polleres, A., Fensel, D.: Automatic location of services. In: ESWC. (2005) 1–16
7. Vitvar, T., Zaremba, M., Moran, M.: Dynamic discovery through meta-interactions with service providers. In: ESWC. (2007)
8. Wang, X., Vitvar, T., Kerrigan, M., Toma, I.: A qos-aware selection model for semantic web services. In: ICSOC. (2006) 390–401
9. Kopecký, J., Roman, D., Moran, M., Fensel, D.: Semantic web services grounding. In: AICT/ICIW. (2006) 127
10. Haselwanter, Thomas: WSMX Core - A JMX Microkernel. PhD thesis, University of Innsbruck (2005)
11. Gelernter, D., Carriero, N., Chang, S.: Parallel Programming in Linda. In: Proceedings of the International Conference on Parallel Processing. (1985)
12. Balzer, S., Liebig, T., Wagner, M.: Pitfalls of owl-s: A practical semantic web use case. In: 2nd international conference on Service oriented computing, New York, USA, ACM Press (2004) 289–298
13. Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combiningOWL and RuleML.Available fromhttp://www.w3.org/Submission/2004/SUBMSWRL-20040521/. Technical report (2004)
14. Genesereth, M., Filkes, R.: Knowledge Interchange Format (KIF), Stanford University Logic Group, Logic-92-1. Technical report (1992)
15. Battle, S., al., e.: Semantic web services framework (swsf) overview, w3c submission, available at http://www.w3.org/submission/swsf/. Technical report (2005)
16. Michel, J., Cutting-Decelle, A.: The process specification language, international standards organization iso tc184/sc5 meeting. Technical report (2004)
17. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M., A.Sheth, Verma, K.: Web service semantics wsdls. technical note, april 2005. available at http://lsdis.cs.uga.edu/library/download/wsdlsv1.html. Technical report (2005)
18. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture. W3C Working Group Note (2004)
19. Paolucci, M., Ankolekar, A., Srinivasan, N., Sycara, K.: The daml-s virtual machine. In: The Semantic Web - ISWC 2003, LNCS 2870, Springer-Verlag. (2003) 290–305
20. Motta, E., Domingue, J., Cabral, L., Gaspari, M.: Irs-ii a framework and infrastructure for semantic web services. The Semantic Web ISWC 2003. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg **2870** (2003) 306–318
21. Fensel, D., Benjamins, V., Motta, E., Wielinga, B.: Upml: A framework for knowledge system reuse. In: Proceedings of the International Joint Conference on AI (IJCAI-99), Stockholm, Sweden (1999)
22. Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The meteor-s approach for configuring and executing dynamic web processes, technical report, available at http://lsdis.cs.uga.edu/projects/meteor-s/techrep6-24-05.pdf. Technical report (2005)
23. Brambilla, M., Celino, I., Ceri, S., Cerizza, D., Valle, E.D., Facca, F.M.: A software engineering approach to design and development of semantic web service applications. In: International Semantic Web Conference. (2006) 172–186