



D13.4v0.1 WSMX Architecture

WSMO Working Draft 22 June 2004

Final version:

<http://www.wsmo.org/2004/d13/d13.4/v0.1/20040622/>

Latest version:

<http://www.wsmo.org/2004/d13/d13.4/v0.1/>

Previous version:

<http://www.wsmo.org/2004/d13/d13.4/v0.1/20040530/>

Editor:

Michal Zaremba

Authors:

Matthew Moran

Michal Zaremba

Reviewer:

Holger Lausen

This document is also available in non-normative [PDF](#) version.

Copyright © 2004 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Table of contents

1. [Introduction](#)
 - 1.2 [Purpose of this Document](#)
 - 1.3 [Scope of WSMX Architecture](#)
 - 1.4 [Document Overview](#)
2. [WSMX Architecture Overview](#)
3. [Black Box Overview](#)
4. [Architecture Components and Interfaces](#)
 - 4.1 [Architecture](#)
 - 4.2 [Components of WSMX Architecture](#)
 - 4.2.1 [User Interface - WSML Editor](#)
 - 4.2.2 [WSMX Manager](#)
 - 4.2.3 [Compiler](#)
 - 4.2.4 [Message Parser](#)
 - 4.2.5 [Resource Manager](#)
 - 4.2.6 [Repository](#)
 - 4.2.7 [Events Manager](#)
 - 4.2.8 [WSMO Registry](#)
 - 4.2.9 [Execution Engine](#)
 - 4.2.10 [Matchmaker](#)
 - 4.2.11 [Selector](#)

[4.2.12 Mediator & XML Converter](#)

[4.2.13 Invoker](#)

[5 . Conclusions and Future Work](#)

[References](#)

[Acknowledgment](#)

1. Introduction

The Web Services Modelling Execution Environment (WSMX) is an execution environment for dynamic discovery, mediation and invocation of web services. WSMX is based on the Web Services Modelling Ontology (WSMO) [Roman et al., 2004], an ontology for describing various aspects related to Semantic Web Services. A web service can be registered with WSMX by describing it in terms of WSMO, using the Web Service Modelling Language (WSML) [Oren et al., 2004], and then by invoking a registration interface provided by WSMX. A service requester with a goal to achieve, submits their goal to WSMX. The WSMX environment takes responsibility for matching the requester goal to capabilities of web services registered to WSMX, selecting the most appropriate web service, mediating between the ontologies of service requester and provider, and finally, invoking the selected web service.

1.1 Purpose of this Document

This document provides the software architecture for WSMX. It starts with a high level picture of the WSMX components and how they relate to each other and goes on to provide detailed descriptions of the components, the interfaces each component provides and the data representation of information passing into and out of these components. One of the key design aims for WSMX is keep individual components decoupled from each other. The intention is for component interfaces to remain stable while component implementations may change provided new implementations support the existing interfaces.

1.2 Scope of WSMX Architecture

There are two operational aspects considered by the WSMX Architecture – compilation and execution. Compilation is the mechanism for making elements relating to Semantic Web Services ready for use by WSMX. WSMO provides an ontology for describing web services, ontologies, mediators and goals. These descriptions are written in WSML and are then compiled to WSMX. Execution, in the simplest case, means discovering and invoking the right web service to carry out a client goal. In more complex cases, execution could mean multiple discovery and invocation operations on different web services carried out in a controlled process. This document describes both compilation and execution in terms of component behaviour. In addition to the other deliverables of the WSMX working group, this document provides the specification for the implementation of the WSMX software.

1.3 Document Overview

Section 2 gives an overview of the architecture. Section 3 provides a black box overview of the WSMX system. Section 4 provides details of the architectural

components and interfaces starting with an overview of the system and then giving detailed information on each component including event and data representation, state management and definition of the execution flow for the architecture. Section 5 concludes the document and looks at future work.

2. WSMX Architecture Overview

In this section we use the term architecture to introduce the abstract software components that make up WSMX and adopt the approach taken in [Bussler, 2003]. We look at the different layers of architecture and describe their responsibilities. We then take a look at the software components that make up each of these layers with respect to WSMX. Section 4 provides a detailed description of how these components interact

Figure 1 shows the layers used in the WSMX architecture.

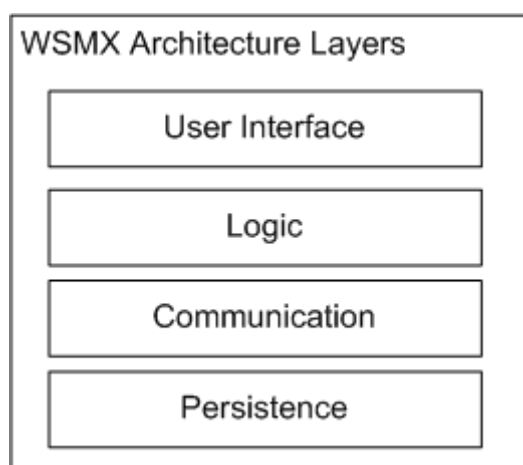


Figure 1: WSMX Architecture Layers

The layers cooperate to provide the overall architecture of the system and follow a top-down invocation pattern. As described in [Bussler, 2003], this means that components in upper layers can invoke components in lower layers but the reverse is not true - components in lower layers may not invoke components in the layers above. This is to prevent circular invocations. Components can also invoke other components in the same layer. The WSMX software components in each layer are shown and described in the following paragraphs.

Figure 2 shows the components in the User Interface layer. At present, the only component in this layer is the **WSMO Editor**. The editor allows a user to create the WSMML descriptions of web services, ontologies, mediators and goals and to send these descriptions to WSMX for compilation.

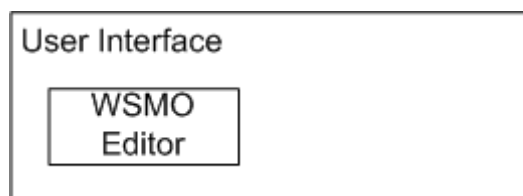


Figure 2: User Interface Layer

Figure 3 shows the software components required by WSMX in the Logic layer.

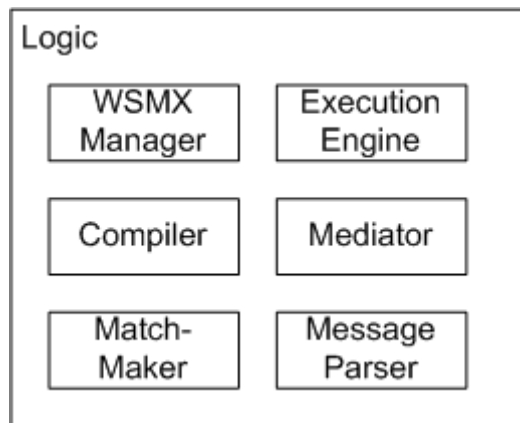


Figure 3: Logic Layer

The **WSMX Manager** and the **Execution Engine** co-ordinate the activities of WSMX following the execution semantics defined in [Oren, 2004]. All data handled inside WSMX is represented internally as an event with a type and state. The WSMX manager manages the processing of all events passing them to other components in the logic layer as appropriate.

The **MatchMaker** is responsible for matching goals to web service capabilities.

In the event that multiple web services are found matching a specific goal, the **Selector** is invoked to select the web service that best fits the requirements of the goal's owner.

The **Mediator** component provide a means of transforming data based on concepts in one ontology to data based on concepts in another ontology. The mapping is based on rules defined between concepts in the source and target ontologies.

In the event that data mediation is required and the mediated data is in a non-XML format, the **XML Converter** can be invoked to translate the results of the Mediator into XML. This is necessary as the web service invocations are via SOAP and the message format for SOAP messages is XML.

The **Compiler** component parses WSML messages received from the WSMO Editor in the User Interface layer, validates the messages against WSMO and then stores the message elements in WSMX repository. The elements compiled to WSMX are the metadata for web services, ontologies, mediators. Once any of these elements have been compiled to WSMX, they are available for use during execution of goals sent to WSMX.

The **Message Parser** parses the WSML Messages containing goals sent to WSMX. The goal is parsed and stored persistently. The functionality of the Message Parser is similar to that of the compiler but there is a conceptual difference. The Message Parser operates on instances of goals while the Compiler operates on the metadata for web services, ontologies, and mediators.

Figure 4 shows the components of the Communication layer.

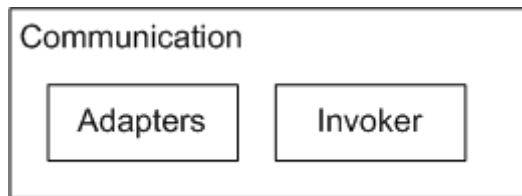


Figure 4: Communication Layer

Adapters allow applications which can not directly communicate with the interfaces provided by WSMX to communicate with WSMX.

The **Invoker** is responsible for making invocations to web services as part of the execution of a goal. Invocations are based on the WSML description of the web service.

Finally figure 5 shows the components in the Persistence layer.

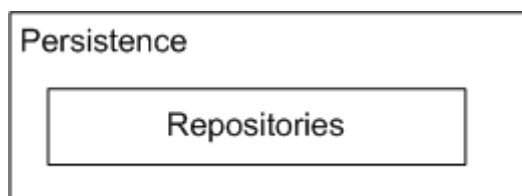


Figure 5: Persistence Layer

The **Repositories** are used to store the definitions of goals, web services, ontologies and mediators within WSMX. The repositories can be either internal to WSMX or external as, for example, the API to the UDDI described in the WSMO Registry [Herzog et al., 2004].

3. Black Box Overview

The purpose of this section is to present the WSMX architecture from a simplified high level perspective to give an overview of what WSMX can do. This section provides an example of how external entities such as intra-enterprise back-end applications or inter-enterprise information systems can interact with the WSMX platform.

In the example below a back end application within an enterprise uses the RosettaNet [RosettaNet] B2B protocol to create messages required to purchase items of computer equipment. The application has the goal to buy computer equipment and has the condition that the messages it expects to send and receive with respect to this purchase will follow the RosettaNet protocol.

Figure 6 illustrates the simple example of a back end application sending a PO to trading partner. The application creates a RosettaNet purchase order (PO) for buying a laptop and sends this to the WSMX adapter. The adapter takes the RosettaNet PO and translates it into a WSML message consisting of a goal that describes what WSMX should execute. Translation means that the PO format has

changed but the data itself has not been changed. Both the RosettaNet and the WSML format for the data obey the same RosettaNet ontology. The goal is then sent to WSMX for execution.

Before WSMX can execute the goal, WSML descriptions of the web services offering the capability to sell laptops, along with the WSML description of the ontologies these web services use, and the RosettaNet ontology, must have been created using the User Interface and compiled to WSMX. When WSMX receives the WSML message specifying the goal, it discovers the web service that best matches that goal, mediates the purchase order data following mapping rules between the RosettaNet ontology and the ontology of the discovered web service and then invokes the web service providing the data to it in the concepts and formats it expects. The main point is that once the web services and ontologies are defined and compiled to WSMX, all the back end application has to do is send the PO to the adaptor and WSMX takes care of the execution.

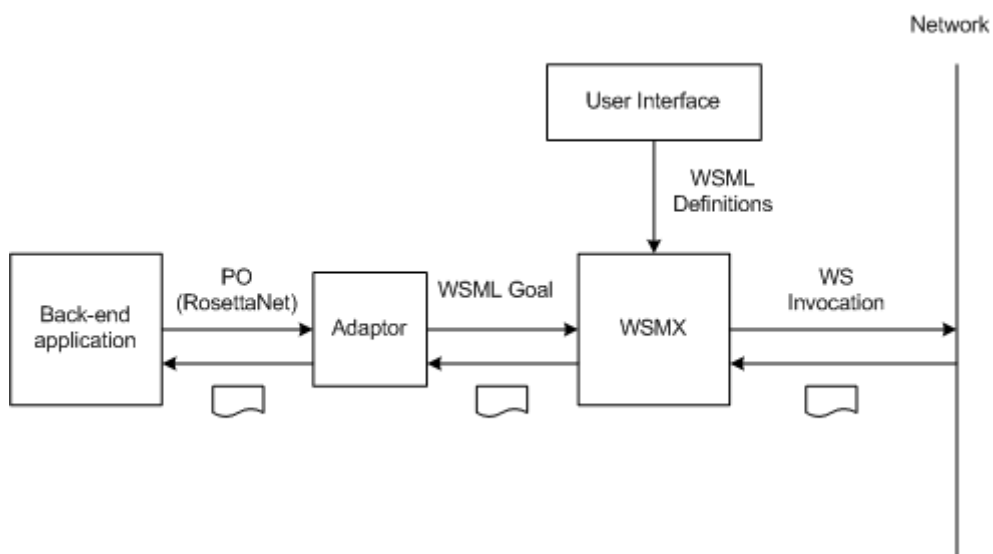


Figure 6: WSMX External View

Figure 7 shows how WSMX itself is a web service and provides a WSDL interface that can be invoked either inside or outside an organisation's network. The diagram shows WSMX receiving three goals, via an adaptor, specifying operations related to RosettaNet PIPs for purchase order handling. In this case WSMX is operating inside the organisation's network.

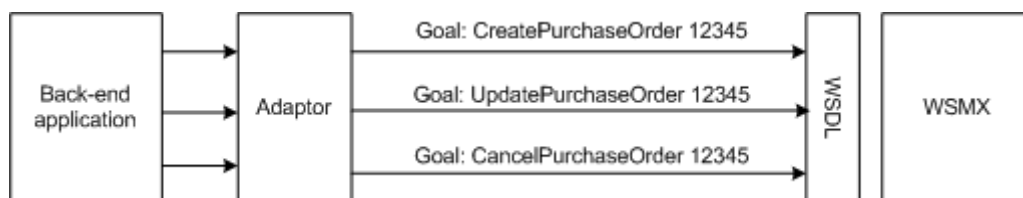


Figure 7: WSMX as a Web Service

Figure 8 [Vasiliu et al., 2004] illustrates multiple WSMX systems could operate together as a virtual distributed system. The figure shows three different WSMX systems. Each WSMX can communicate locally with private back-end application

systems. Additionally each WSMX can communicate with other WSMX systems and web services that have been made known to it through the compilation of the corresponding WSMX descriptions.

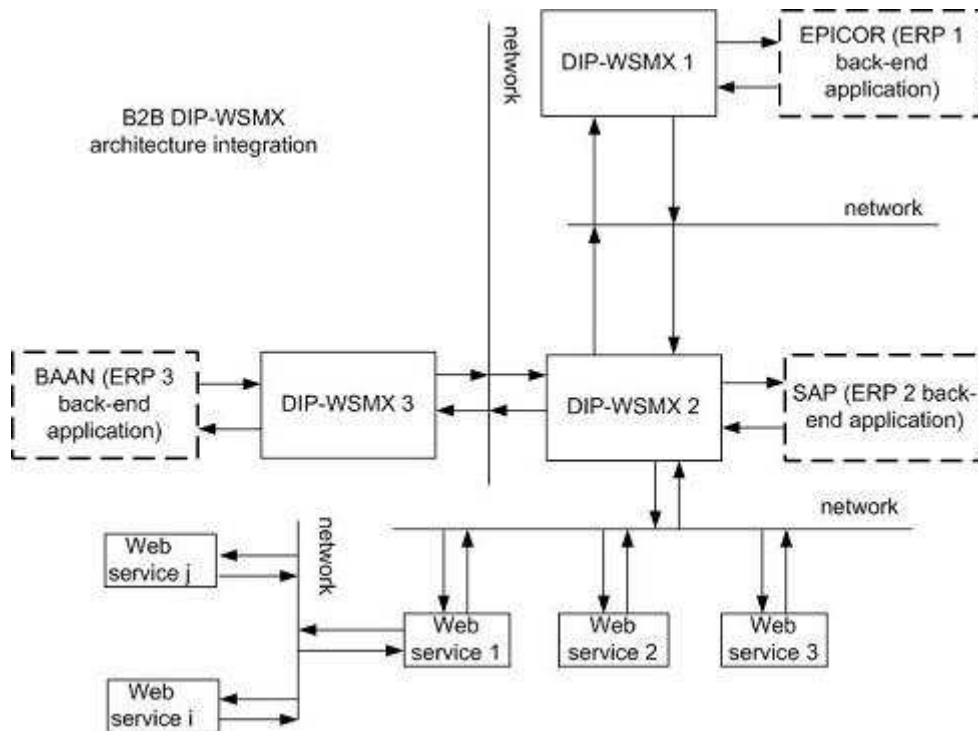


Figure 8 WSMX in a Virtual Distributed System [Vasiliu et al., 2004]

4. Architecture Components and Interfaces

4.1 Architecture

WSMX architecture consists of set of loosely coupled components as presented on figure 9, which in most of the cases can be easily plugged-in and plugged-out from the system (e.g. OO Mediator provided with the reference implementation of WSMX can be easily substituted with the component provided by any commercial company). For each of these components public interfaces are defined, which can be either accessed by other components provided with the reference implementation, or by components provided by independent providers. The WSMX reference implementation aims to be the complete implementation of all of the components, but users of the system may still decide to use components provided by other providers. This section aims to explain the functionality and the purpose of each of the WSMX architecture components.

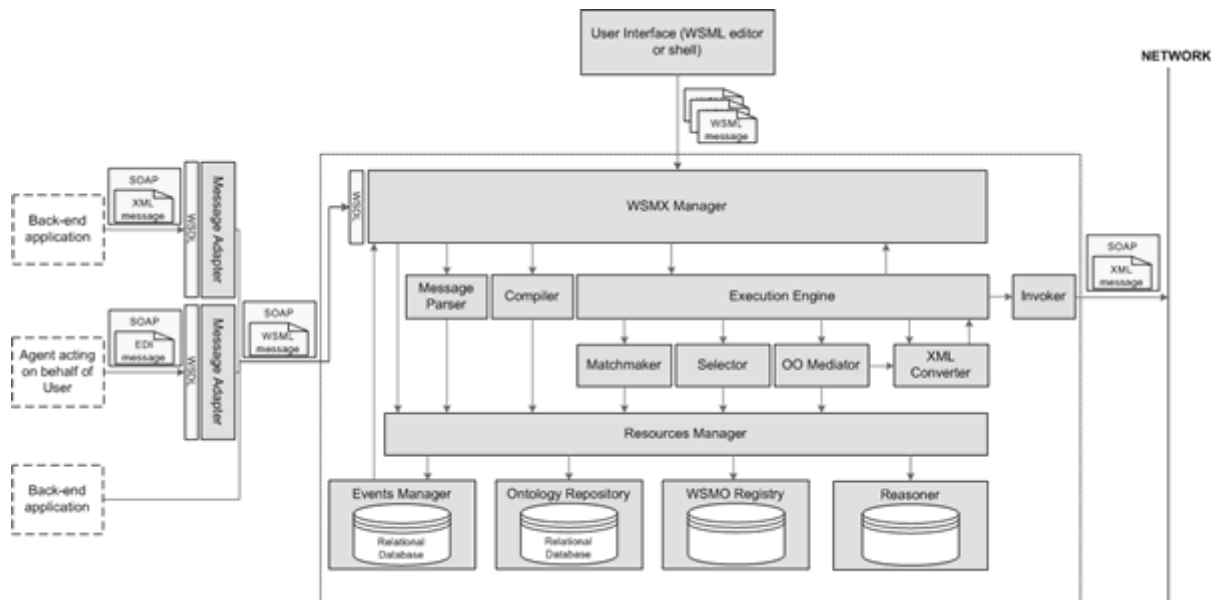


Figure 9: WSMX Architecture (click to enlarge)

There are two types of components in the WSMX architecture - components dedicated to compilation of WSML messages (messages, which describe **capabilities** of the web services and **goals** of requesters e.g. back-end applications or agents acting on behalf of user), and components dedicated to run time execution of the system. Three of the components are specifically dedicated to the compilation process of WSML messages and seventh of them are dedicated to the run time execution process of WSML messages. WSMX Manager and Resource Manager components are shared among compilation and execution functions of the WSMX platform. Components dedicated to compilation of WSML messages are: WSML Editor, Parser and Compiler. Components dedicated to run time execution of the WSML messages are: Events Manager, Execution Engine, Message Parser, Matchmaker, Selector, OO Mediator, XML Converter and Invoker. Externally to architecture remains Message Adapters to allow transforming from any message format (e.g. RosettaNet, EDI, xCBL etc.) to WSML message format. In the future WSMX aims to provide the framework, which will describe how to build new adapters and use them with WSMX. Both WSMX Manager and Message Adapters describe their interfaces using WSDL.

4.2. Components of WSMX Architecture

This section aims to define the functionality (scope) of components of the WSMX system as well as it defines interfaces for these components. Architecture is designed in a way to enable loose coupling of components. This section should be used as a reference by developers of the particular components.

4.2.1 User Interface - WSML Editor

User interface component is used to create new WSML definitions. User Interface (editor) initiates the compilation process of new WSML definition into WSMX platform by sending SOAP messages with WSML definition attachments. The graphical environment should enable owners of web services to describe capabilities of their web services using WSML language. Any new definition can be saved in Ontology Repository (WSMO Registry). Particular implementation of editors will probably also provide the ability to store WSML definitions locally on machines, which run WSML editors. Saving WSML definition remotely on the WSMX platform

allows retrieving definitions, while working in distributed environment. Once the definition is complete it can be compiled into platform. Editor should support browsing of WSMML definitions compiled previously by a user and enabling removing any definitions, which should not be any longer supported (e.g. the web service can become obsolete and their definitions should not be longer available in WSMX). Access control mechanism based on roles, usernames and passwords should be implemented in WSMX to enable restrictive access to some definitions and to reporting mechanism. WSMX aims to come with the set of predefined templates and a mechanism allowing creating new WSMML templates defined by users. Communication between User Interface component is based on a interface based on web services protocols (set of WSDL operations with their inputs and outputs), so both "fat" and "thin" clients editors written in any computer language can be provided to communicate with the WSMX Manager component.

This section described only a set of a suggested functionality, which should be provided by WSMML editors, but developers are advised to come with their own ideas to achieve competitive advantage of their own solutions.

4.2.2 WSMX Manager

WSMX Manager is a gateway to WSMX. It can be accessed by two different WSDL interfaces. The first WSDL interface is an entry point for a run time WSMML messages arriving from agents and back-end application systems and the second WSDL interface is a gateway for WSMML definitions coming from WSMML Editors. WSMX manager processes business logic of the system and observes/records any events happening on the WSMX platform.

Once requested by WSMML Editor, WSMX manager schedules jobs to Compiler. If compilation is successful, WSMX manager as part of an output object, returns back "success" reply. Otherwise it returns "failure" reply back to WSMML Editor. Reporting functionality (e.g. number of messages received, number of generated events, error and exception situations etc.) is exposed through the WSDL interface as well.

WSMX Manager component provides **message listeners**, running as part of the application server, awaiting for any new messages incoming from the network. Each new message is handled by business logic implemented by the Manager component. WSMML messages coming from the back-end application system or from the end-user of the platform are forwarded to Resource Manager to be permanently recorded in the datastore. Any message incoming from the WSMML editor is given to the compiler. Based on the compilation result, the reply is given back to editor, and the objects defined in the WSMML definition are recorded in the datastore as well. Independently from this, WSMX Manager handles logic inside of the system. All events happening in the system are traced by this component and whenever new event happens or the status of the existing event changes, WSMX Manager takes care to make this information permanent.

WSDL interface exposed to back-end application systems is not a stand-alone component itself (because it does not provide any extra functionality), but it is only an endpoint which describes externally exposed functionality of the WSMX platform. This endpoint can be used by any external entities (e.g. back-end applications or information systems from other enterprises), which want to process WSMML messages. There is only one external function/operation provided by WSDL interface to other information systems in WSMX platform called `receiveWSMMLMessage` (see table 1).

Method Summary	
SOAPMessage	<pre>receiveWSMLMessage(SOAPMessage message)()</pre> <p>This operation takes as an input an instance of SOAP message, and returns back another instance of SOAP message. SOAP messages carries WSML payload in form of attachments (e.g. purchase orders), while output message would encapsulate WSML messages with unique identifiers UUID, which are assigned to new, instantiated events in the WSMX platform.</p>

Table 1: WSMX Manager interface run time interface definition

`receiveWSMLMessage` takes as input an instance of SOAP message, and returns back another SOAP message. While input messages would attachments describing in WSML language the **goals** of requesting parties, they output messages returns back SOAP messages with unique identifiers assigned to a request.

4.2.3 Compiler

Compiler checks if the syntax of integration WSML definitions is correct. Only if correct, definitions are decomposed into internal data representation as defined by class model and Compiler request Resource Manager to store them.

Method Summary	
Result	<pre>compile(WSMLMessage message)()</pre> <p>Compiles WSMLMessage</p>

Table 2: Compiler method definiton

4.2.4 Message Parser

Message Parser provides functionality similar to the compiler, but it parses WSML messages processed at run time (messages coming from the back-end applications and from the end-users).

Method Summary	
Result	<pre>parse(WSMLMessage message)()</pre> <p>Parses WSMLMessage</p>

Table 3: Parser interface method definiton

4.2.5 Resource Manager

Resource Manager is a gateway to any persistent storage used by a WSMX system.

4.2.6 Repository

Repository component is used a persistent storage for capabilities, goals and for mediated fragments of ontologies (it will be extended in the future to support selection and discovery components).

During run time, mediation component can use ontology repository to retrieve Mappings Rules, which are necessary to carry mediation.

4.2.7 Events Manager

Events Management component saves events and their status, when they are happening in the system. An externally operations supported by this component and exposed by Resource Manager are presented in table 3 (this table is incomplete - will be updated in the next releases of this document).

Method Summary	
UUID	<code>saveMessage(SOAPMessage message)()</code> Saves WSMX message
WSMXMessage	<code>getMessage(UUID uuid)()</code> Retrieves WSMX message
UUID	<code>createNewEvent(EventName name, RelatedEvent uuid)()</code> Creates new Event
Reply	<code>updateEvent(EventStatus status, EventUUID uuid)()</code> Creates new Event

Table 4: Events Manager interface methods definition

4.2.8 WSMO Registry

A separate deliverable D10 defines WSMO Registry component, which is used by WSMX

4.2.9 Execution Engine

Execution engine controls the process of matchmaking, selecting, mediating and invoking of the Semantic Web Services.

4.2.10 Matchmaker

Matchmaker component match **goals** of requesters with the **capabilities** of web services.

Method Summary	
WebService []	<code>match(WSMXGoal goal)()</code> This method matches capabilities of web services with the goal specified by requester. Method receives WSMXGoal as an argument and returns back an array of web services.

Table 5: Matchmaker interface method definition

4.2.11 Selector

Selection component select only one web service from the collection of web services, which is capable to meet goal of the invoker, based on WSMX sender preferences.

Method Summary	
WebService	<code>select(WebServices[] webServices, Preference[] preferences())</code> Selects the best web service based on the provided preferences.

Table 6: Selector interface method definition

4.2.12 OO Mediator & XML Converter

Mediation engine is a web service so any new mediator can be plugged into WSMX instead of standard mediator provided with the platform. Mediator interface is presented in table 4.

Method Summary	
Payload	<code>mediate(String sourceOntologyID, String targetOntologyID, Payload message())</code> This operation takes as an input identifiers of ontologies and a payload message, and returns back mediated Payload message

Table 7: OO Mediator

4.2.13 Invoker

Invoker component calls web services in other information systems. Invoker converts objects into messages format, packs them into SOAP messages and invokes web service provided by other information systems.

Method Summary	
InvocationResult	<code>invoke(WebServices webService())</code> Invokes selected web service.

Invoker 7: Invoker

5. Conclusions and Future Work

At this stage this document presents the general overview of the WSMX architecture and interfaces exposed by each of the components. Future work will include the detail specification of the functionality of each of this components, class model for objects of the system and data model used by the persistent storage.

References

[Bussler, 2003] C. Bussler, B2B Integration, Concepts and Architecture, Springer-Verlag, 2003, ISBN 3-540-43487-9

[ebMS, 2002] Message Service Specification, Version 2.0, OASIS ebXML Messaging Services Technical Committee, 1 April 2002, <http://www.ebxml.org>

[Herzog et al., 2004] R. Herzog, H. Lausen, D. Roman, M. Stollberg, P. Zugmann. WSMO Registry, <http://www.wsmo.org/2004/d10/v0.1/>

[Oren, 2004] E. Oren. WSMX Execution Semantics,
<http://www.wsmo.org/2004/d13/d13.2/v0.1/20040531/index.pdf>

[Oren et al., 2004] E. Oren, M. Kifer, J. de Bruijn, H. Lausen, D. Roman, M. Felderer, BNF grammar for WSMML user language,
<http://www.wsmo.org/2004/d16/d16.1/v0.2/20040418>

[Roman et al., 2004] D. Roman, H. Lausen, U. Keller. Web Services Modeling Ontology - Standard (WSMO - Standard),
<http://www.wsmo.org/2004/d2/v02/20040306/>

[RosettaNet] The RosettaNet consortium for open eBusiness standards and services, <http://www.RosettaNet.org>

[Stollberg et al., 2004] M. Stollberg, H. Lausen, A. Polleres, R. Lara, U. Keller, M. Zaremba, D. Fensel, M. Kifer. WSMO Use Case Modeling and Testing,
<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040524/>

[Vasiliu et al., 2004] L. Vasiliu, M. Moran, C. Bussler, WSMO in DIP, WSMO Working Draft v0.1, 7 June 2004, Digital Enterprise Research Institute, available from <http://www.wsmo.org/2004/d19/d19.1/v0.1/>

[W3C Note, 2001] Web Services Description language (WSDL) 1.1, W3C Note 15 March 2001, <http://www.w3c.org/TR/wsdl>

Acknowledgment

The work is funded by the European Commission under the projects DIP, Knowledge Web, Ontoweb, SEKT, and SWWS; by Science Foundation Ireland under the DERI-Lion project; and by the Austrian government under the CoOperate programme.

The authors would like to thank to all the members of the WSMO working group for their advises and inputs to this document.



[webmaster](#)