WSMX Deliverable

D13.4 v0.2
WSMX Architecture

WSMX Final Draft – 13 June 2005

Authors:
Michal Zaremba, Matthew Moran, and Thomas Haselwanter

Editors:
Michal Zaremba and Matthew Moran

Final version:
http://www.wsmo.org/TR/d13/d13.4/v0.2/20050613
Latest version:
http://www.wsmo.org/TR/d13/d13.4/v0.2/
Previous version:
http://www.wsmo.org/2004/d13/d13.4/v0.2/20050412

Copyright © 2005 DERI ®, All Rights Reserved.
DERI liability, trademark, document use, and software licensing rules apply.

Abstract

This deliverable describes Web Services Execution Environment Architecture - WSMX Architecture. The document provides both a high-level overview of the necessary system components and the interactions between them (conceptual architecture) as well as a low-level definition of component interfaces, connectivity and the applied events mechanism used by the current reference implementation.

Copyright © 2005 DERI ®, All Rights Reserved.
DERI liability, trademark, document use, and software licensing rules apply.

Contents

1 Introduction 4

1.1 Motivation for the Architecture . . . 4
1.2 Architecture Style . . . 5
1.3 Architecture versus Design & Implementation . . . 6
1.4 Organisation of the Document . . . 6

2 Architecture Overview 7

2.1 A P2P Network of WSMX Nodes . . . 7
2.2 Overview of a single WSMX Node: SOA vs Layers . . . 8

3 Structural View 10

3.1 Architecture Components . . . 10
3.1.1 Core Component . . . 10
3.1.2 Resource Manager . . . 11
3.1.3 Service Discovery . . . 11
3.1.4 Non-functional Selector . . . 12
3.1.5 Negotiation (Functional Selector) . . . 12
3.1.6 Data Mediator . . . 12
3.1.7 Process Mediator . . . 12
3.1.8 CommunicationManager . . . 13
3.1.9 Choreography Engine Interface . . . 13
3.1.10 Parser Interface . . . 13
3.1.11 Orchestration Interface . . . 13
3.1.12 Web Service Modelling Toolkit Interface . . . 14
3.1.13 Reasoner Interface . . . 14

4 Behavioural View 15

4.1 Standardization of WSMX External Behaviour . . . 15
4.2 Dynamic Execution Semantics . . . 17

5 Summary 19

6 Acknowledgement 21

7 Appendix A. Changelog 21

8 Appendix B. Component Interface Tables 22

1 Introduction

This deliverable describes a reference architecture of an execution environment for Semantic Web Service (SWS) called *Web Service Execution Environment* (WSMX). WSMX is a comprehensive software framework for the discovery, selection, mediation, invocation and interoperation of Web services based on their semantic descriptions. It is a reference implementation of the Web Services Modeling Ontology (WSMO) [8] which defines a conceptual model for various aspects of Semantic Web services. By implementing an environment supporting service requester goals and goal driven discovery and invocation, WSMX enables service requesters and providers to come together to achieve specific tasks even when these service requesters and providers are not aware of each other in advance and may have significant differences in their data and public behaviour models.

The work described in this deliverable was carried out jointly between the DERI-Líon project funded by Science Foundation Ireland (SFI) and the DIP project funded by European Union's IST programme (no. FP6 - 507483). The editors would also like to thank the members of the WSMO, WSML, and WSMX working groups for their advice and input into this document.

1.1 Motivation for the Architecture

In general where advances in computer and information science occur their uptake is greatly assisted by having tools and frameworks that allow both software engineers and potential users to try out the new technology. WSMX aims to address both these interest groups. For software developers interested in developing functional components that exploit the extra knowledge made available in Semantic Web service descriptions, WSMX is provided as open-source software with a license intended to make it as easy as possible for anyone to use the software for research or commercial purposes. The WSMX architecture includes a mechanism to allow components e.g. dicovery, to be developed by a third party and plugged in to the WSMX framework with minimum requirements on the component's developers.

From the perspective of potential users of Semantic Web services, WSMX provides a means to focus on maximising the benefits offered by being able to unambiguously describe the functional and non-functional aspects of Web services without having to invent a framework that knows how to handle those descriptions. WSMX provides the glue that brings the features promised by Semantic Web service technology to life. In the same way as middleware systems hide the detail of how they link autonomous heterogeneous applications and systems together, a Semantic Web services environement such as WSMX hides the detail and complexity of how the requester and provider of Web services can communicate with each other even where they do not know each other in advance, speak different languages and expose different behaviours at their service interfaces. WSMX has the potential to offer a credible lightweight solution to the problems faced in integrating applications and business systems across the Web.

There are no comparable architectures of execution frameworks for Web services based on WSDL [3] descriptions. Specifications such as SOAP [6], WSDL and UDDI [2] provide powerful underlying tools which WSMX also uses. However the technologies for traditional Web service discovery, composition and invocation are limited by the absence of semantics which keeps much of the responsibility for linking Web services together on the shoulders of the developers

creating the services, the clients and, in the case of composition, the designers of the workflow descriptions (e.g. using BPEL or Microsoft Biztalk).

1.2 Architecture Style

Software architectures are concerned with providing a high level description of a system explaining how the system, from a design perspective, addresses the problems it needs to solve. It takes account of the immediate and longer term requirements of the system, the existing technologies, new technologies that are coming on-streeam and provides the vision of how the system can be designed to achieve its goals. WSMX is a service oriented architecture (SOA) that adopts the two principles below taken from the Web Services Modelling Framework (WSMF) [4].

- **Component decoupling**
  This is a principle of SOAs that emphasises the strong de-coupling of the various components that realize a software system. Making components self-contained supports a clear separation of concerns. Each component has a well defined functionality that can be utilized by other components.

- **Standardization of external behaviour of components**
  The external interfaces of the components within the WSMX architecture are not expected to change very often. By having well defined component interfaces and behaviour descriptions, we separate the implementation of the individual components from the operation of the system as a whole. One of the long-term aims for WSMX is the support of dynamic execution semantics. Execution semantics provide a formal description of the operation of a system. By dynamic execution semantics, we mean that it should be possible to load a description of how WSMX should operate at runtime without having to restart the system. For example, a particular deployment of WSMX might never require data mediation because all data and processes available to the WSMX are homogeneous. In such a case an execution semantics might be defined to ignore this component. This can be relevant in systems where, for optimization reasons, even a pass-through call to a component where the component takes no action might be too time consuming.

  We can also imagine scenarios where a deployment of WSMX is provided with a new component whose functionality was not considered in the overall WSMX design - it may be a specific requirement for that specific deployment. This could lead to a requirement to extend the principle of standardized component interfaces in the future.

An initial set of components have been identified to make WSMX a reality and to provide the basic functionality required for executing SWS, namely discovery, mediation, composition and invocation. By adopting the SOA style and by building a highly flexible messaging infrastructure at its heart, WSMX is well placed to include new components offering functionality as required. Indeed the possiblity of adding new components on the fly is a strong motivational factor in the ongoing design and development of WSMX.

So far we have talked about how WSMX looks on the inside but a second crucial aspect of WSMX is how it adapts to a distributed system model. In other words, by design WSMX is not intended as a standalone centralised system. Rather each deployment of WSMX is intended to be considered a node in a peer to peer (P2P) network. In this network, each node of WSMX will have

the same lightweight event driven SOA at its core. Components will be plugged into any any node of WSMX as required. Architecturally speaking, it is not necessary that any particular functional component be deployed on the same physical server as WSMX itself although, in the short to mid-term, phase of development, this is the most likely design that will be implemented. By viewing WSMX as a federation of nodes in a P2P network, additional approaches for service discovery and composition open up.

1.3 Architecture versus Design & Implementation

This deliverable provides both a high level conceptual architecture with an overview of system components and their functionality, as well as come detail on more technical aspects of the system design. Despite the inclusion of these more detailed technical design aspects of WSMX , this document is not intended as a technical system documentation or as a programmers reference. A detailed technical design document describing the implemention aspects of WSMX will be included with the open source code and documentation.

1.4 Organisation of the Document

The rest of this document is organised as follows. Section 2 provides a high level view of the architecure both in terms of the functional components of WSMX and how WSMX can operate in a P2P network. Section 3 provides a structural view of WSMX with functional descriptions of the components currently defined for a WSMX node and the definition of their interfaces. Section 4 describes the execution semantics while Section 5 discusses the behavioural view of WSMX - how the execution semantics (described in section 4) are reflected in the architecture. Section 6 provides a document summary.

2 Architecture Overview

This section is intended to provide an overview of two aspects of WSMX architecture brought up in the introduction. The first is the idea that each deployment of WSMX is intended to be considered as a node in a P2P network. The second aspect we look into here is how the architecture of WSMX relates to the style of grouping system functionality into layers where the interaction between layers is well defined.

2.1 A P2P Network of WSMX Nodes

In general a P2P network is one in which there is no central controling entity. Each node in the network can communicate with any other node to co-operate together in performing some task(s). The most well known P2P networks in recent years have been concerned with sharing video and audio files among a community of users. Each user has software enabling the P2P network on their machine and the network addresses of machines in the network are available to all users of the P2P services. In reality, many networks advertised as P2P have a central node used for member management but this is not relevant to our point.

P2P networks offer the advantage of allowing data and functionality to be split up across the network while at the same time allowing peers to co-operate in achieving tasks requiring this data or functionalty even though it may be distributed across different locations. From the WSMX perspective, a typical use-case would be for service discovery. A WSMX node receives a goal definition and uses the discovery component at that node to search for matching services. The search takes place across Web service descriptions that are known to the service registry for that node. Where no match is found, in a P2P network, the first WSMX node could forward the discovery request to another WSMX node known to it. Again if the second node has no success, it could possibly forward the request again to a third WSMX node and so on. We envisage that there would need to be policies in place to control the depth of recursion that would be allowed.

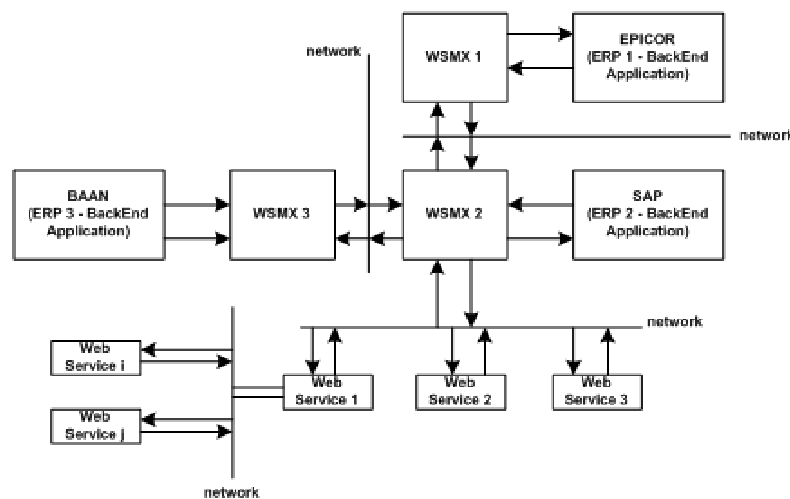

Figure 1: WSMX Nodes in a P2P Network
[9]

As each WSMX node would be itself a Web service with a WSMO semantic description, the invocation of a component in another WSMX node in the network would be the same as for any other Web service invocation. Figure 1 shows how a simple P2P WSMX network with three nodes would look. In the figure each WSMX has a different backend system associated with it. For the discovery example mentioned earlier, each WSMX node could have different service descriptions in its registry corresponding to specific functionality offered by the associated backend application.

A P2P network architecture of WSMX nodes is a long term vision for WSMX. It is likely that the development of this aspect of WSMX will be driven by the requirements of service discovery as Web services are situated all around the Internet and it is extremely unlikely that one single global registry of service descriptions will ever exist.

2.2 Overview of a single WSMX Node: SOA vs Layers

The most dominant style adopted by WSMX is that of a Service Oriented Architecture, a software system consisting of a set of collaborating software components (or services) with well defined interfaces that, by exchanging messages, together perform a set of tasks. It is not necessary for the components to necessarily live in the same address space, nor even in different address spaces on the same machine. They may very well live on physically separated servers communicating over a network channel through multiple communication protocols.

Loose coupling of system components is the cornerstone of SOAs. This is reflected in WSMX where independent pieces of functionality are provided in components. Each WSMX component provides a service that consists of a logical unit of system code, application code, persistency layers and a public interface to acccess the service. In short anything that as a unit can carry out an application-level operation. Services are often characterized by exactly these operations, which they provide to other components. Preferably these services have machine-processable meta-data descriptions.

The internal communication model for WSMX is based on an event-based messaging mechanism. This enables the good practice of implementation-hiding, where the implementation of a service should be of no concern to an entity using the service. In the long term the decoupled nature of WSMX allows for load-balancing and clustering of components for optimization. The benefit of the approach is increased flexibility, better extensibility and dramatically improved reusability. Having an SOA facilitates, does not guarantee, that these goals will be achieved but is a good basis from which to start.

Layering is a very common architectural style where the functionality of a system is decomposed into logical layers. Each layer corresponds to an abstraction of some aspect of the system. In most cases higher up layers can request services from lower layers but not the other way around. Software systems have progressed from initial one layer systems (e.g. mainframes) through two layer systems (e.g. simple client server) through to n-layer systems where multiple layers representing the business logic of the system are defined. An example of a layered architecture was used in version 0.1 of the WSMX architecture and is shown in figure 2

In figure 2 the user interface included the functionality for registering service descriptions with WSMX and for displaying information on the execution of the system. The logic layer included the functional components and the communication layer was responsible for handling the exchange of messages between

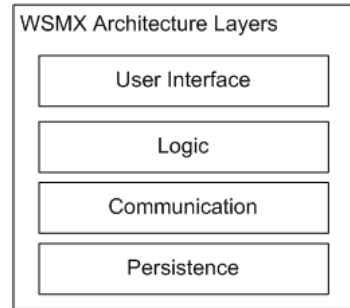

Figure 2: Layers Used in WSMX v0.1 Architecture

WSMX and the requesters and providers of Web services. Finally the persistence layer consisted of the database system used for data storage and retrieval.

With the adoption of a SOA for WSMX the layered pattern used for WSMX v0.1 is less appropriate. The functional components previously intended for the logic layer e.g. discovery, mediation etc. are now avalable to WSMX as internal services. WSMX has a microkernel at its core which manages the mechanism for adding, removing and updating functional components and which implements an event based message exchange mechanims. The functionality of the user interface is still a discrete layer of functionality but it also appears to WSMX as a service that can accept and receive messages. This is a difference with layered models that allow higher level layers to call lower level layers but not the other way around. The communication layer is now also provided as a set of components acting a services. In the case of persistency, WSMX still uses an external data storage system for storing functional and non-functional data. However access to the persistency is again represented by a component that communicates with the event system in the same way as the other functional components. Figure 3 illustrates this structure. In this figure the encircled boxes can be considered to represent clusters of related functionality in some ways similar to the idea of grouping related functionality in layers.

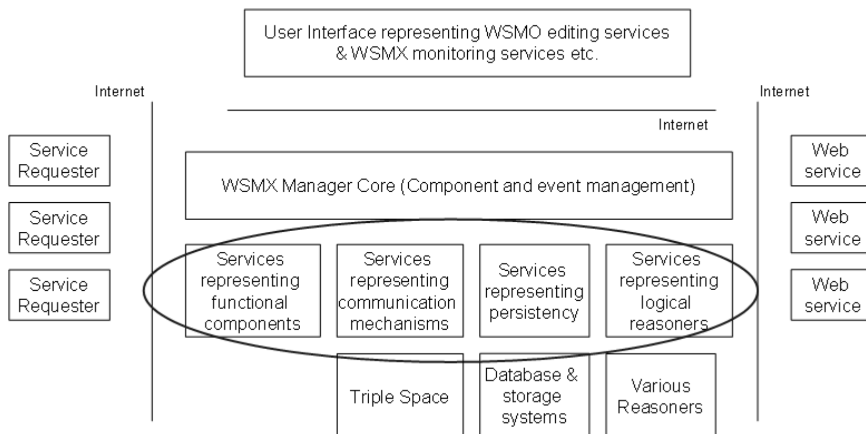

Figure 3: Overview of Services in WSMX v0.2 SOA

3 Structural View

This section takes a more detailed look at the various components that have been defined as part of the architecture for WSMX. The components discussed here are those that are included for the current version of WSMX. To acheive its aims of providing an industrial strength integration framework based on Semantic Web services, WSMX will need many more components. In particular, issues like security, transactionality, reliability, negotiation, monitoring etc. must be addressed. The developement plan for WSMX intends to incrementally add such functionality.

This section is organised as follows. We first look at the overall structural view of the WSMX architecture. The following subsections describe the intended functionality of each of the components. The descriptions are not precise specifications - this will be the responsibility of the individual component developers. However, the interfaces described for the components in these subsections is the exact interface that the implementations of the components must support. This refers to one of the two principles adopted by WSMX described in the introduction, namely, the *standardization of external behaviour of components*.

The next sections provide a short description of each of the functional components in the current version of the WSMX architecture. The interface for each component is defined but is included separately in Appendix B for document space and formatting reasons.

3.1 Architecture Components

The WSMX architecture consists of a set of loosely-coupled components as presented in Figure 4. These can be plugged-in and plugged-out from the system. The components that are provided with the WSMX reference implementation can be easily replaced by other components, e.g. by those provided by third parties.

For each component, public interfaces are defined, that can be either accessed by components provided with the reference implementation, or by components provided by independent component providers. The WSMX reference implementation provides the complete implementation of all of the components, but deployments of WSMX are free to use components provided any party as long as they implement the defined interface.

3.1.1 Core Component

The Core Component is the central part of WSMX, and all the other components are managed by it. All interactions between components will be coordinated through the Core Component. The business logic of the system, the events engine, the internal workflow engine, the distributed components loading, etc. will all be subcomponents of the Core Component. At this stage the Core Component is the central module of WSMX. In the future, in order to ensure increased reliability of WSMX, a clustering mechanism for distributed Core Components might be developed. For resource intensive operations, many instances of the Core Components can be instantiated on several machines to create a WSMX cluster.

Interface. There is no specific interface defined for the Core Component. Although it is possible to envisage an implementation of WSMX in which components communicate directly without use of the Core Component at all, it is

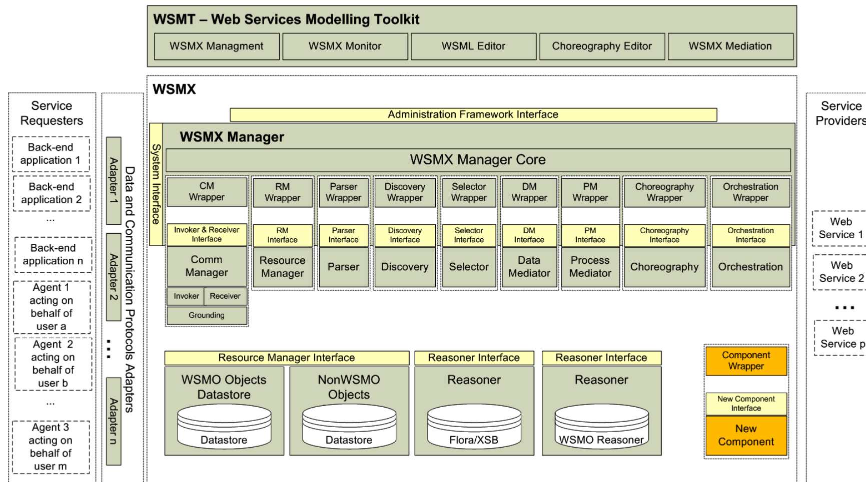

Figure 4: WSMX Architecture

recommended to have the Core Component coordinate and manage all other components. No specific interface is enforced at presesent.

While at this stage the specification is under development, the interface of the Core Component is not standardized. This is subject to change in the future, as federated WSMXs are planned, where the Core Components might be distributed (in the current architecture design, the core component remains the only centralized entity of the whole system) and provided by different vendors.

In the reference implementation, an intermediate layer of wrappers is provided in order to interface the Core Components with all the other components. To enable one component of the system to communicate with another component, its wrapper must implement the interfaces of the components with which this component communicates.

3.1.2 Resource Manager

The ResourceManager is the interface for WSMX persistent storage. The component implementing this interface is responsible for storing every data item WSMX uses. The WSMO API provides a set of Java interfaces that can be used to represent the domain model defined by WSMO. WSMO4j[1] provides both the API itself and a reference implementation but it is not a pre-requisite that implementations of the Resource Manager use WSMO4j.

Currently WSMX defines interfaces for six repositories. Four of these repositories correspond to the top level concept of WSMO i.e. Web services, ontologies, goals, and mediators. The fifth repository is used by WSMX for non-WSMO data items e.g. events and messages. Finally the sixth repository stores WSDL documents used to ground WSMO service descriptions to SOAP or SOAP/HTTP. In the longer term the WSMX working group recognises that ebXML [7] and UDDI repositories can be used for WSMX data persistence.

Interface. Tables 3, 4, 5, 6, 7 and 8 describe these interfaces.

3.1.3 Service Discovery

The WSMX Dicovery component is concerned with finding Web service descriptions that match the goal specified by the service requester. It is not in

[1] http://wsmo4j.sourceforge.net/

the scope of this architecture document to specify how the discovery is implemented but the intent is that the WSMO description of the goal a user wishes to achieve (described in terms of a desired capability with preconditios, assumptions, effects and postconditions) is matched to the WSMO description of Web services known to WSMX (described in terms of offered capabilities). The discovery component returns a (possibly empty) list of Web service descriptions.

Interface. Table 9 describes the interface.

3.1.4 Non-functional Selector

The Non-functional Selector is a component used to select the most suitable service from a list of matching services matched by discovery. For example, a service requester may define preference for the selection of the most suitable discovered Web service. If discovery results in more than one service that satisfies the goal, the selection interface is used to chose one based on specified preferences preferences. Selection does not involve making an invocation on the service.

Interface. Table 10 describes the interface.

3.1.5 Negotiation (Functional Selector)

The Functional Selector is used to select the most suitable service from a list of matching services where the services must be actually invoked as part of the selection procedure. For example a discovered book selling service may match all the rquire preferences but still may need to be actually invoked to determine if the required book is available from the specific service.

Interface. Table 11 describes the interface.

3.1.6 Data Mediator

A WSMX DataMediator component has the role of reconciling the data heterogeneity problems that can appear during discovery, composition, selection or invocation of Web Services. This interface is dedicated for the runtime phase mediation process. The run-time component implementing this interface has the role of retrieving from storage the already-created mappings, to transform them into rules, and finally to execute them against the specified incoming instances (input) in order to obtain the target instances (output). Since the mappings represent the connection point between the two sub-components (design-time and run-time) one of the dependencies for the run-time component relates to the mapping storage level. Another crucial dependency relates to the reasoning system used for executing the rules in the final stage of the mediation process.

Interface. Tables 12 describes the interface.

3.1.7 Process Mediator

A WSMX Process Mediator has the role of reconciling the public process heterogeneity that can appear during the invocation of Web Services. That is, ensuring that the public processes of the invoker and the invoked Web Service match. Since both the invoker and the Web Service publish their public processes as choreographies, and the public processes are executed by sending/receiving messages, the Process Mediation Component will deal with reconciliation of message exchange patterns based on choreography.

Interface. Tables 13 describes the interface.

3.1.8 CommunicationManager

The Communication Manager is responsible for dealing with the protocols for sending and receiving messages to and from WSMX. Its external behaviour is accessed through the Invoker and Receiver interfaces. The WSMX Receiver interface expects the contents of all messages it receives to be expressed in WSML. Each WSML messages may represent a goal to be achieved or be a message corresponding to a choreography or orchestration instance that already exists. The Communication Manager accepts the message and handles any transport and security protocols used by the message sender. The execution semantics of WSMX determine how the WSML message should be handled based on the defined execution semantics of the system.

The Invoker is used by the execution semantics of WSMX when a Web service needs to be invoked. The invoker receives the WSMO description of the service and the data that the service expects to receive. It is responsible for making the actual invocation of an operation on a service. In the majority of existing Web service implementations, this means ensuring that the semantic description of both the data and the behaviour of the Web service are grounded to the corresponding WSDL descriptions. It is anticpated that a separate grounding component to work with the invoker will be required in future versions of WSMX.

The external behaviour of the system is defined in a new WSMO4j interface called EntryPoint. The intent is that the EntryPoint interface be implemented by any WSMO compliant Semantic Web services environments to facilitate seamless run-time integration of these systems. The EntryPoint interface represents the external behaviour of the system. It is currently implemented by the Communication Manager but is described separately in the section 4.1.

Interface. Table 14 describes the Invoker interface and table 15 describes the Receiver interface.

3.1.9 Choreography Engine Interface

A WSMO Choreography defines how to interact with the web service in terms of exchanging messages the so-called communication patterns.

Interface. Table 16 descibes the Choreography component interface.

3.1.10 Parser Interface

The Parser checks if the syntax of receieved WSML descriptions is correct. Only if correct, the descriptions are parsed into WSMO4j data object used as the internal data representation for WSMO.

Interface. Table 17 descibes the Parser component interface.

3.1.11 Orchestration Interface

The functionality and interface for the WSMX Orchestration are not defined in this version of the document.

3.1.12 Web Service Modelling Toolkit Interface

The Web Services Modeling Toolkit (WSMT) provides user interface tools for WSMX management and monitoring including a WSML editor. The architecture and design of WSMT is described in the WSMX deliverable D9.1. Its current documentation is available at [5].

3.1.13 Reasoner Interface

There is no agreement on a stable Resasoner interface so far. Once work on WSMO reasoner implementation will finalize, its interface will become standardized through WSMX infomodel.

4 Behavioural View

WSMX is a Service Oriented Architecture (SOA), which means that it is a software system consisting of a set of collaborating software components with well defined interfaces that together perform a task. These components do not necessarily live in the same address space, not even in different address spaces on the same machine, instead they may very well live on different continents communicating over a network channel through multiple protocol stacks. This situation creates its own unique demands, born out of latency, memory access, concurrency and failure of subsystems, which the architecture must be able to cope with. All these aspects are going to be addressed in subsequent steps during designing and implementing reference implementation of WSMX.

SOAs differentiate themselves from other distributed systems through the concept of loose coupling brought to its extremes. Strong de-coupling of the various components that realize an e-commerce application is one of major feature of WSMO. In WSMX conceptually independent pieces of functionality has been grouped in components. Each of WSMX components provides services - a single of which is a logical unit of system code, application code, persistency layers, in short anything that as a unit can carry out an application-level operation. Services are often characterized by exactly these operations, which they provide to other components. Preferably services have descriptions in machine-processable meta-data. The architecture described in this document describes an internal communication mechanism for an SOA based on an events-based mechanism. However one can also envision an infrastructure coordinated without events.

In WSMX, communication between components takes place through events. It enables the good practice of implementation-hiding, in which the implementation of a service should be of no concern to the client of the service. All of this together should result in increased flexibility, better extensibility and dramatically improved reusability. in particular it facilitates the addition, upgrade or removal of functional components while at the same time maintaining the stability of the syste. Situations where new components are added to WSMX that were not considered during the initial design are examples of where there would be benefits for a flexible, dynamic mechanism of defining how the system operates. This is what we mean when we discuss the topic of *dynamic execution semantics* for WSMX.

4.1 Standardization of WSMX External Behaviour

Functionality provided by the WSMX system as the whole can be described in terms of its entry points - the standardized interfaces of the system enabling communication with any external entities requesting services from the system. More details on system functionality can be found in the execution semantics document [11], which formally specifies the desired operational behaviour of WSMX, serving not only as a reference for developers but also as a means of validation and model-checking.

In the current version of the system we define execution semantics with three possible branches each of them starting with one entry point. These three mandatory entry points must be available in each working instance of any system, which is WSMX compliant. Entry points also define the required functionality of WSMX compliant systems. By selecting a given entry point a client request the functionality that is associated with it and the corresponding execution semantic is triggered to operate on the set of components that is

necessary to fulfill the request. While the entry points are about what WSMX does, the execution semantics that are associated with it are about how WSMX does this. These three obligatory entry points have signatures described in table 1:

Table 1: Entry Point Interface

Method Summary	
Context	achieveGoal(WSMLDocument wsmlDocument)
Context	getWebService(WSMLDocument wsmlDocument)
Context	invokeWebService(WSMLDocument wsmlDocument, Context context)

We provide a short description of each of these methods:

Context achieveGoal(WSMLDocument)
Any external entity, which expects to get its goal realised might wish to provide a formal description of a Goal (in WSMO terms). The requester receives a Context which he can use as a correlation identifier. WSMX discovers, selects and executes a Web service on behalf of the service requester. The service requester might receive a final confirmation or data returned from the invoked service. Two ways in which this can be carried out are (i) the requester provides an endpoint which can be invoked by WSMX to send the return data and (ii) WSMX sends the return data to an intermediary adapter which may return the data to the requester as if all operations carried out by WSMX had happened synchronously. In general WSMX is based on an asybchronous model for service invocation.

Context getWebService(WSMLDocument)
The getWebService entry point addresses the a likely realistic scenario where the service requester might wish to consult WSMX to find a single Web services capable of satisfying their Goal. In this call, service requester provides Goal and and expects to get back a context in return. This context can then be used by the requester in a subsequent call to WSMX to make the service invocation.

Context invokeWebService(WSMLDocument, Context)
The invokeWebService entry point allows a requester to invoke a Web service using a context established at some eralier points.

An additional interface called WSMORegistry is defined for WSMO compliant execution environments for operations related to the storage, retrieval and removal of WSMO objects from the implemented WSMO registry. The types used in this interface are defined in WSMO4j. The methods of the interface are described in table 2:

Additionally to these three entry points, we plan that WSMX will provide an engine to support dynamic execution semantics enabling execution of any formal description of system behaviour. In this way we can also define multiple operational paths through the system independent of the code used to implement the functional components.

Table 2: Entry Point Interface

Method Summary	
void	store (WSMLDocument wsmlMessage) The Store method provides an administration interface for the system enabling to store any WSMO related entities (like Web Services, Goals, Ontologies) and making them available for other parties using WSMX system.
Set<Identifiable>	retrieve(Namespace namespace) Retrieve the set of all Identifiables for the specified Namespace.
Identifiable	retrieve(Identifier identifier) Retrieve the Identifiable corresponding to the specified identifier.
void	remove (WSMLDocument wsmlMessage) Remove the WSMO elements contained in the input WSML document.

4.2 Dynamic Execution Semantics

During design process of WSMX several steps have been undertaken, including describing its conceptual model, specifying its formal system execution semantics and designing its architecture. By providing the conceptual model the common reference vocabulary for the development team has been defined, which has been meant to be used at different phases of the project. Execution semantics - the formal specification of the operational behavior is normally used for a number of reasons during software development.

As described by [10] in the context of WSMX we were initially interested in modeling the execution semantics of WSMX in such a way that developers understand the system, that certain properties of the system can be inferred and that it enables model-driven execution of the system's components. Execution semantics also provide a clean way to separate the functional semantics from the operational semantics, which makes the introduction of completely new components or changes in how components interact possible without affecting clients of WSMX.

The WSMX Architecture delivered a detailed description of the overall system, components interfaces and specification of the functionality expected from the particular components. One of the key design decisions we undertook for WSMX has been to keep individual components decoupled one from each other and to enable components distribution across the network. The development of the high-speed networks makes it possible to distribute services across various machines achieving complete functionality by combining these partitioned and distributed resources. The effective way to exploit advantages provided by network and achieve system scalability requires partitioning system functionality into coarse-grained components with well defined interfaces which can provide a small but complete functionality required by this system.

The initial version of WSMX included only one possible execution semantic which was hard-coded into components of the system. This approach very quickly showed weaknesses of WSMX design revealed by new requirements and use cases provided by potential users of the system. The initial architecture approach was refined to accommodate a mechanism to incorporate new compo-

nents and methods enabling adding and removing any new formal definitions of execution semantics describing operational behavior of the system without the need to recompile the whole platform any time such new execution semantic becomes available. The first step to enable new execution semantics in the system has been the design of execution semantics (or in SOA terminology - "business processes") - a group of business activities undertaken by a management component in pursuit of a common goal.

While for WSMX we used Petri nets [1], we do not restrict dynamic execution engine to any specific formalism. One tool used, CPNTools[2], make it possible to model so-called high-level Petri-nets, extending classical Petri nets with hierarchy, color and time. The tool used for definition of business process must make it possible to verify certain properties of the model; it must check some simple properties such as syntactical correctness, unreachable states or unsatisfiable conditions. While we define process, we made an abstract declarations that services will be requested by using services interfaces (see figure 5), but actually we do not bind the process to the concrete service, which is going to be invoked during run time.

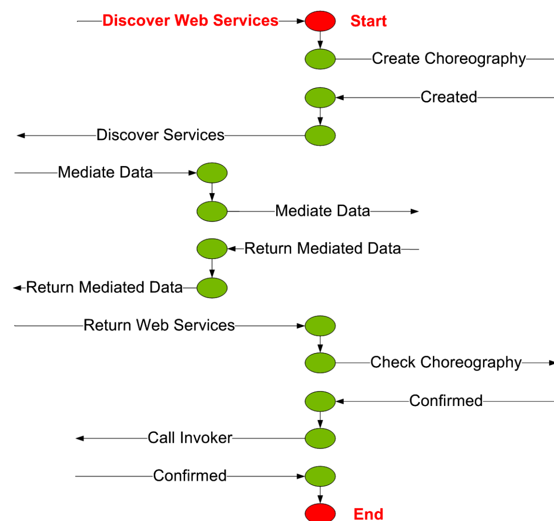

Figure 5: Sample Abstract Execution Semantics Definition

Apart from process definition, we recognized another requirement to enable dynamic execution semantics - runtime ability to plug-in and plug-out components. In WSMX we enable reconfiguration, management, and monitoring of available software components. We maintain that system must be capable to host deployable components and to reconfigure them during initialization and as during runtime.

The persistent configuration support is responsible for loading the various components into memory, for which sophisticated and decentralized configuration systems provide the necessary flexibility. Configuration of the individual components can either be deliver as a descriptor file or as code annotations. Having abstract process definition and components installed in the system, we generate wrappers for components (see figure 6). The purpose of the wrapper is to separate components from transport layer for events.

As mentioned before although we also recommend building the communication inside Semantic Web Services architecture based on event paradigm, the system components can be coordinated without events. WSMX is an event-

[2]http://wiki.daimi.au.dk/cpntools/cpntools.wiki

based system, consisting of many wrappers that communicate using events. Wrappers exhibit an asynchronous form of communication. One wrapper raises an event with some message content and another wrapper can at some point in time consume this event and react upon it. Components offering services to WSMX remain unaware of event infrastructure, while they solely communicate with their own wrappers, while event consumptions and production is taking place only on a wrapper level. In fact the component themselves are even unaware of their wrappers since they are the process that drives the component, acting as a kind of distributed schedulers. The transport mechanism has been also decoupled from the system by using Transport interface, which hides details of event transport mechanism.

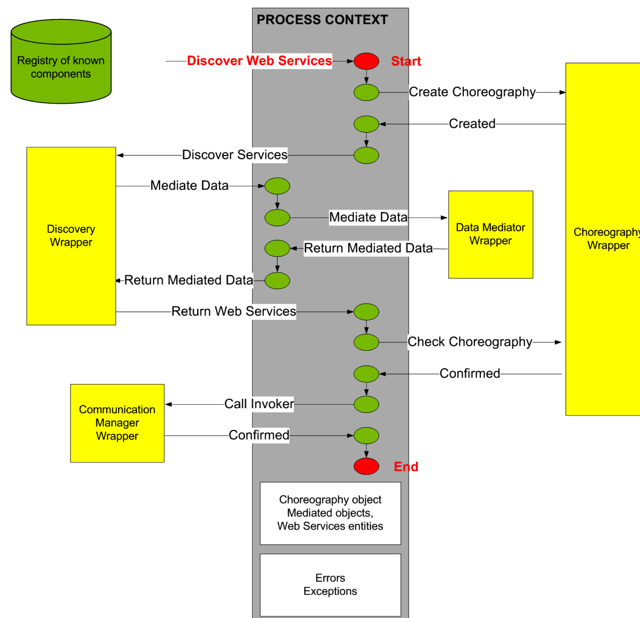

Figure 6: Process, its wrappers and context

Figure 7 depicts the complete architecture enabling decoupling of components from execution semantics in Semantic Web Services architecture. Deployment of any new execution semantics will remein transparent to components. Based on execution semantics definition, these wrappers will be only capable to consume and produce particular types of events. In a running system dynamic execution semantics is achieved by mapping abstract system behavior into real event infrastructure of the system.

To assure that there are no design flaws, livelocks or deadlocks the formal verification of the model must take place before model is deployed to the system.

5 Summary

In this document we described the architecture of the current version of WSMX. In particular we focussed on how WSMX has evolved as a Service Orineted Architecture (SOA) from earlier versions. The adoption of the principles of component decoupling and standardized component interface definitions provide WSMX with flexibility as the implementations of functional components mature and components are removed or added.

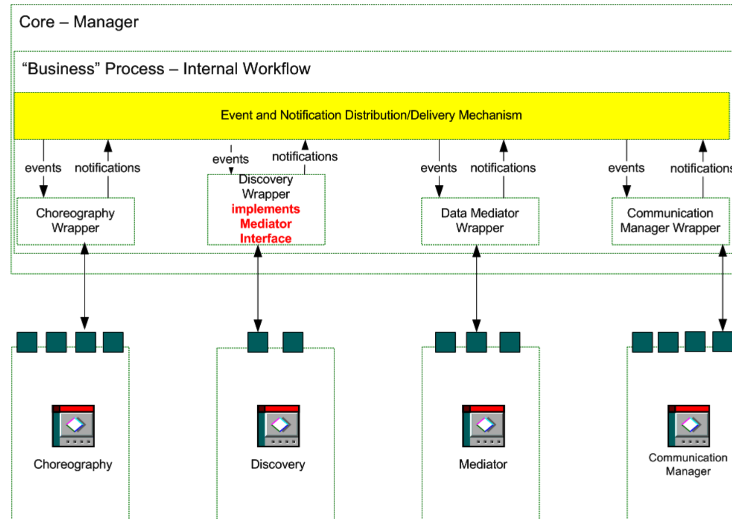

Figure 7: Event SOA for WSMX

The document described the structural view of the current WSMX components, listing and describing their publicly observable interfaces. In the section on dynamic execution semantics we discussed the benefits of separating the functional and behavioural description of a system. By adopting an architecture where the execution semantics can be updated or changed on-the-fly, we are aiming at providing as flexible a system as possible as well as attempting to future-proof the design. We can not predict all possible components and combinations of components that could be useful in a WSMX deployment but we aim to support the plug-in and out of such components as they become available and are required.

References

[1] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[2] Tom Bellwood, Luc Clment, and Claus von Riegen. UDDI Specification Version 3.0.1. UDDI Spec Technical Committee, October 2003. Available from `http://uddi.org/pubs/uddi-v3.0.1-20031014.htm`,.

[3] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreaum, Sanjiva Weerawarana, and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 1.2. World Wide Web Consortium, March 2003. Available from `http://www.w3.org/TR/2004/WD-wsdl20-20040326/`,.

[4] D. Fensel and C. Bussler. The Web Services Modeling Framework (WSMF). *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[5] Mick Kerrigan. D9.1v0.2 The Web Services Modelling Toolkit (WSMT). Technical report, 2005. `http://www.wsmo.org/TR/d9/d9.1/v0.2/`.

[6] Nilo Mitra (Ed.). SOAP Version 1.2 Part 0: Primer. World Wide Web Consortium, June 2003. Available from `http://www.w3.org/TR/soap12-part0/`,.

[7] OASIS/ebXML Registry Technical Committee. OASIS/ebXML Registry Services Specification v2.0. Technical report, 2002. `http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrs.pdf`.

[8] D. Roman, H. Lausen, and U. Keller. Web Service Modeling Ontology Standard. WSMO Working Draft v02, 2004.

[9] Laurentiu Vasiliu, Matthew Moran, Christoph Bussler, and Dumitru Roman. WSMO in DIP. Technical report, June 2004. Available at `http://www.wsmo.org/2004/d19/d19.1/v0.1/20040621/`.

[10] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–26, September 1990.

[11] Maciej Zaremba and Eyal Oren. D13.2v0.2 WSMX Execution Semantics. Technical report, Feb 2005. `http://www.wsmo.org/TR/d13/d13.2/v0.2/`.

6 Acknowledgement

The work is funded by the European Commission under the projects DIP, Knowledge Web, InfraWebs, SEKT, SWWS, ASG and Esperonto; by Science Foundation Ireland under the DERI-Lion project; by the Vienna city government under the CoOperate programme and by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects RW^2 and TSC.

The editors would like to thank to all the members of the WSMO, WSML, and WSMX working groups for their advice and input into this document.

7 Appendix A. Changelog

2005.06.13

- Refactored the document structure
- Updated the interface descriptions in the structural view to be up to date with current WSMX release.
- Added summary

2005.04.12

- Refactored the document structure
- Initial draft of section on architecture overview and methodology
- Revised section on structural view

2005.04.11

- Further refinements of structure
- Further refinements of dynamic execution semantics

2005.03.07

- Added first cut for dynamic execution semantics

2005.01

- Document created

8 Appendix B. Component Interface Tables

Table 3: Goal Resource Manager Interfacse

Method Summary	
void	saveGoal(Goal goal) Save a goal to the repository
void	removeGoal(Goal goal) Remove a goal from the repository
Set<Identifier>	getGoalIdentifiers() Get all goal identifiers from the repository
Set<Identifier>	getGoalIdentifiers(Set<Object>searchTerms, boolean conjunctive) Gets a set of goal identifiers from the repository that match the searchterms joined using the conjunctive in the input parameters
Set<Identifier>	getGoalIdentifiers(Namespace namespace) Gets the set of goal identifiers corresponding to a particular namespace from the repository
boolean	containsGoal(Identifier identifier) Determine if the repository contains the goal corresponding to the identifier
Goal	loadGoal(Identifier identifier) Returns a goal object corresponding to the identifier from the repository
Set<Goal>	loadAllGoals() Returns a set of all goal objects from the repository

Table 4: Mediator Resource Manager Interface

Method Summary	
void	saveMediator(Mediator mediator) Save a mediator to the repository
void	removeMediator(Mediator mediator) Remove a mediator from the repository
Set<Identifier>	getMediatorIdentifiers() Get all mediator identifiers from the repository
Set<Identifier>	getMediatorIdentifiers(Set<Object>searchTerms, boolean conjunctive) Gets a set of mediator identifiers from the repository that match the searchterms joined using the conjunctive in the input parameters
Set<Identifier>	getMediatorIdentifiers(Namespace namespace) Gets the set of mediator identifiers corresponding to a particular namespace from the repository
boolean	containsMediator(Identifier identifier) Determine if the repository contains the mediator corresponding to the identifier
Mediator	loadMediator(Identifier identifier) Returns a mediator object corresponding to the identifier from the repository
Set<Mediator>	loadAllMediators() Returns a set of all mediator objects from the repository

Table 5: Web Service Resource Manager Interface

Method Summary	
void	saveWebService(WebService webService) Save a WebService to the repository
void	removeWebService(WebService webService) Remove a WebService from the repository
Set<Identifier>	getWebServiceIdentifiers() Get all WebService identifiers from the repository
Set<Identifier>	getWebServiceIdentifiers(Set<Object>searchTerms, boolean conjunctive) Gets a set of WebService identifiers from the repository that match the searchterms joined using the conjunctive in the input parameters
Set<Identifier>	getWebServiceIdentifiers(Namespace namespace) Gets the set of WebService identifiers corresponding to a particular namespace from the repository
boolean	containsWebService(Identifier identifier) Determine if the repository contains the WebService corresponding to the identifier
WebService	loadWebService(Identifier identifier) Returns a WebService object corresponding to the identifier from the repository
Set<WebService>	loadAllWebServices() Returns a set of all WebService objects from the repository

Table 6: Ontology Resource Manager Interface

Method Summary	
void	saveOntology(Ontology ontology) Save a Ontology to the repository
void	removeOntology(Ontology ontology) Remove an Ontology from the repository
Set<Identifier>	getOntologyIdentifiers() Get all Ontology identifiers from the repository
Set<Identifier>	getOntologyIdentifiers(Set<Object>searchTerms, boolean conjunctive) Gets a set of Ontology identifiers from the repository that match the searchterms joined using the conjunctive in the input parameters
Set<Identifier>	getOntologyIdentifiers(Namespace namespace) Gets the set of Ontology identifiers corresponding to a particular namespace from the repository
boolean	containsOntology(Identifier identifier) Determine if the repository contains the Ontology corresponding to the identifier
Ontology	loadOntology(Identifier identifier) Returns a Ontology object corresponding to the identifier from the repository
Set<Ontology>	loadAllOntologys() Returns a set of all Ontology objects from the repository

Table 7: Non WMSMO Resource Manager Interface

Method Summary	
void	saveMessage(Context Context, MessageId messageId, String message) Save a message received or being sent from WSMX. The messageId is generated by the execution semantics.
Set<Context >	getContexts() Get all contexts in the repository. Contexts are used to ideentify instances of execution semantics.
Set<MessageId>	getMessageIds(Context context) Get all message identifiers for a specific context.
Map <Context, Set <MessageId>>	getMessageIds(Set <Object>searchTerms, boolean conjunctive) Gets a map of contexts to sets of message ids where the map matches the searchterms joined using the conjunctive in the input parameters
Set <MessageID>	getMessageIds(Context context, Set<Object>searchTerms, boolean conjunctive) Gets a set of MessageIds for messages in the specified context where the set matches the searchterms joined using the conjunctive in the input parameters
Map<MessageId, String>	load(Context context) Gets a map of messageIds to the associated messages (as strings) corresponding to the specified context
String	load(Context context, MessageId messageId) Gets the string content of the message corresponding to the specified context and the messageId
Map<Context, Map<MessageId, String>>	loadAll() Get a map conatining all messageIds and the corresponding message strings for the specified context
void	registerWSDL(Context context, Goal goal, WSDLDocument wsdlDocument) Register the WSDL document for the specified goal and context
WSDLDocument	getWSDL(Context context, Goal goal) Get the WSDL document corresponding to the specified goal and context

Table 8: WSDL Resource Manager Interface

Method Summary	
void	registerWSDL(WebService webService, WSDLDocument wsdlDocument) Register a WebService with an associated WSDL document
WSDLDocument	getWSDL(WebService webService) Get the WSDL document for the specified Webservice
void	deregisterWSDL(WebService webService) Dereigter the WSDL document from the specified WebService

Table 9: Service Discovery Interface

Method Summary	
List <WebService>	discover(Goal goal) Calls discovery providing a Goal and expects result as a (possibly empty) list of WebServices

Table 10: Selection Interface

Method Summary	
WebService	select(Collection<WebService> services, Preferences preferences) Selects Web Service based on Preferences

Table 11: Negotiation Interface

Method Summary	
List<WebService>	select(Collection<WebService> services, Preferences preferences) Negottiate with Web Services to find Web Services that can deliver a specific product or service

Table 12: Data Mediator Interface

Method Summary	
Map<Identifiable, List<Identifiable>>	mediateData(Ontology sourceOntology, Ontology targetOntology, Set<Identifiable> data) Calls the mediator for a target and source ontology and a set of identifiables representing data instances that need to be mediated. The method returns the mediated data as a map between the initial (source) instances and the mediated data (the instances in terms of target ontology).
List<Identifiable>	mediate(Ontology sourceOntology, Ontology targetOntology, Identifiable data) Transforms a give source ontology instance into instances of the target ontology.

Table 13: Process Mediator Interface

Method Summary	
Map<Identifier>, List<Identifiable>>	generate(Identifier sourceOntology, Identifier targetOntology, Set<Identifiable> data) Generates a list of identifiable of the concepts/instances to be sent to either of the two involved parties and returns the identifier of the choreography instance of the partner that needs to receive some messages, together with the list of identifiable objects containing the actual information; each of the two partners may receive one or more messages
Map<Identifier>, List<Identifiable>>	generate(Ontology sourceOntology, Ontology targetOntology, Set<Identifiable> data) Generates a list of identifiable of the concepts/instances to be sent to either of the two involved parties and returns the identifier of the choreography instance of the partner that needs to receive some messages, together with the list of identifiable objects containing the actual information; each of the two partners may receive one or more messages
Map<Identifier>, List<Identifiable>>	generate(Choreography sourceChoreography, Choreography targetChoreography, Set<Identifiable> data) Generates a list of identifiable of the concepts/instances to be sent to either of the two involved parties and returns the identifier of the choreography instance of the partner that needs to receive some messages, together with the list of identifiable objects containing the actual information; each of the two partners may receive one or more messages

Table 14: Invoker Interface

Method Summary	
void	invoke(WebService service, List<Identifiable> data, Axiom operation) Invoke the Web service using the list of identifiables as objects and the specified operation as the WSDL operation to be used.

Table 15: Receiver Interface

Method Summary	
Context	receive(WSMLDocument wsmlMessage, Context context) Receive a WSML message corresponding to a particular converation context if one exists. If this is the first message in a conversation, the context is created by WSMX and returned to the WSMX client. Otherwise the value of the returned context is the same as the value of the context passed as an input parameter.

Table 16: Choreography Engine Interface

Method Summary	
void	registerChoreography(Goal goal) When a goal is received, the choreography instance of the requestor needs to be registered
void	registerChoreography(WebService webService) When a WS that satisfy a certain goal is discovered, an instance of its choreography needs to be registered.
void	updateState(URI origin, Identifiable message) Attempt to update the internal state of the choreography engine and determine if the received message from a given origin results in a valid next state. If it doesn't, this method throws a subclass of ComponentExcetion that carries more information why the resulting state is not a valid one ie if we expect to get creditcard data from the client next but receive location information instead.

Table 17: Choreography Engine Interface

Method Summary	
Set<Identifiable>	parse(WSMLDocument wsmlDocument) Parse the received WSML into corresponding WSMO4j objects.