



# D13.4v0.1 WSMX Architecture

WSMO Working Draft 26 April 2004

**This version:**

<http://www.wsmo.org/2004/d13/d13.4/v0.1/20040426/>

**Latest version:**

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

**Previous version:**

<http://www.wsmo.org/2004/d13/d13.4/v0.1/20040404/>

**Editor:**

Michal Zaremba

**Authors:**

Matthew Moran

Michal Zaremba

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](#)
  - 1.1 [Overview](#)
  - 1.2 [Purpose of this Document](#)
  - 1.3 [Scope of WSMX Architecture](#)
  - 1.4 [Documente Overview](#)
2. [Black Box Overview](#)
  - 2.1 [Overview](#)
  - 2.2 [Compilation Functionality](#)
  - 2.3 [Excecution Functionality](#)
3. [Architecture Components](#)
  - 3.1 [Introduction](#)

## [3.2 Detailed Architecture](#)

### [3.2.1 Components Dedicated to Compilation of Integration Definition Types](#)

### [3.2.2 Components Dedicated to Execution of WSMX Messages](#)

## [3.3 Components](#)

### [3.3.1 User Interface](#)

### [3.3.2 WSMX Management](#)

### [3.3.3 Compilation Engine](#)

### [3.3.4 Ontology Repository](#)

### [3.3.5 WSMX WSDL](#)

### [3.3.6 Events Management](#)

### [3.3.7 Event Transformer](#)

### [3.3.8 Discovery](#)

### [3.3.9 Selection](#)

### [3.3.10 Mediation](#)

### [3.3.11 Invoker](#)

## [5 . Conclusions and Future Work](#)

### [References](#)

### [Acknowledgment](#)

# 1. Introduction

## 1.1 Overview

The Web Services Modelling Execution Environment (WSMX) is an execution environment for dynamic discovery, selection, mediation and invocation of web services. WSMX is based on the Web Services Modelling Ontology (WSMO) [Roman et al., 2004] which describes all aspects related to this discovery, mediation, selection and invocation. The execution operations of WSMX are made possible by describing elements relating to web services using the Web Service Modelling Language (WSML) [Oren et al., 2004] and then compiling these descriptions into WSMX.

## 1.2 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, then, goes on to provide details of the how these components are made up, the interfaces each component provides and the flow of information through these components.

## 1.3 Scope of WSMX Architecture

There are two aspects to the WSMX Architecture - compilation and execution. This document

describes both in terms of component behaviour and the interfaces provided by each component. Building on the other deliverables of the WSMX working group, this document provides the specification for the implementation of the WSMX software.

## 1.4 Document Overview

Section 2 provides a black box overview of the WSMX functionality. Section 3 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.

# 2. Black Box Overview

## 2.1 Overview

The purpose of this section is to present the WSMX architecture from an external, high level perspective (as a black box, which accepts some inputs and provides some outputs) and to give an overview of what exactly WSMX 'can do'. This section defines how external entities such as intra-enterprise back-end applications or inter-enterprise information systems can interact with WSMX platform.

There are two main objectives of the WSMX architecture:

- Compilation of the integration definition types described in WSML
- Execution of WSMX Messages. A WSMX Message consists of a goal a user wishes to achieve along with the data required for this goal (payload)

The WSMX architecture is built around these two objectives.

Figure 1 below, provides a simplified overview of the WSMX architecture. It includes the functional components of compiler, invoker, persistent storage, ontologies, and history component. Each of these components has its own interface defined in this document. Additionally two elements of this architecture are designed as web services - WSMX itself as the whole and the mediator component (any new mediator can be plugged as Web Service into WSMX). This document defines WSDL interfaces for these two Web Services as well.

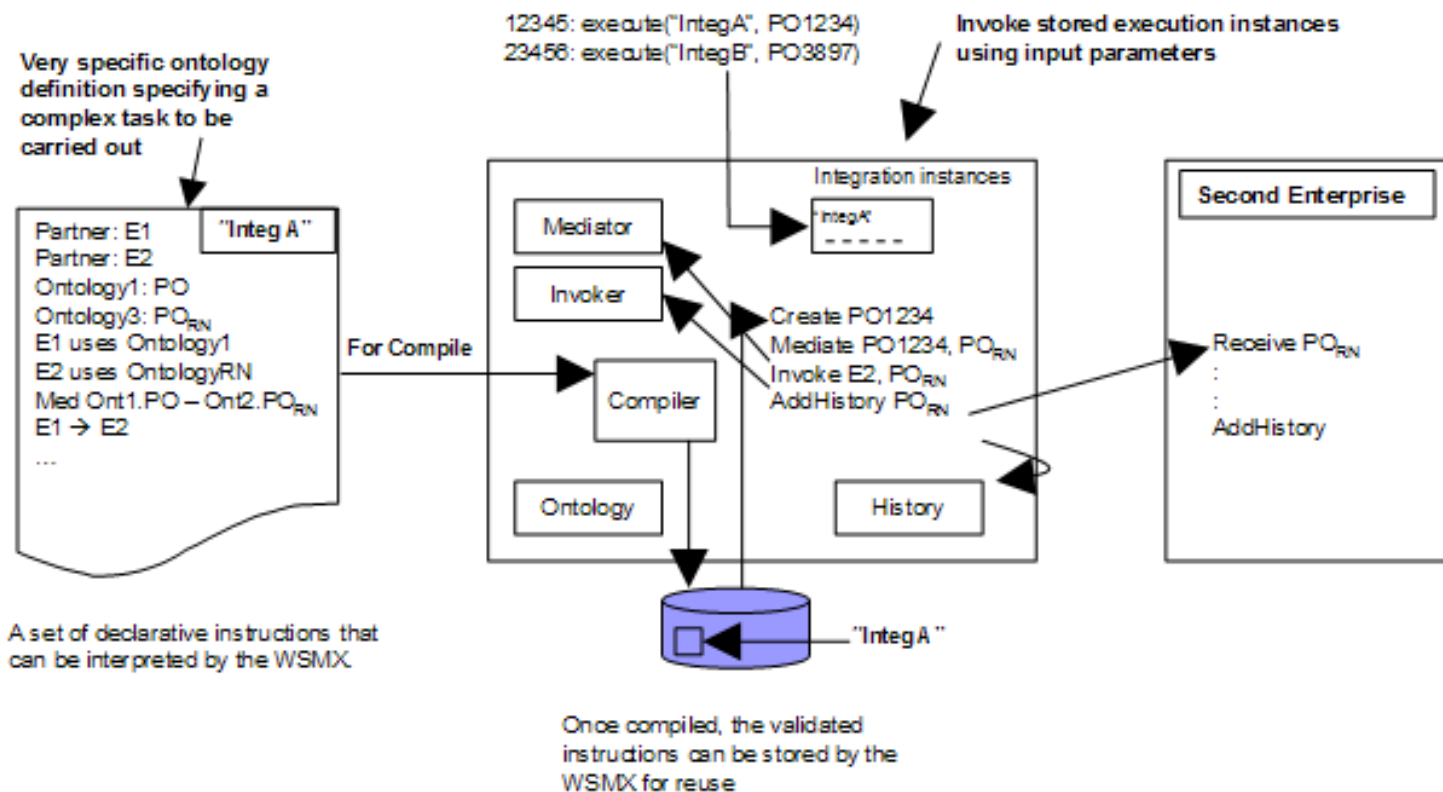


Figure 1: Informal WSMX Architecture - high level overview

## 2.2. Compilation Functionality

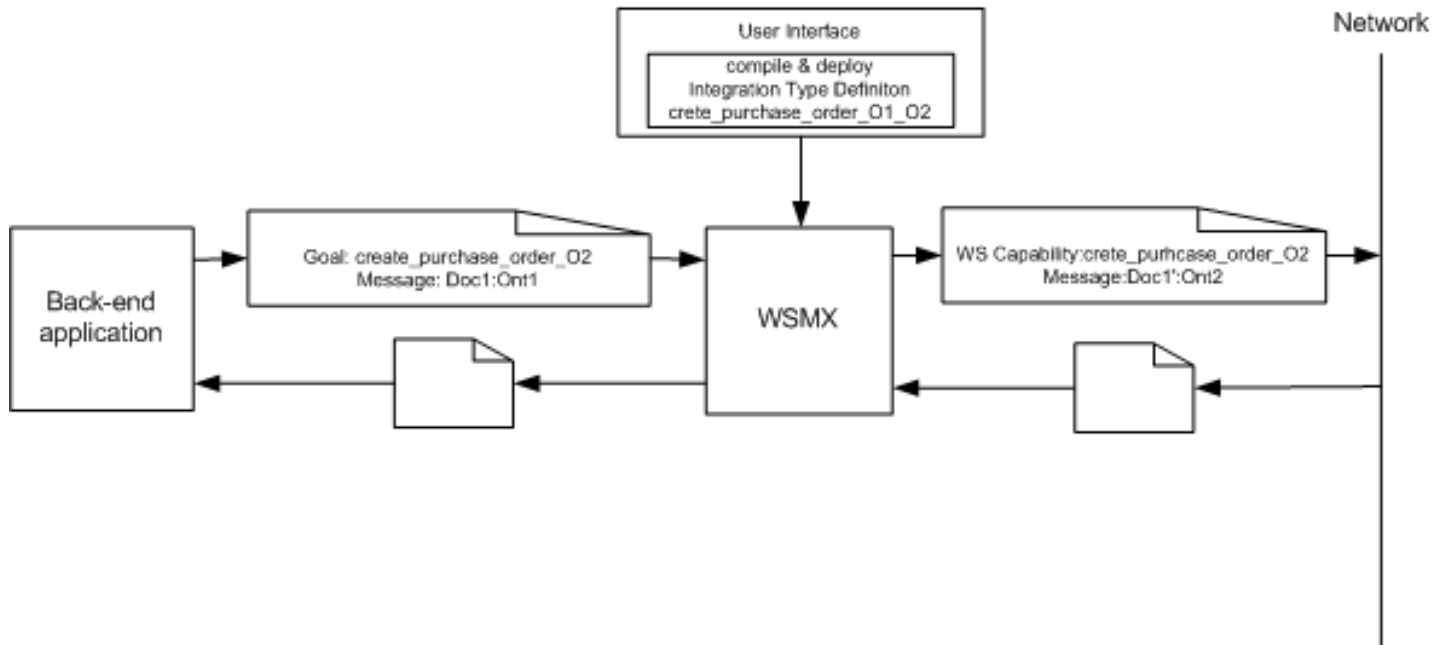
A design environment such as the WSMO Editor [Lausen et al., 2004] is used to generate the WSMO descriptions of elements required for Semantic Web Services. These descriptions, called Integration Types, are created in WSMO and sent to WSMX for compilation. An example of an Integration Type is the WSMO description of a web service. Once an Integration Type has been compiled to WSMX, it is available for use in the execution of user-specified goals. The design environment send Integration Types to WSMX for compilation using a public interface defined in WSDL.

## 2.2. Execution Functionality

WSMX accepts WSMX Messages requesting the execution of a user-specified goal. WSMX Messages consist of a goal along with whatever payload data is required to carry out the task corresponding to that goal. External systems send messages to WSMX using a public interface defined with WSDL. Once a message has been received, WSMX discovers appropriate web services, selects one of them, carries out any required mediation and finally invokes the web service.

The high level, external overview example of the compilation and the execution of integration type and instances definitions with WSMX platform is presented in figure 2. In this example during compilation phase the user defines a new integration definition type and sends it to WSMX compiler using the user interface. During execution time, the back-end application sends

a message to WSMX with the goal to make a specific purchase. The payload of the message contains a purchase order in the format of ontology O1 (for example in RosettaNet [RosettaNet] format). WSMX takes care of mediating the given document and finding the appropriate Web Service, capable of fulfilling the requested goal (accept purchase order in ontology O2 format). WSMX forwards the mediated purchase order to the right endpoint. There are many phases in the whole process where exceptions and errors could happen, but WSMX takes care for the whole process of error handling.

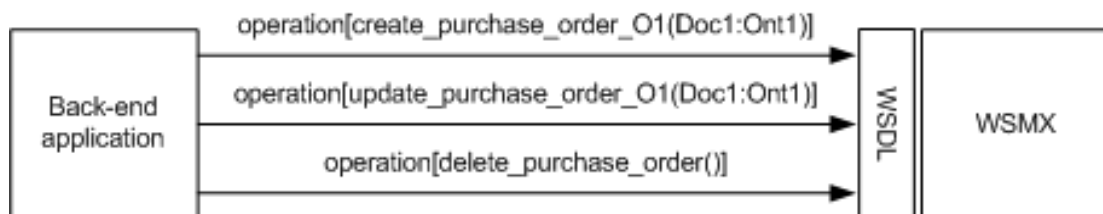


**Figure 2: High external view of the WSMX platform**

WSMX does not provide any predefined integration type definitions, so any new definition can be written and compiled. WSMX platform is able to handle instances of any correctly defined and compiled type. For example three of the RosettaNet PIPs: 3A4 (Request Purchase Order), 3A8 (Request Purchase Order Change) and 3A9 (Request Purchase Order Cancellation) could be defined as four integration definition types:

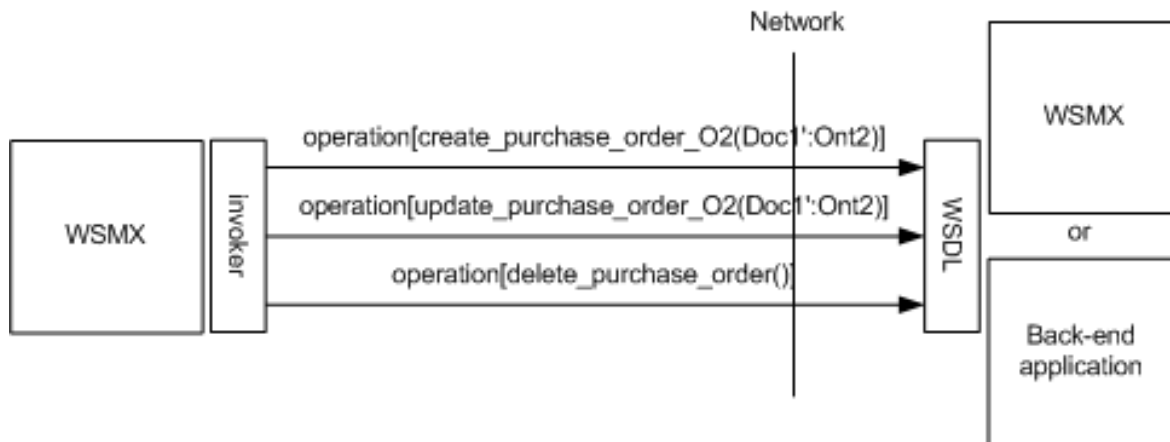
- create\_purchase\_order\_O1\_O2,
- acknowledge\_purchase\_order\_O2\_O1,
- update\_purchase\_order\_O1\_O2,
- delete\_purchase\_order.

Figure 3 shows how the back-end application can invoke operations, which handle three of the given integration type instances.



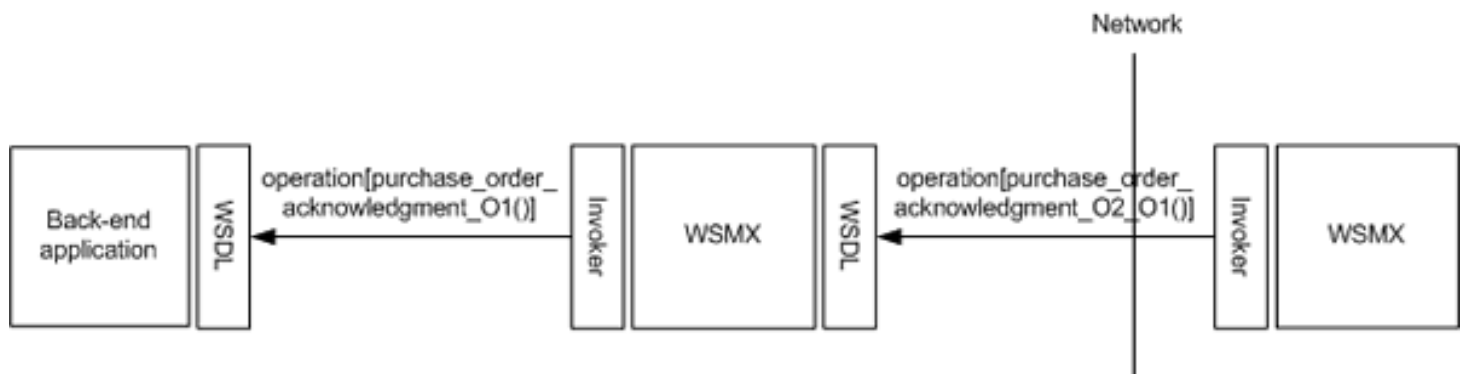
**Figure 3: An example of the invocation between back-end application and WSMX platform**

An integration definition type specifies which ontologies should be used and identify Web Services, which are able to handle operations (e.g. that Web Service X and Y are capable to sell books and receive purchase orders in RosettaNet format). WSMX takes care to mediate the business message (e.g. purchase order) to the appropriate ontology and forwards it to the right endpoint using one of the Web Service, which has capabilities matched with the goal of the requester. (see figure 4).



**Figure 4: Communication between two WSMX platforms**

From the perspective of usability of such a system, it would be desirable that WSMX platform should not differentiate between calls coming from the back-end application systems (intra-company information systems) and from the network. We do not consider at this stage security concerns related to such a design, but we only focus on the simplification of the WSMX architecture. Figure 5 presents how instance of the purchase\_order\_acknowledgment\_O2\_O1 type could come from the network from the other business partner information system (in this case another WSMX platform). WSMX platform takes care for the mediation and forwarding of a new mediated message to the back-end application system.

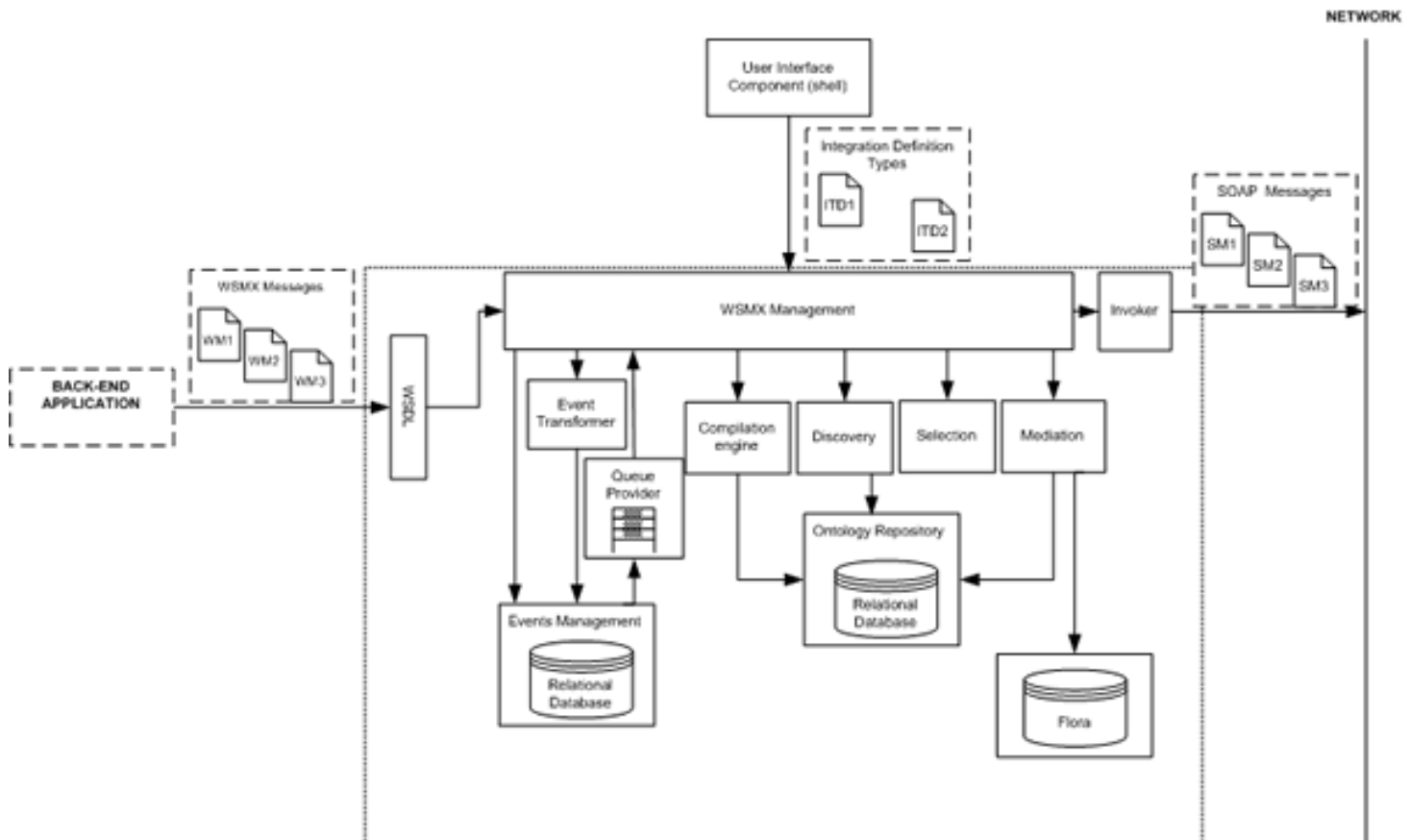


**Figure 5: Invocation of WSMX from another enterprise back-end application system**

## 3. Architecture Components and Interfaces

### 3.1 Introduction

WSMX architecture consists of set of components (see figure 6). Each of these elements has its own public interface, which can be either accessed by other components or by other systems. This section aims to explain the functionality and the purpose of each of the WSMX architecture components.



**Figure 6: WSMX Architecture**

There are two types of components in the WSMX architecture - components dedicated to compilation of integration definition types and components dedicated to execution of WSMX messages/events. Two of the components are specifically dedicated to compilation of the integration definition types and eight of them are dedicated to execution of the WSMX messages/events. WSMX Management and Ontology Repository components are shared among compilation and execution functions of the WSMX platform. Components dedicated to compilation of integration definition types are: User Interface and Compilation Engine. Components dedicated to execution of the WSMX messages/events are: Events Management, Event Transformer, Discovery, Mediation and Selection engines and Invoker.

## 3.2. Detailed Architecture

### Semantics Used for Detailed Architecture Models

The architecture is modelled using slightly modified UML Activity Diagrams with the following semantics. Each oval represents an action. Each edge (or arc) joining two actions represents an interface. The direction of the arrowhead specifies the flow direction for data between two actions. An expression enclosed in square brackets on an edge denotes a guard condition that

must be fulfilled before the edge may be used. A solid dot indicates the starting point for the model. A solid dot enclosed in a circle indicates an end point. Edges are labelled with the data types that make up the interface represented by the edge. Actions are labelled with the action that they carry out. In the model actions are grouped together in boxes or “swim lanes”. Swim lanes group together operations belonging in the same architectural component. The name of the corresponding component is provided at the top of each swim lane. Additionally we introduced datastore representation, which is not defined in UML Activity diagram definition.

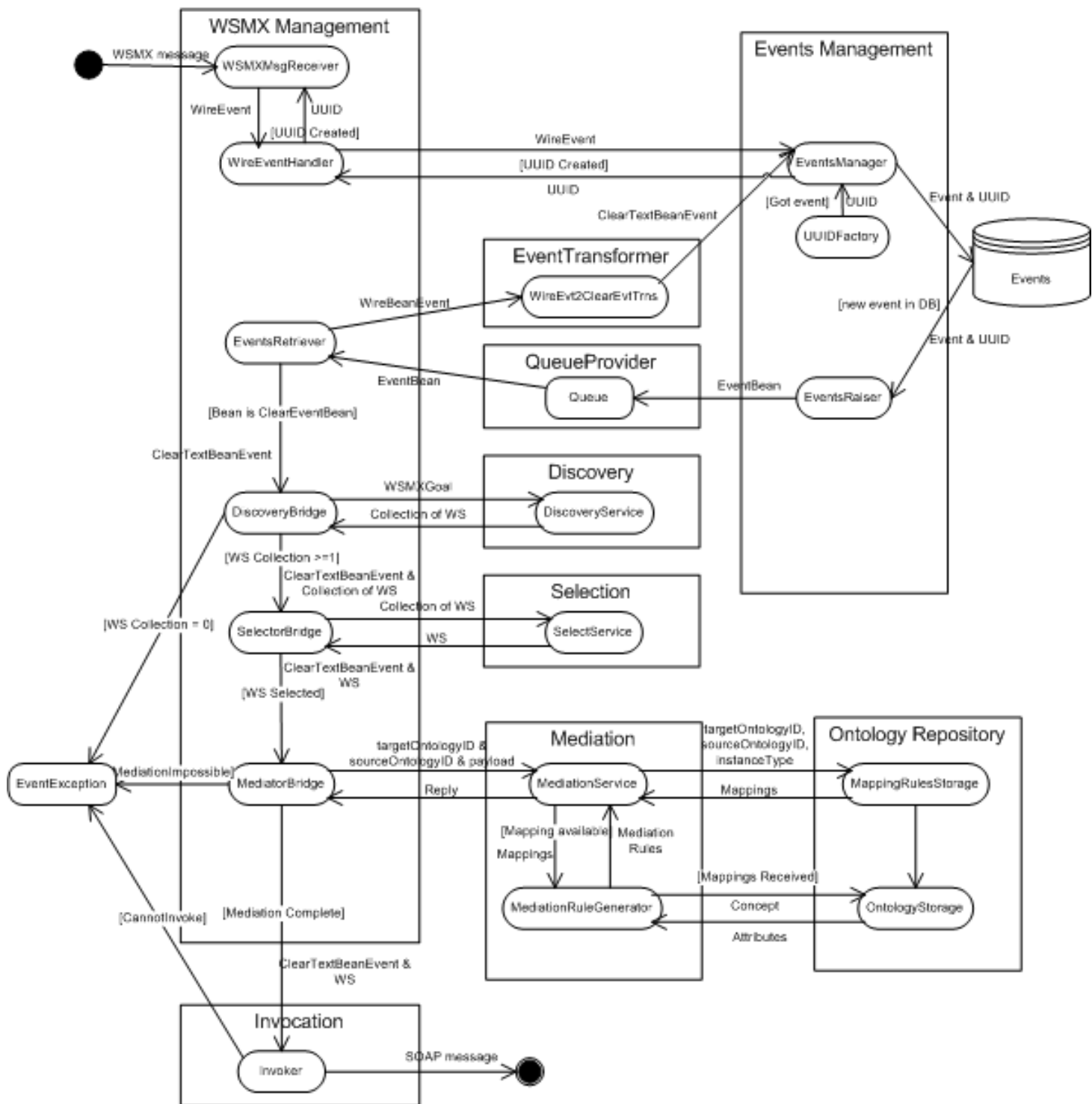
### **Intention of the Detailed Models**

The detailed models are built on the WSMX Architecture diagram provided earlier. They show operations in each component and specify the interfaces for these operations. The model shows the sequence of operations within the architecture required for both the execution functionality and the compilation functionality.

#### **3.2.1. Components Dedicated to Compilation of Integration Definition Types**

#### **3.2.2. Components Dedicated to Execution of WSMX Messages**

The detailed architecture presenting components involved during execution after WSMX message gets to the system is presented on figure 7.



**Figure 7: WSMX Architecture - WSMX message execution (click to enlarge)**

To provide the reliability, each event and its state is recorder permanently in the datastore. Any new event in the system is assigned a universally unique identifier - UUID, which is a 128-bit value. Event state changes are recorded in the database when discovery, selection, mediation and invocation take place but we do not show this in the diagram as it would add clutter. Various event states are part of the execution semantic, but for convenience, they have been presented in table 1.

WSMX events states	
<code>created</code>	Any new event recorded in the database is getting first the status <code>created</code> . The event is assigned a unique UUID, which can be used later on to figure out the status of the event, as well as to find related events. From the implementation perspective, status <code>created</code> will usually not last longer than a couple of milliseconds till the event is scheduled for processing and insert in the queue.
<code>raised</code>	Once event is scheduled for processing, it is assigned event status <code>raised</code> . There are various status codes assigned to this state designed as enumeration types. For example event with the status <code>raised</code> and status code <code>RaisedEventStatusCode.STATUS_IN_QUEUE=1</code> indicates that event was given to Queue Provider for processing; event with the status <code>raised</code> and status code <code>RaisedEventStatusCode.STATUS_ENTER_SELECTION=3</code> indicates event scheduled for processing to Selection Bridge (codes of the events will be documented in detail in the next versions of this document).
<code>consumed</code>	The lifetime of <code>consumed</code> event has finished. It remains in the main datastore as long as all the related events are still processed and their status remain in any two of the states discussed so far. Once the whole process consisting of the set of events concludes, all <code>consumed</code> events are getting archived.
<code>error</code>	Errors and exception might happen in any of the components during execution of messages/events. WSMX platform must be able to handle them and to record any error states. From some of the states WSMX platform should be able to recover (e.g. Selection Component Temporary Unavailable), from some it should stop further processing after recording an error message (e.g. Mediation not possible, because ontology is not known). Errors are classified by errors codes (detailed error codes and their semantic will be defined in the next version of this document). For example error code <code>ErrorEventStatusCode.STATUS_QUEUE_PROVIDER_UNAVAILABLE=1</code> means that Queue Provider is unavailable. WSMX platform should record an <code>error</code> state for the event with the given code, and attempt to access again Queue Provider after some period of time.

**Table 1: WSMX events states**

EventStateBean, a part of EventBean, is characterised by a stateName and stateCode.

## 3.3. Components

### 3.3.1 User Interface

User interface component is used to compile new integration definition types for WSMX platform. In the future it will be also a bridge for graphical environment used to create integration definition types.

### 3.3.2 WSMX Management

WSMX management is a business logic component of the WSMX platform.

### Compilation Related Functionality

### Execution Related Functionality

WSMX management component is running as two separate processes. One part of the Management component is a **message listener** running inside of J2EE container and waiting for new messages incoming from the network. Each new message is handled to Wire Event Handler, which takes care to communicate with Events Management Component to register new events in WSMX datastore. The second part of Management component is running as a **system process** which regularly collects any new EventsBeans from Queue Provider and forwards them to appropriate components.

### 3.3.3 Compilation Engine

Compilation engine checks correctness of the syntax of integration definitions types, decomposes them and triggers operations of the Ontology Repository component to get them stored.

### 3.3.4 Ontology Repository

Ontology repository component is a persistent storage for integration definition types and for mediated instances of messages (it will be extended in the future to support selection and discovery components).

### Compilation Related Functionality

### Execution Related Functionality

During execution time, mediation component uses ontology repository to retrieve Mappings and Attributes to carry mediation.

### 3.3.5 WSMX WSDL

WSDL interface in WSMX is not a stand-alone component (as it does not provide any 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 aim to instantiate *integrations type definitions* compiled into WSMX platform. There is only one external function/operation provided by WSDL interface to other information systems in WSMX platform called `invokeOperation` (see table 1).

Method Summary
----------------

WSMXMessage	<p><b><code>invokeOperation</code></b>(<code>WSMXMessage</code> message)()</p> <p>This operation takes as an input an instance of WSMX message, and returns back another instance of WSMX message. In most of the cases input WSMX messages would carry payload (e.g. purchase orders), while output message would encapsulates unique identifiers UUID, which would be assigned to new, instantiated events in the WSMX platform.</p>
-------------	--

**Table 2: WSMX WSDL interface method definition**

Invoke operation takes as input an instance of WSMX message, and returns back another instance of WSMX message. While input messages would usually carry extensive information about the requirements of the invoking application such as for example goals, payloads, signatures and various other elements, output messages would, in most of the cases, forward only the unique identifiers generated by *Events Management* component. We consider to represent WSMX messages as subclass of an ebXML message, which has been standardized in ebXML Messaging specification [ebMS, 2002]. It would require providing of the implementation of ebXML message on top of SOAP message, and next extending it to WSMX message to encapsulate properties not handled by ebXML messages (e.g. Goals).

`java.lang.Object`

└ `javax.xml.soap.SOAPMessage`

└ `com.sun.xml.messaging.jaxm.ebxml.EbXMLMessage`

└ `ie.deri.wsmx.message.WSMXMessage`

ebXML messaging specification designers recognized most of the requirements for electronic business messaging allowing additionally flexible including new elements in their messages. At this stage standard ebXML message header elements are for example `id`, `From`, `To`, `ConversationId`, etc., but WSMX requires also additional object to be included in the WSMX message header. An example of such an additional element handled by ebXML message header could be a Goal element, which would specify objectives that a client application can has when it consults a Web Service. ebXML messages allows carrying any payload in any format as MIME attachments.

### 3.3.6 Events Management

Events Management component stores wire events and clear text events. An externally operations supported by this component is presented in table 3.

Method Summary	
Response	<p><b><code>saveWireEvent</code></b>(<code>WSMXMessage</code> message)()</p> <p>Saves wire event</p>
Response	<p><b><code>saveClearTextEvent</code></b>(<code>Payload</code> message, <code>Goal</code> goal, <code>String</code> from)()</p> <p>Saves clear text event</p>

Response	<code>saveClearTextEvent(Payload message, Goal goal, String from, Signature signature)()</code> Saves clear text event
----------	---

**Table 3: Events management component interface methods definition**

Four types of message related objects can be handled by WSMX platform (three of them can be instantiated):

- **WSMXMessage** is a wire based message. To provide system reliability, any message, which is coming from the network must be first recorded in the datastore, before any processing can happened;
- **EventBean** is an abstract class which encapsulate basic information about **WireEventBean** or **ClearTextEventBean**. EventBean is only a super class, which cannot be instantiated, but two its subclasses can;

```
ie.deriv.wsmx.message.EventBean
```

```
└ ie.deriv.wsmx.message.WireEventBean
```

```
└ ie.deriv.wsmx.message.ClearTextEventBean
```

- **WireEventBean** - object representation of wire events;
- **ClearTextEventBean** - object representation of clear text events.

### 3.3.7 Event Transformer

Event transformer extracts pieces of information from the wire event header and body and saves them as a clear text event.

### 3.3.8 Discovery

Discovery component provides the collection of Web Services, which are capable to meet goals - discovery match goals with the capabilities.

### 3.3.9 Selection

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 wish.

### 3.3.10 Mediation

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	
Reply	<pre><b>mediate</b>(String sourceOntologyID, String targetOntologyID, Payload message)()</pre> <p>This operation takes as an input identifiers of ontologies and a payload message, and returns back mediated Payload message</p>

**Table 4: Mediation Engine interface**

Mediation component makes a decision based on target and source ontologies IDs, and the payload of the message if the payload should be mediated. Reply object will include either mediated payload, information that mediation is not necessary or error code if mediation is impossible.

### 3.3.11 Invocation

Invocation component invokes Web Services in other information systems. Invoker extracts elements from EventBean, which are next packed into SOAP message, which is handled to other systems.

## 4. Conclusions and Future Work

Future work will include even more detailed definition of the interfaces and more detail functionality description for some of the components (as selection or discovery engines).

## References

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

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

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

**[Oren et al., 2004]** Oren et al., BNF grammar for WSML user language, <http://www.wsmo.org/2004/d16/d16.1/v0.2/20040418>

**[Roman et al., 2004]** Roman et al., Web Service Modeling Ontology - Standard (WSMO - Standard), <http://www.wsmo.org/2004/d2/v02/20040306/>

## 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