



D13.5v0.1 WSMX Implementation

WSMO Working Draft 19 July 2004

This version:

<http://www.wsmo.org/2004/d13/d13.5/v0.1/20040719/>

Latest version:

<http://www.wsmo.org/2004/d13/d13.5/v0.1/>

Previous version:

<http://www.wsmo.org/2004/d13/d13.5/v0.1/20040315/>

Editors:

Matthew Moran

Authors:

Michal Zaremba

Matthew Moran

Emilia Cimpian

Adrian Mocan

Eyal Oren

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

Table of contents

1 Introduction

2 Overview of WSMX Architecture

3 Overview of Implementation

[3.1 WSMX Web Service Interfaces](#)

[3.2 Event Management in WSMX](#)

4 Datamodel

[4.1 Ontology](#)

[4.2 Data Mediation](#)

[4.3 Web Service](#)

[4.4 Goal](#)

[4.5 Events](#)

[4.6 Messages](#)

5 Components

[5.1 WSMX Manager](#)

[5.1.1 WSMX Manager Listener](#)

[5.1.2 WSMX Manager Core](#)

[5.1.3 Message Scanner](#)

[5.1.4 Events Scanner](#)

[5.2 Event Listeners](#)

[5.2.1 Parser Listener](#)

[5.2.2 MatchMaker Listener](#)

[5.2.3 Selector Listener](#)

[5.2.4 Mediator Listener](#)

[5.2.5 Invoker Listener](#)

[5.3 Adapter](#)

[5.4 Compiler and Message Parser](#)

[5.5 MatchMaker](#)

[5.6 Selector](#)

[5.7 Mediator](#)

[5.7.1 XML Converter](#)

[5.8 Invoker](#)

[5.9 Resources Manager](#)

6 References

7 Acknowledgement

8 Appendix

[8.1 Javadoc Description for WSMX](#)

[8.2 Mapping Rules Editor Implementation](#)

[8.2.1 Graphical User Interface](#)

[8.2.2 Mapping Rules Creator](#)

1 Introduction

The Web Services Modelling Execution Environment, WSMX, is a software system that allows web services whose semantics have been formally described to be discovered, selected, mediated and invoked to carry out specific client tasks. Semantics, in this context, is the meaning of various aspects of web services that allow machines to automatically carry out tasks using the web services with a minimum or no human intervention. WSMX uses the conceptual model described by WSMX-O [Cimpian et al., 2004]. WSMX-O has its foundation in the Web Service Modeling Ontology (WSMO) [Roman et al., 2004a] and the Web Service Modelling Framework, WSMF, [Fensel & Bussler, 2002]. The goal of WSMX is to provide a flexible environment for application and business integration based on strongly decoupled physical components with strong mediation services enabling every party to speak with each other as advocated in WSMF.

The execution semantics of WSMX [Oren, 2004] provide a formal description of the operation of WSMX. This allows the reader to unambiguously determine the behaviour of the WSMX at any point in its operation. The execution semantics for WSMX have been described using Petri Nets which themselves have a formal semantics. This document does not duplicate the formal description of the operation of WSMX or duplicate the comments included in the actual java source code . Instead, the document provides natural language descriptions of the software components developed to implement the WSMX architecture.

This deliverable is one of a series describing the different aspects of WSMX. [Oren et al., 2004] provide an overview of WSMX. The conceptual model is described in [Cimpian et al., 2004]. Mediation in the context of WSMX is described in [Mocan & Cimpian, 2004] and the architecture for WSMX is presented in [Zaremba et al., 2004].

This deliverable describes the implementation of a minimal but complete first version of WSMX. Complete means that the core functionality developed in this first version of WSMX is sufficient to discover and select a web service matching a goal formally

described in terms of WSMX-O, to mediate between the data required by the web service and that offered by the goal, and then to actually make the web service invocation. The implementation is minimal in the sense that in the case of some components, only a simple design and implementation has been carried out to make the component operational. The focus in the first phase of WSMX has been to get components and their interfaces in place to provide a sound framework where specialised components supporting the WSMX interfaces can be plugged in without affecting the stability or integrity of the overall system.

Section 2 gives an overview of the WSMX architecture. Section 3 provides an overview of the implementation of WSMX including description of the event management system, configuration of WSMX and how WSMX itself acts as a web service. The structure of the datamodel is described in section 4 and brief descriptions of each of the implementation of each of the architectural components along with their external interfaces is given in section 5. A link to the java documentation for the current source code is available in the appendix. This documentation was generated using the javadoc tool and follows the usual conventions for java source code.

2 Overview of WSMX Architecture

Full details of the WSMX architecture are available in [Zaremba et al., 2004]. This section is intended as a brief review.

Figure 1 provides the architecture for WSMX. This version of WSMX provides a framework to accept a service requester goal expressed in WSML and to ultimately make an invocation to a web service which can satisfy that goal. The next paragraph steps through how this is achieved using WSMX.

The **WSMO editor** is used to create WSMO descriptions of web services, ontologies, mediators and goals. WSMX provides an interface to accept these descriptions and the **Compiler** component parses and saves them so that they are ready for use by WSMX. Parsing means validating the descriptions and then storing them persistently in the Ontology Registry.

The **WSMX Manager** controls the operational flow of the system and is also responsible for event management, keeping a full record of the life cycle of each of the various data events created as a user goal request is processed by the system. Events and event management in WSMX are described in section 3. WSMX provides a web service interface described in WSDL [WSDL, 2003] to accept service requester goals (see appendix for WSML). When it is running WSMX regularly scans for new messages. Once a new message representing a requester goal is picked up, it is decomposed, validated and translated into an internal persistent WSMX representation by the **Message Parser**. The **Matchmaker** then attempts to match the client goal to a capability known to WSMX. The concept of Capability is used by WSMX-O to formally describe the functionality a web service offers. Any web service with a capability that matches the goal is returned by the Matchmaker. In this version of WSMX, the logical expressions representing the postcondition and effect of the requester goal are compared with the expressions for the postcondition and effect of capabilities for web services known to WSMX. The **Selector** component selects the web service that provides the best match for the goal. The initial version of the Selector in WSMX uses a simple algorithm. The **Data Mediator** component finds a mediator in the **Ontology repository** that can mediate between

the ontology used by the goal (source) and that used by the selected web service (target). Mediation is carried out based on a set of mapping rules between the two ontologies that are applied to the instance data contained in the goal. Once the data has been mediated, it may need to be translated from a logical language format to an XML format before it is included in the SOAP message sent by the Invoker to the target service. This translation is the job of the **XML Converter**. Once the data is in a format that can be used during web service invocation, the **Invoker** makes the actual web service invocation on the selected web service using the mediated and converted data.

The scanner and listener components shown in figure 1 are used in the implementation of an events driven architecture for WSMX. These components are described further in section 5 of this document.

NETWORK

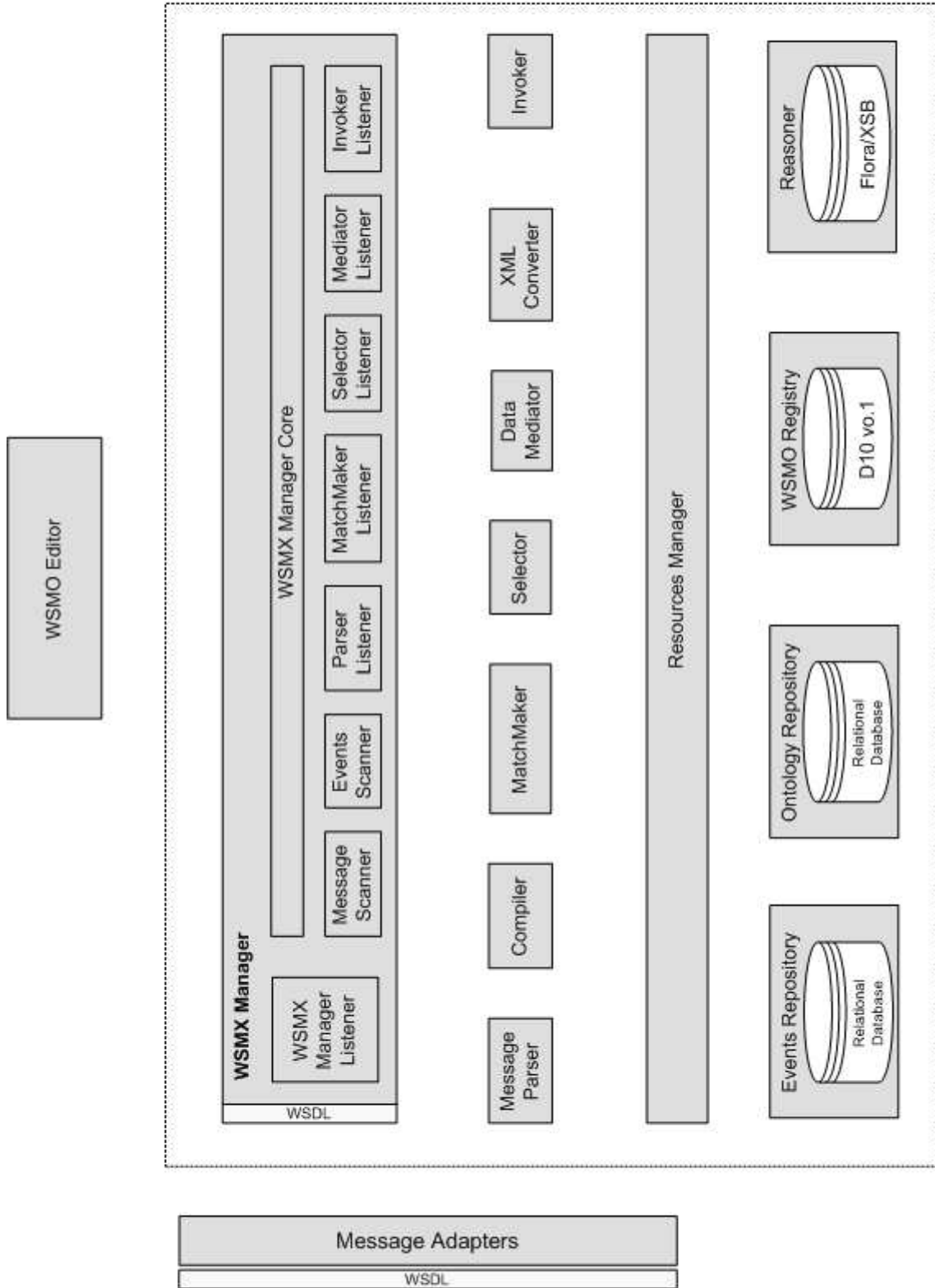


Figure 1 WSMX Architecture Overview

3 Overview of Implementation

3.1 WSMX Web Service Interfaces

WSMX provides two web services. The first service provides an interface for compiling WSML description of concepts relating to web service as described in the WSMX-O conceptual model. Once compiled, the WSML descriptions are ready for use by WSMX at run-time.

Compiler Interface:

WSMX Web Service: Compiler	
CompilationResult	<code>compile(WSMLDocument wsmlDocument)</code>

The second web service provides an interface for to allow external applications or adapters to send WSML messages to WSMX representing service requester goals. Messages received by WSMX while the WSMX Manager core component is not running are stored persistently and automatically processed the next time the core component is started.

WSMXManagerListener Interface:

WSMX Web Service: WSMXManagerListener	
WSMLDocument	<code>receiveWSMLDocument(WSMLDocument wsmlDocument)</code>

The WSDL documents for both web services are provided in the appendix.

3.2 Event Management in WSMX

WSMX is implemented as an event-based system. Service requester goals provided as input to WSMX in the form of a WSML Message go through several different internal data representations before eventually a web service that can satisfy the requester goal is invoked. In WSMX each of these data representations is represented by an event and each event has a fixed number of states it can take.

As described briefly in section 2 and, in more detail in section 5, the WSMX Manager component scans for events in WSMX that have a state indicating the event requires processing. Events are persistent and independent of each other. This provides the framework for a system that can continue processing events after a planned shutdown as well as recovering after an unexpected shutdown or crash. Using an event based architecture isolates the operational aspects of WSMX from the functional aspects provided by components such as the MatchMaker and the Selector. It provides a mechanism allowing WSMX to scale up. As more events are processed by WSMX, more event listeners can be added which in turn may be able to pass the data represented by the event to a pool of relevant components.

4 Datamodel

Where the conceptual model provided in [Cimpian et al., 2004] describes the concepts required for WSMX, the datamodel in this section describes how that conceptual model has been implemented using a relational database management system. Additionally the WSMX datamodel contains tables that are required to support the event management architecture of WSMX. We define the datamodel for this version of WSMX using SQL92 syntax. For this implementation we use the MySQL [MySQL] database system to execute the SQL script against the database. As we use configuration files to connect to the database, any other SQL92 compatible database system could be used instead of MySQL. All tables used by WSMX in this version are included in a single database. This includes the tables implementing the registry of WSMO web service descriptions, WSMO descriptions of other elements related to web services, instances of WSMO goals, instances of mapping rules between specific ontologies and the tables required to support an event based system. The next sections look at the datamodel in terms of different related groups of tables.

4.1 Ontology

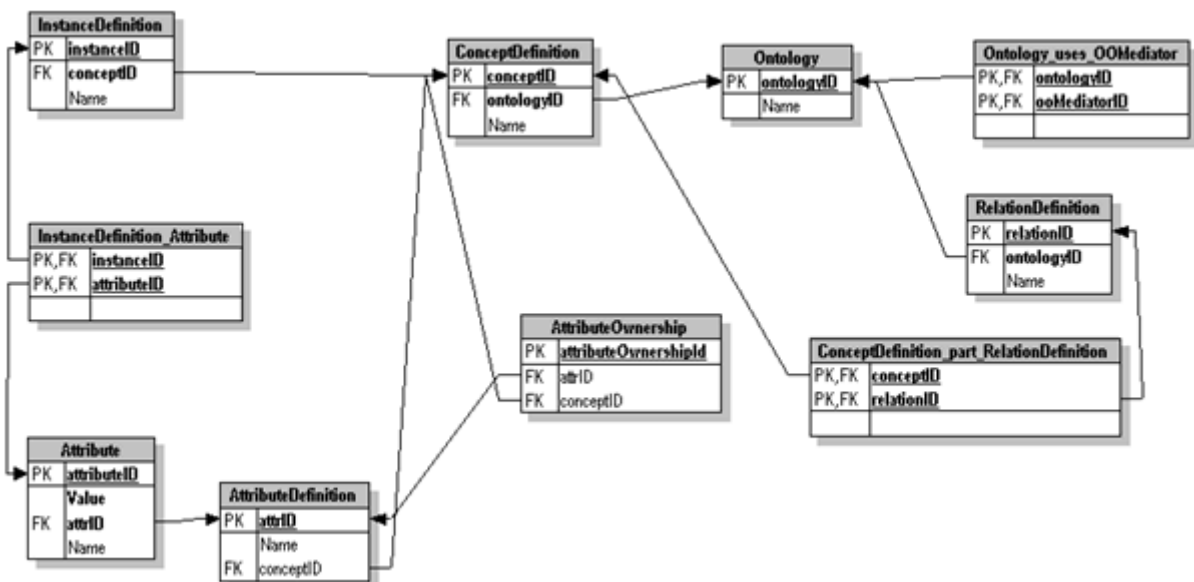


Figure 2 Ontology Tables

An ontology may use the services of several OOMediators and this relation is represented using the table `Ontology_uses_OOMediator`.

Each concept can have attributes (represented in the `AttributeDefinition` table), instances (`InstanceDefinitionTable`). The value of a certain `AttributeDefinition` for an Instance is provided in the `Attribute` table.

The other two tables from the above picture, `ConceptDefinition_part_RelationDefinition` and `AttributeOwnership`, are used to represent in the 5th Normal Form (NF5) the relation between `ConceptDefinition` and `RelationDefinition`, and between `ConceptDefinition` and `AttributeDefinition`.

4.2 Data Mediation



Figure 3 Data Mediation Tables

The name, OOMediator used in the this implementation and in the following represents data mediation at the ontological concept level. An OOMediator is characterized by an unique identifier (the primary key of the OOMediator table) and a name. The mediator offers mediation facilities between a source and a target ontology. For having a normalized database, two additional tables were introduces for modelling these relations: OOMediator_source_Ontology and OOMediator_target_Ontology.

The Mediation Component uses the database for storing/retrieving both the working ontologies and mappings created between concepts or attributes. The Mapping Rules Creator GUI for creating the mapping rules is described in [Mocan & Cimpian, 2004]. This editor reads/writes ontologies and mapping rules from/to the same database tables used by WSMX. Figure 3 presents the tables used by the WSMX Mediation Component and the relations between them:

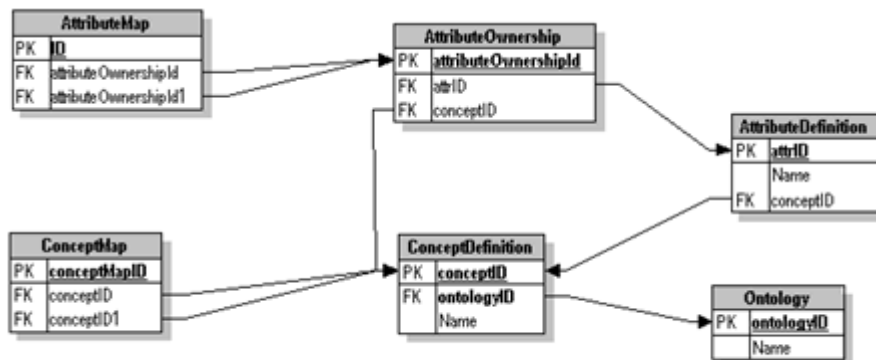


Figure 4 Additional Tables for Data Mediation

The AttributeMap and ConceptMap are used for storing the mappings while the other four tables are used for saving the ontologies, concepts and attributes.

4.3 Web Service

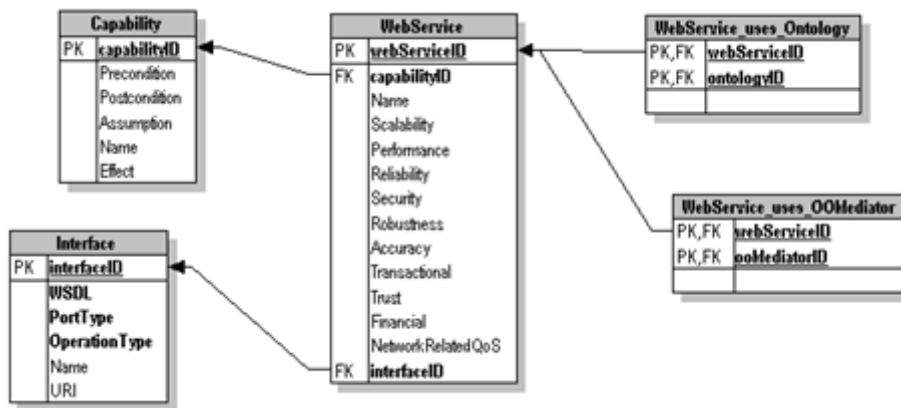


Figure 5 Web Service Tables

A Web Service is characterized by its non functional properties. The primary key of the WebService table is again the unique identifier (webServiceId). Any Web Service offers a Capability, and the way of consuming this capability is expressed by its Interface.

Additionally, for representing the Web Services, we modelled the relations between Web Services and Mediators, and between WebServices and Ontologies, by using two additional tables (WebService_uses_OOMediator and WebService_uses_Ontology).

4.4 Goal

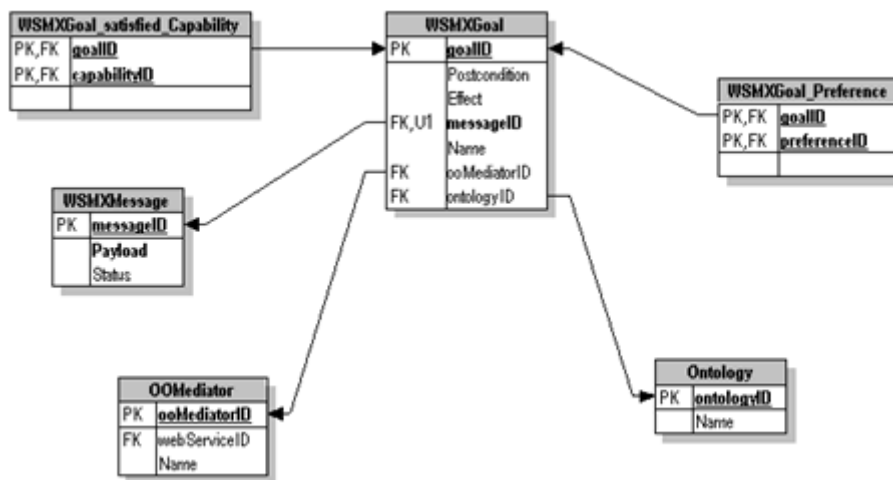


Figure 6 Tables for Goals

The goals are represented by using WSMXGoal table. A goal is characterised by the unique identifier (which is the Primary Key of the table), Name, Postcondition and Effect (there could be more than one postcondition, and more than one effect, but in the initial phase we consider two fields each containing all postconditions and all effects respectively). There also may be some preferences (represented in the WSMXGoalPreference field) which are used by WSMX when selecting a web service to satisfy the goal.

A goal is expressed in terms of one ontology, and may use an OOMediator (in this first version). The relation between goal and capability is modelled by using the

additional table WSMXGoal_satisfied_Capability. Each WSMXGoal corresponds to a WSMXMessage (WSMXMessage table).

4.5 Events

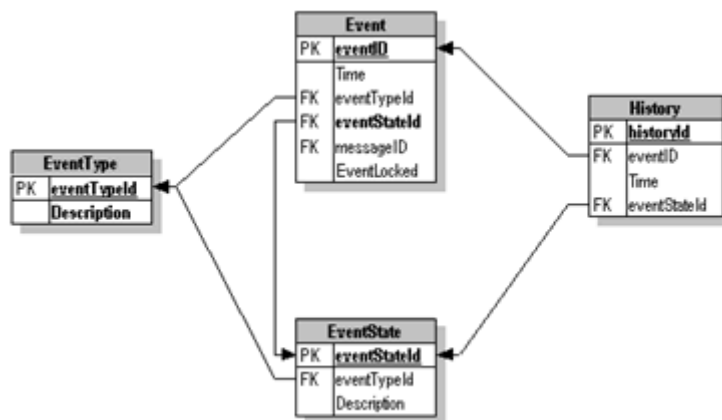


Figure 7 Tables for Events

Each event is characterised by an eventId (unique identifier for the event), type (EventType table) and state (EventState). Not shown in the diagram above is the EventLock table used by WSMX when a particular component ‘locks’ an event for processing. The EventLock table has the fields EventID, State, Component. Each time an event is created or changes states, a new entry should be created in the History table. This has not been implemented in this version of WSMX.

4.6 Messages



Figure 8 Tables for Messages

Data exchanged with WSMX but outside the boundary of the WSMX system are represented by messages. The WSMX message is modelled using the id of the message (which plays the role of primary key in the WSMXMessage table) and the status of the message. All messages received by WSMX are saved without any modification in the WSMX database. Each WSMX Goal is related to exactly one WSMX Message. Similarly each event in WSMX is also related to exactly one WSMX Message.

5 Components

5.1 WSMX Manager

The execution semantics of WSMX as described in [Oren et al., 2004] is implemented in this component. Consequently the WSMX Manager is responsible for co-ordinating the overall operation of WSMX This includes events management, providing the implementation for the web service interfaces offered by WSMX as described in the WSMX WSDL documents [include WSDL in appendix] and control of the start-up and shut-down of the individual process threads for the event listeners described in the following sub-sections.

Note on 'Status' Column in the Following Sections

For simplicity the values in the status column of the following database table representations are shown as text values. In the implementation the status values are represented by enumerated values in the database and by objects containing both an enumerated value and a textual description at the java source code level.

5.1.1 WSMX Manager Listener

The implementation of the web service accepting WSML Messages sent to WSMX either from a message adapter, an external application capable of creating valid WSML Messages or another WSMX instance. This has been described already in this document in the section on the external web service interfaces of WSMX.

5.1.2 WSMX Manager Core

This contains the main method for WSMX. WSMX startup and shutdown are initiated by running the startWSMX.bat and stopWSMX.bat batch files available in the bin sub directory of the WSMX root directory. The WSMX Manager Core starts the process threads for each of the Message Scanner, the Events Scanner, the MatchMaker Listener, the Selector Listener, the Mediator Listener and the Invoker Listener. These components are described in the following sections. Once all process threads are started, the Event Scanner scans for events which are in neither of the states - CONSUMED or ERROR. Each event retrieved is then broadcast to all listener components known to the WSMX Manager Core. Listeners plug-ins are presently hard coded, but in the future we will provide a mechanism to get them easily connected/disconnected from WSMX Manager Core.

5.1.3 Message Scanner

This component scans, at fixed intervals, the WSMXMessage table in the WSMX database representing the Events Repository, looking for entries with the status of NEW_MESSAGE.

Table: WSMXMessage

messageID	payload	status
99	<content of message>	NEW_MESSAGE

When a new message is retrieved from the repository, the component creates a new event in the Events table with type WSML_Message and state CREATED. A new entry is also created in the eventLock table. The eventLock table is used to lock events when they have been picked up for processing by a specific component. Finally the status of the entry for the message in the WSMXMessage table to

EVENT_CREATED

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CREATED	99

Table: EventLock

eventID	state	component
1	UNLOCKED	NONE

Table: WSMXMessage

messageID	payload	status
99	<content of message>	EVENT_CREATED

5.1.4 Events Scanner

The Events Scanner scans, at fixed intervals, the Events table in the WSMX database representing the Events Repository, looking for events whose lockStatus in the eventLock table is IS_UNLOCKED and whose whose state in the Events table is not CONSUMED or ERROR.

A Java object of class Event is created for each retrieved event and added to a collection of event objects stored by the events scanner. This collection of events is used by the WSMX Manager Core to provide the events for processing by the other components in the WSMX Manager.

5.2 Event Listeners

The event listener components are capable of accepting and processing events picked up by the events scanner. Each listener component extends the EventListener abstract class and can handle events of particular types and states. The EventListener class provides the handleNewEvent() method, run() method and the protected data member object SystemComponent for all listeners. The handleNewEvent() method is called by the WSMXManagerCore component to pass events to each of the listeners.

When a listener for a specific component is instantiated, the SystemComponent member object is instantiated with the type of that component. Each listener runs in its own thread and the run() method uses the SystemComponent object to determine whether a specific listener can handle a specific event. If the event can be handled by a listener then it uses the EventHandler class, to update the entry for the event in the EventLock table of the WSMX database setting LOCK_STATUS = IS_LOCKED and systemComponent = <name of the listener>. Once the event has been locked, the runListener() method for the specific listener type is called. The actions carried out by the runListener() method are described in the following subsections. Once the runListener() method has returned, the entry for the event in the eventLock table of the WSMX database is again updated setting

LOCK_STATUS = IS_UNLOCKED and systemComponent = 'none'.

5.2.1 Parser Listener

The Parser Listener handles events of type WSML_MESSAGE with the states CREATED or BEFORE_PARSING. The listener first updates the entry for the event in the events table of the WSMX database to have the state BEFORE_PARSING. Next an instance of the Parser component is created and the parse() method on the Parser is invoked passing the MessageID of the message that should be parsed.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	BEFORE_PARSING	99

Table: eventLock

eventID	state	component
1	LOCKED	PARSER

If the Parser does not return an error or throw an exception, an entry for a new event is created in the events table with type NON_MEDIATED_OBJECT, state CREATED and the MESSAGE_ID set to the same value as the old event. The entry for the old event (type = WSML_MESSAGE) in the events table is then updated to CONSUMED.

If the Parser returns an error the entry for the event in the events table is updated to set the state = ERROR.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	CREATED	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

5.2.2 MatchMaker Listener

The MatchMaker Listener handles events of type NON_MEDIATED_OBJECT with the state CREATED and BEFORE_MATCHMAKING. The listener first updates the event in the events table of the WSMX database to have the state BEFORE_MATCHMAKING.

Table: Event

eventID	type	state	messageID
1	NON_MEDIATED_OBJECT	BEFORE_MATCHMAKING	99

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	BEFORE_MATCHMAKING	99

Table: eventLock

eventID	state	component
1	LOCKED	MATCHMAKER

Using the messageID for the event, the goal description corresponding to the message is retrieved from the WSMX Ontology Repository. An instance of the MatchMaker component is then created and the the match() method is called, passing the goal as input parameter.

If the MatchMaker does not return an error or throw an exception, the collection of matching web services are saved to the Ontology Repository in the table MatchMakedWebServices. The entry for the event in the WSMX database (type NON_MEDIATED_OBJECT) is updated to have state AFTER_MATCHMAKING.

If the Parser returns an error the entry for the event in the events table is updated to set the state = ERROR.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	AFTER_MATCHMAKING	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

5.2.3 Selector Listener

The Selector Listener handles events of type NON_MEDIATED_OBJECT with the state AFTER_MATCHMAKING and BEFORE_SELECTION. The listener first updates the event in the events table of the WSMX database to have the state BEFORE_SELECTION.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	BEFORE_SELECTION	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

eventID	state	component
1	LOCKED	SELECTER

The listener uses the Resources Manager component to get the collection web services found by the MatchMaker that match the goal. The select() method of the selector component is called, passing the collection of web services and goal as input parameters.

If the selector does not return an error or throw an exception, the selected web service selected for the goal is saved in the Ontology Repository in the table, SelectedWebServices. The entry for the event in the WSMX database (type NON_MEDIATED_OBJECT) is updated to have state AFTER_SELECTION.

If the Selector returns an error the entry for the event in the events table is updated to set the state = ERROR.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	AFTER_SELECTION	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

5.2.4 Mediator Listener

The Mediator Listener handles events of type NON_MEDIATED_OBJECT with the state AFTER_SELECTION or BEFORE_MEDIATION. The listener first updates the event in the events table of the WSMX database to have the state BEFORE_MEDIATION and then retrieves the web service selected by the Selector component, from the Ontology Repository.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	BEFORE_MEDIATION	99

Table: eventLock

eventID	state	component
1	LOCKED	MEDIATOR

It also retrieves the goal corresponding to the event being handled, from the Ontology Repository. The data to be mediated is taken from the goal. This data,

along with the identifiers of the ontologies used by each of the selected web service and the goal, are passed to the Mediator component as input parameters of the mediate() method.

If the Mediator does not return an error or throw an exception, the mediated data returned by the Mediator is saved in the MediatedPayload table of the Ontology Repository. A new event is created in the events table with type MEDIATED_OBJECT and state CREATED. The entry for the event in the WSMX database (type NON_MEDIATED_OBJECT) is updated to have state CONSUMED.

If the Mediator returns an error the entry for the event in the events table is updated to set the state = ERROR.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	CONSUMED	99
2	MEDIATED_OBJECT	CREATED	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

5.2.5 Invoker Listener

The Invoker Listener handles events of type MEDIATED_OBJECT with state CREATED and BEF ORE_INVOCATION. The listener first updates the event in the events table to have the state BEFORE_INVOCATION and then uses the messageID corresponding to the event to retrieve the web service to be invoked and the mediated data to be sent to the web service, from the Ontology Repository.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	CONSUMED	99
2	MEDIATED_OBJECT	BEFORE_INVOCATION	99

Table: eventLock

eventID	state	component
1	LOCKED	INVOKER

The invoke() method of the Invoker component is called with the web service and mediated payload data as input parameters.

If the Invoker does not return an error or throw an exception, the entry for the event

in the WSMX database (type = MEDIATED_OBJECT) is updated to have state CONSUMED.

If the Invoker returns an error the entry for the event in the events table is updated to set the state = ERROR.

Table: Event

eventID	type	state	messageID
1	WSML_MESSAGE	CONSUMED	99
2	NON_MEDIATED_OBJECT	CONSUMED	99
2	MEDIATED_OBJECT	CONSUMED	99

Table: eventLock

eventID	state	component
1	UNLOCKED	NONE

5.3 Adapter

Adapters allow external systems using their own message format to communicate with WSMX. In this version of WSMX, an instance of an adapter has been implemented that transforms received UBL (Universal Business Language) [UBL v0.1, 2004] files from a user (through a user interface) or directly from a back-end application, into WSML files. After the Adapter receives a UBL document, this document is parsed looking for specific constants (keywords) in order to decide what kind of UBL file type has been received. Once the received UBL file type is identified, particular data is extracted from it. This particular data provides the necessary parameters to call the doTransformation method from the Transformation.java class. The doTransformation method performs the conversion from the UBL file into the WSML file by the help of a WSML template file. The WSML template file is stored in a local SQL database and it is retrieved from there. The WSML template has several variables that are replaced with specific parameters extracted from the initially parsed UBL files. Once the variables are replaced, the new created WSML file is sent further to WSMX. WSMX is sending back to the user or back-end application the acknowledgement of receipt and an order number.

If the UBL initial file is not recognized, the adaptor returns the message "file not recognized". The conversion performed into the doTransformation method is highly particularized for the received UBL files.

Interface:

Method Summary: ie.deri.wsmx.adapter.UBL2WSMLAdapter	
UBLDocument	transform (UBLDocument source, String wsmxEndpoint)

5.4 Compiler and Message Parser

The functionality of the Message Parser and Compiler components described in the architecture are almost identical and both are implemented in the `ie.deri.wsmx.parser` package. The only difference is in the interface, where the Compiler interface directly accepts a WSML document and the Message Parser interface accepts a messageID referring to a message in the repository.

The `ie.deri.wsmx.parser` package provides the functionality to parse WSML documents to check whether the contents conform to the WSML specification. If a document is indeed valid, the contents of the document are stored in the database representing the WSMX Ontology Repository. The document to be parsed can be passed to the compiler directly, or one can pass the messageID that refers to a message in the WSMX Events Repository. In the latter case, the component first retrieves the message from the repository and then parses it.

Compiling a document is done based on the grammar of WSML, as specified in `ie.deri.wsmx.parser.wsmxlGrammar.g`. From this grammar a lexer and a parser are generated by ANTLR [ANTLR]. The lexer reads a document and constructs tokens from the characters; the parser reads these tokens and constructs an abstract syntax tree from them.

When the document conforms to the grammar, the compiler walks through the abstract syntax tree and constructs objects for every concept it encounters. These objects are part of `ie.deri.wsmx.datamodel`. When the tree is completely traversed, and all necessary objects are created, each object is told to save itself in the repository (not all components, in this version, have been implemented to save themselves but this is our intention). All objects implement the `ie.deri.wsmx.datamodel.util.SerializeWSMXObject` interface, and provide a `saveWSMXObject` method for saving themselves in the repository.

Interface:

Method Summary: ie.deri.wsmx.parser.Parser	
ParsingResult	<code>parse(MessageID messageID)</code> Parse the message corresponding to the input messageID parameter.
ParsingResult	<code>parse(WSMLDocument document)</code> Parse the WSML document passed in as input.

5.5 MatchMaker

The MatchMaker component is implemented by the `ie.deri.wsmx.matchmaker.MatchMaker` class and takes a `WSMXGoal` object as input and then uses the `WebServiceDB` interface in the Resources Manager component to retrieve and populate an array of all known web services from the WSMX Ontology Repository. The array of web service objects and the `WSMXGoal` object are then passed to an internal `match()` method which implements the matchmaking algorithm. The `match()` method in this implementation does a string comparison of the `WSMXGoal` post-condition and effect with the capability post-condition and effect of each web service in the array. The MatchMaker interface returns an array of `WebService` objects.

Interface:

Method Summary: ie.deri.wsmx.matchmaker.Matchmaker

WebService[]	match (WSMXGoal goal) Find web service descriptions matching the goal passed in as input.
--------------	---

5.6 Selector

The current implementation for the Selector component returns the first web service from the collection of web services found to match a specific WSMXGoal. The messageID corresponding to the WSMXGoal is maintained by the Selector Listener component described earlier.

Interface:**Method Summary: ie.deri.wsmx.matchmaker.Selector**

WebService	select (WebService[] webServices, Preference[] preferences) Select one web service to invoke based on preferences associated with the WSMX goal.
------------	--

5.7 Mediator

The WSMX Mediation component offers a default implementation for dealing with the potential heterogeneity problems that may appear during the web services execution process. For the first implementation of WSMX the mediation component offers only ontology to ontology mediation facilities, addressing a limited yet relevant set of problems. In the next versions solutions for addressing a larger set of problems will be implemented as well as solutions for dealing not only with data mediation but also with process mediation.

The current implementation offers the possibility of transforming a given instance expressed in terms of the source ontology in an instance(s) expressed in terms of target ontology. The required inputs for this component are of course, the source instance together with the identifier of the source and target ontology. These identifiers are required for being able to use the appropriate mappings and mapping rules during the mediation process.

The interface to this component is designed for allowing the easy integration of different types of mediators. That is, a factory class is used for creating the appropriate mediator depending on the representation of the instance to be mediated. Currently only a Flora instances mediator is available and it can be retrieved using the following interface provided by the ie.deri.wsmx.mediation.MediatorFactory class.

Interfaces:**Method Summary: ie.deri.wsmx.matchmaker.MediatorFactory**

Mediator	createMediator (int payloadType) This method returns a mediator according to the specified payload type. For this first implementation the only legal value for the payload type is CONSTANTS.FLORA_PAYLOAD.
----------	--

For using the retrieved mediator one has to use the following interface (provided by `ie.deri.wsmx.mediation.Mediator`):

Method Summary: <code>ie.deri.wsmx.matchmaker.Mediator</code>	
<code>StringBuffer</code>	<p>mediate(<code>String sourceOntologyID</code>, <code>String targetOntologyID</code>, <code>StringBuffer payload</code>)</p> <p>The method takes as inputs the source ontology id, the target ontology id, and the payload which is the instance that has to be mediated. It returns the mediated payload, i.e. instance(s) referring to the target ontology.</p>

5.7.1 XML Converter

Descriptive paragraph required here including naming any tables from the datamodel used by this component.

5.8 Invoker

The Invoker component in this version of WSMX is implemented as follows. The URI for the WSDL of the selected web service is retrieved using the Interfaces member object of the `WebService` class. The Interfaces member object is of type `java.util.Vector`. At present WSMX takes the first Interface object from the Vector and uses this object to get the URI for the WSDL document corresponding to the web service to be invoked.

The WSDL4J library provided with Apache AXIS is used to parse the WSDL document and retrieve the actual service endpoint. The operation to invoke on the selected web service is retrieved from the web service's Interface object (again just taking the first operation in the collection). The Apache Axis SOAP implementation is used to make the actual web service invocation. Initially the web services used to test this implementation of WSMX expect data of type `RosettaNetDocument` and return data of type `RosettaNetDocument`.

Interface:

Method Summary: <code>ie.deri.wsmx.invoker.Invoker</code>	
<code>InvokerResult</code>	<p>invoke(<code>WebService webService</code>, <code>Document document</code>)</p> <p>Call the invoker component that actually makes the call to the web service passing in the <code>WebService</code> object representing the <code>WebService</code> to invoke and the high level <code>Document</code> class containing the data that should be passed to the web service.</p>

5.9 Resources Manager

The Resources Manager component represents an abstraction layer between the persistent storage of WSMX and the other WSMX components. It provides the required functionality for accessing both the Ontology Repository and the Events Repository, both of which are implemented using a MySQL relational database in this version of WSMX.

Currently, the Database Manager consists of the following packages: `dbOntology`,

dbMediation, dbWebService and dbCapability, dbGoal, dbEvent, dbMessage and dbUtil. A brief description of all these packages is provided in the following paragraphs.

dbOntology

The dbOntology package offers methods for accessing all the tables related to WSMX (WSMO) ontologies (see Figure 2). These methods include: the creation of new entries in the database, the deletion and/or the update of already existing entries, and retrieving of information for further use.

dbMediation

The dbMediation package offers the methods needed for storing, deleting, updating or just consulting the tables containing information about the Data Mediator (Figure 3) and the mappings between concepts and attributes in two ontologies (Figure 4).

dbWebService and dbCapability

The dbWebService, together with the dbCapability, are used for performing the needed operation on the tables containing information referring to the Web Services (Figure 5).

dbGoal

The dbGoal package is used for managing the tables containing information about the service requester goals (Figure 6).

dbEvent

The dbGoal package is used for managing the tables relating to events (Figure 7).

dbMessage

The dbMessage package is used for managing the tables relating to messages (Figure 8).

6 References

[ANTLR, 95] Parr, T.J. and R.W. Quong, Jul 1995: ANTLR: A predicated-LL(k) parser generator. *Software, Practice and Experience*, 25(7), pp. 789-810

[Cimpian et al., 2004] Cimpian E., Mocan A., Moran M., Oren E., Zaremba M. (2004). WSMX Conceptual Model. WSMO Working Draft v0.1, Digital Enterprise Research Institute (DERI), available from <http://www.wsmo.org/2004/d13/d13.1/v01>

[Fensel & Bussler, 2002] D. Fensel, C. Bussler, *The Web Service Modeling Framework WSMF*. *Electronic Commerce Research and Applications*, Vol. 1, Issue 2, Elsevier Science B.V.

[Mocan & Cimpian, 2004] A. Mocan, E. Cimpian . WSMX Mediation, WSMO Working Draft v0.1, 26 May 2004, Digital Enterprise Research Institute (DERI),

available from <http://www.wsmo.org/2004/d13/d13.3/v0.1>

[MySQL] MySQL OpenSource database server, available at <http://www.mysql.com/>

[Oren, 2004] E. Oren. WSMX Execution Semantics, WSMO Working Draft v0.1, 31 May 2004, Digital Enterprise Research Institute, available from <http://www.wsmo.org/d13/d13.0/v0.1/20040531>

[Oren et al., 2004] E. Oren, M. Moran, M. Zaremba, Overview and Scope of WSMX, WSMO Working Draft v0.1, 24 May 2004, Digital Enterprise Research Institute (DERI), available from <http://www.wsmo.org/2004/d13/d13.0/v0.1/20040524/index.pdf>

[Roman et al., 2004a] D. Roman, H. Lausen, U. Keller. The Web Service Modeling Ontology Standard (WSMO-Standard) v0.2, 6 March 2004, Digital Enterprise Research Institute. Available at <http://www.wsmo.org/2004/d2/v0.2/20040306/>

[UBL v0.1, 2004] B. Meadows, L. Seaburg (eds.), Universal Business Language (UBL) 1.0 Committee draft, <http://docs.oasis-open.org/ubl/cd-UBL-1.0/>

[WSDL, 2003] Web Services Description Language (WSDL) 1.2, W3C Working Draft, 3 March 2003

[Zaremba et al., 2004] M. Zaremba, M. Moran, E. Oren, E. Cimpian, A. Mocan, WSMX Architecture, WSMO Working Draft v0.1, 30 May 2004, Digital Enterprise Research Institute, available from <http://www.wsmo.org/2004/d13/d13.4/v0.1>

7 Acknowledgement

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 programm.

The editors would like to thank to all the members of the WSMX working group for their advice and inputs to this document.

8 Appendix

8.1 Javadoc Description for WSMX

The javadoc descriptions are available at [WSMX JavaDoc](#)

8.2 Mapping Rules Editor Implementation

8.2.1 Graphical User Interface

It is well stated now that the mapping creation process cannot be entirely performed automatically. As a consequence, the challenge remains how to offer better assistance to the human user during the mapping process, in order to reduce his/her efforts to simple choices and validations. For this, a graphical tool which is able to

offer visual representation of his tasks and actions is a must, as part of any mediation component.

The WSMX mediation component includes a graphical user interface for presenting the ontologies, for offering suggestions and guidance during the mapping process, and for providing facilities for saving/loading the already created mappings. This GUI is used during the runtime and can be integrated in other applications by just calling the constructor of the class `ie.deri.wsmx.mediation.gui.UserInterface`.

8.2.2 Mapping Rules Creator

This subcomponent has the role of transforming the mappings created using the GUI in mapping rules. It loads the necessary mappings from the database and uses them for the creation of the appropriate rules. The rationale for this component is to offer a strongly decoupled architecture for the mediation component and the possibility of plugging different mapping rules creator modules, according to the available execution environment for the rules. For this version only a module for creating Flora mapping rules is offered and it can be used by calling the following methods from `ie.deri.wsmx.mediation.flora.FloroRulesGenerator`:

Method Summary: <code>ie.deri.wsmx.mediation.flora.FloroRulesGenerator</code>	
<code>StringBuffer</code>	<code>getMappingRule(String srcConcept, String tgtConcept)</code> Creates the mapping rule for transforming the instance of the source concept in an instance of the target concept.
<code>java.io.File</code>	<code>saveMappingRule(String srcConcept, String tgtConcept)</code> Puts flora mapping rule in a file; asks the user to choose a destination file.



[webmaster](#)