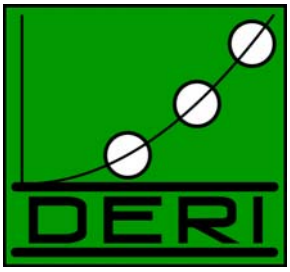

WSMX DOCUMENTATION

Author: Maximilian Herold

Digital Enterprise Research Institute Galway, Ireland

Status: Final
Version: 1.0
Date: 2008-05-01

Abstract

This document provides documentation on and guidelines for using the current WSMX prototype implementation ("current" is referring to the SVN repository version on the date of this document).

Document history

Date	Rev.	Description / Changes	Author	Reviewed by
2008-05-01	V1.0	ADDED: note on discovery mediation; SWING UC1; choreography response modification; … UPDATED / CHANGED: data mediation and discovery related sections; a lot in the installation guidelines section; architecture figure; choreography controlled state related sections; several fixes …	Maximilian Herold	Maciej Zaremba
2008-03-19	V0.4	Initial document draft	Maximilian Herold	Maciej Zaremba

Note on document updates

When any WSMX components functionality are changed, please check especially the following sections of this document for any necessary updates:

- 1.4 Execution semantics – exemplified by AchieveGoal
- 2.2.3.2 Configuration Options
- 3 SWS Development and Examples
- 4.1 Components Overview
- Appendix (FAQ)

Open issues / TODOs

- Core / Architecture / Common
  - AchieveGoal process figure (e.g. using BPMN)
- Invoker
  - Lifting / lowering: explain the AdapterOntology method
- Discovery
  - More details on QoS and instance-based discovery

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

ACRONYMS

ASM	Abstract State Machine
GUI	Graphical User Interface
NFP	Non-functional property
OASM	Ontologized Abstract State Machine
SEE	Semantic Execution Environment
SOA	Service Oriented Architecture
SSH	Secure Shell
SWS	Semantic Web Service
TUI	Text User Interface
WSDL	Web Service Description Language
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology
WSMT	Web Service Modeling Toolkit
WSMX	Web Service Execution Environment

1 INTRODUCTION

1.1 About this document

This document provides the documentation for the current version of WSMX as an implementation prototype for the WSMO architecture described in detail in several previous documents, amongst others in [1] and [5], whereas the latest updates and additions can be found in [1]. This document is a fully revised, updated and extended version of previous versions of the prototype description ([2],[6]).

The aim of this document is to act as a guide to WSMX, to show the current state of implementation, as well as to outline ongoing and future developments. The targeted audience is anyone with a general interest in WSMX, in using the prototype, or in further developing WSMX, e.g. by writing a new component. Selected aspects may be skipped in this document for simplicity, but references to theoretical background and more detailled descriptions are provided wherever applicable.

The remainder of this first section gives an overview of WSMX and its main concepts and technologies. It concludes outlining how the system works as a whole. Section 2 contains information on WSMX releases, the installation, the build process; how to run, configure access, monitor, and administer WSMX; and related tools and libraries. Section 3 describes how to use WSMX in context of developing semantic web services and outlines several examples. Section 4 provides detailled descriptions of the currently implemented WSMX components, including their current development state. In addition, it contains useful hints for developers. Section 5 concludes with an outlook on current and future developments. Answers to frequently asked questions not addressed in the main body of the document can be found in Appendix A.

1.2 Purpose, employed concepts and technologies

WSMX is an execution environment for semantic web services (SWS). In contrast to regular web services, which are only described on a syntactical level, SWS include additional semantic information in their descriptions. This aims at enabling the automation of tasks like service discovery, selection, mediation, invocation, and dynamic interoperation of web services. The goal is to enable a scalable SOA in which content and process level interoperability do not need to be solved by using hard-wired manual configurations of services or workflows.

More precisely, WSMX is a reference implementation for a semantic execution environment (SEE) middleware[1] based on the conceptual model provided by the Web Service Modeling Ontology WSMO [3]. Accordingly, the Web Service Modeling Language WSML [4] is used as the underlying language for the semantic descriptions. The SWS descriptions are grounded to standard WSDL-based web services.[2]

[1] as specified within the OASIS SEE Technical Committee, see http://www.oasis-open.org/committees/semantic-ex

[2] for detailled descriptions see: WSMO: [3],[9]; WSML: [4]; SWS descriptions and grounding: section 3.1

1.3 Architecture and functionality overview

WSMX itself is based on SOA principles. It is divided into self-contained components which provide different functionality to the system. From a service point of view, the components act as middleware service providers. The communication between the components is event-based, using a publish-subscribe mechanism.

The central component in WSMX is the Core (see Figure 1). It provides middleware framework functionality such as finding and loading components, handling the messaging between components, and defining paths of execution ("execution semantics", see below). The Core also defines the types of components in the Integration API. In essence, the Core is responsible for all cross-cutting concerns regarding the other components (i.e. the Core realises the vertical layer as described in [1]).

The other components are decoupled from the each other (incl. from the Core itself) by wrappers. This architecture enables to easily plug in additional components and to switch any of the provided components to a third-party component that realises the same functionality.

The currently available components, apart from the Core, include: choreography, communication manager, data mediator, invoker, parser, resource manager, service discovery, and web service discovery[3]. The following section gives an example how they are used in WSMX to process SWS.

[3] for a more detailled description on each of the components and their current status, please refer to section 4.1.

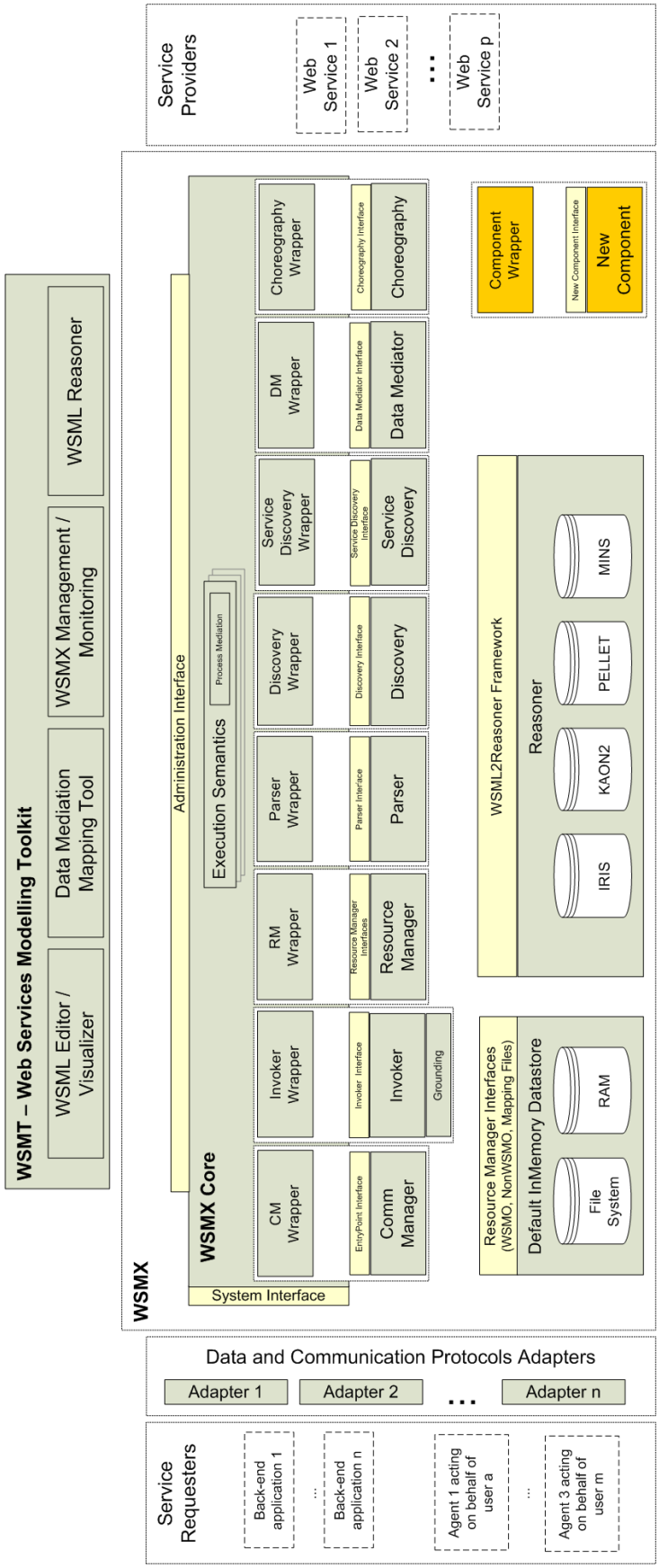

Figure 1: Current WSMX Architecture

1.4 Execution semantics – exemplified by AchieveGoal[4]

Execution semantics can be viewed as executable processes that define a path through the system. Thus they enable the combined execution of functional components. Execution semantics are specified by the core and are triggered by specific requests to the system[5]. Currently, there is one execution semantic specified: AchieveGoal. It defines the goal-based invocation of services, i.e. the process to be executed in order to get a response for a specified goal.

AchieveGoal comprises of the following steps (for a detailled description of the involved components, refer to section 4.1):

(1) **Late Binding: Service Discovery and Selection** (the following steps are executed sequentially and just once each):

a. **Web Service Discovery**: The given goal definition is used to find web service definitions in the repository which can fulfill that goal.

Currently, a keyword discovery which regards NFPs and a WSML-Flight-based discovery which regards postconditions are used as a default. Others can be activated..

Note that mediation for discovery (i.e. between web service and goal descriptions on the level of pre-/postconditions) is still an open issue, so in this part of the service descriptions the same ontologies need to be used in order for a service to be discovered.

[Components: Web Service Discovery in `/discovery`]

b. **Service Discovery** (optional): It may be necessary that web services need to be invoked with the actual instance data that was provided with the goal in order to determine whether a service can actually fulfill the concrete goal[6].

See section 3.1.6 and examples in section 3.3.

[Components: Service Discovery in `/serviceDiscovery`]

c. **Service Selection** (optional): Additional criteria like QoS and user preferences are used to rank the discovered services.

See section 3.1.6 and examples in section 3.3.

Currently, after this step, the top ranked service is selected by the execution semantics. Conceptually, there is supposed to be an optional validation step following this ranking in which a combination of user approval and service analysis methods can be used to select the service.

[Components: Service Discovery in `/serviceDiscovery`]

(2) **Execution:**

a. **Process Mediation (Orchestration/Choreography)** resolves process heterogenity between requester and provider. This may e.g. be needed when

[4] this section is based on [1], where a conceptually complete execution semantics is depicted

[5] The ways to acces WSMX and trigger an execution semantic are described in section 2.3. See also: 4.1.3 (Communication Manager)

[6] **abstract vs. concrete goal:** [1] refers to the goal definition as *abstract goal* and to the goal including concrete instance data as *concrete goal*. An example of an abstract goal would be "book a flight from ?x to ?y on the ?date", a concrete goal would be "get a flight from Paris to New York on the 25th May, 2008". Accordingly, discovery first takes place on the abstract level (web service discovery) and then on the concrete level (service discovery).

the provider service has a different granularity and multiple invocations need to be made to fulfill a goal. To achieve this, the choreography interfaces of both goal (requester) and service (provider) are processed, taking into account any concrete data. This data can be initially provided with the goal, changed by choreography rules, or received from a service. This subprocess can be seen as a conversation between requester and provider. There may be several steps involving updating the processing memory of the choreography component regarding requester or provider, mediating data, and invoking a service.

Currently, only the Choreography component is integrated and used, which is sufficient in all current use cases (although it is sometimes misused for orchestration funcionality). See also section 3.1.4.

[Components: Choreography in `/choreography`]

b. **Data Mediation** resolves data heterogenity between requester and provider. It transforms instances of the ontologies known to the requester to instances of the ontologies known to the provider, or vice versa. Assuming there is a data hetereogenity problem, data mediation is performed two times[7]: at the beginning, to convert the instances provided by the goal to instances of the web service; and at the end, to convert the response instances of the web service to instances comprehensible by the goal.

The implementation of data mediation is based on mapping files (.map) that describe mappings between ontologies in an abstract mapping language. They can be created using WSMT (see section 2.4.1). An OO-Mediator needs to be registered for each mapping file to let the system know about the availability of the mapping. See section 3.1.3.

[Components: Data Mediator in /mediator]

c. **Service Invocation** and **Grounding**: The information which WSDL operation of a web service needs to be called with what instance data and the type of instances that is returned is specified in the grounding information of the WSMO web service description. For the actual invocation of the WSDL-based web service, the input data needs to be lowered from ontology instances to an XML-based message and later the response lifted to ontology instances.

Currently, this transformation needs to be hard-coded; the lifting of XML to WSDL may be defined declaratively e.g. via XSLT. A generic grounding mechanism without the need for code changes is yet to be implemented. See section 3.1.1.

[Components: Invoker in /invoker]

It should be noted that from the WSMO *xx*-Mediator conceptual elements, only OO-Mediators are (partially) implemented and used at the moment. Nonetheless, process and data mediation are available; only the way they work might diverge in some details from the description in WSMO conceptual documents.

[7] if orchestration was integrated into WSMX, it could be more than two times, depending on the number of orchestration steps

2 INSTALLING AND USING WSMX

2.1 Release Information

2.1.1 Available Releases

2.1.1.1 Sources via SVN

The WSMX sources are available on sourceforge.net via anonymous access to the Subversion repository: http://sourceforge.net/svn/?group_id=113321. ***Anyone who wants to do more than testing the examples needs to obtain the sources and build WSMX***[8].

2.1.1.2 Binary distribution

The binary distribution can be downloaded at http://sourceforge.net/projects/wsmx. The available files are:

- **distribution**: the full distribution
- **wsmx-core**: the core component, incl. WSMX Integration API
- **wsmx-components**: all the other currently implemented components
- **wsmx-integration-api**: only the WSMX Integration API incl. javadoc *(needed to build an SEE compliant infrastructure like WSMX, or to implement a component; defines types of components and related data types)*

For official releases, a detailed list of all the improvements and changes of the WSMX prototype is available in the CHANGELOG at http://wsmx.svn.sourceforge.net/viewvc/wsmx/trunk/CHANGELOG?view=markup.

2.1.1.3 Nightly builds

It should be noted that WSMX is under constant development, which may not be reflected in the release cycles on sourceforge.net. In order to benefit from the latest features and bugfixes, a nightly build binary distribution is available at http://www.wsmx.org/downloads.html.

2.1.2 License Information

WSMX uses the GNU Lesser General Public Licence[9].

The third party software components and libraries included in the current WSMX release are using diverse licences. Since the used libraries still change frequently, they will not be listed or updated in this document.

2.2 Installation guidelines

2.2.1 Software Prerequisites

- **Required**
  - *Java JDK 6 Update 6*, available from `http://java.sun.com/javase/downloads/`
- **Optional**

[8] this is due to the fact that, in the current implementation, the addition of a new web service still needs some minor code additions and recompilation for the lifting/lowering step, as described in section 3.1

[9] http://www.opensource.org/licenses/lgpl-3.0.html

- **For distributing different components to different machines / JVMs**: a JavaSpaces compliant installation in order to start a tuple space, e.g.: *JINI*, available from https://starterkit.dev.java.net/downloads/index.html and *IncaX & JavaSpaces Starter Kit*, available from http://www.incax.com/. IncaX is a visual development and deployment platform for JINI. Follow the respective installation instructions.

2.2.2 Building WSMX

If you already have a binary distribution and don't need to build from the sources, you may skip this step.

An overview of the structure of the WSMX sources is shown in the following listing:

```
adaptation                  (work-in-progress, currently not used)
adapter                     (work-in-progress, currently not used)
choreography                Choreography component
client                      WSMX SOAP client
common                      libraries used across components
communicationmanager        Communication Manager component
conf                        (deprecated artifact)
core                        Core component; default config; incl.
                              sources for common libraries and
                              WSMX Integration API
dbManager                   (currently not used)
discovery                   Web Service Discovery component
docs                        WSMX documentation (this document)
invoker                     Invoker component
mediator                    Data Mediator component
monitoring
orchestration               Orchestration component (currently not used)
parser                      Parser component
processMediator             (currently not used)
reasoner                    (deprecated artifact)
resourcemanager             Resource Manager component
resources                   contains resources to be used at runtime
serviceDiscovery            Service Discovery component
thirdparty                  thirdparty libraries used by different
                              components
tools                       build related sources and artifacts (build
                              snippets used in component build files)
unittest                    main junit test cases
visualisation               SWING-based Visualizer ("WSMX Monitor")
webservices                 WSMX entry points as web service
```

Listing 1: WSMX source structure reference

2.2.2.1 Prerequisites

Apache Ant 1.7.0 or higher[10] is needed to build WSMX.

To build from inside Eclipse[11], you need to add the following to the Ant Runtime Classpath[12]:

- WSMX related: `common/lib/ant-wsmx.jar` (add to Tasks: Task name `WSMXClasspath`, Class `ie.deri.tools.ant.taskdef.WSMXClasspath.class`)

[10] available from http://ant.apache.org/
[11] http://www.eclipse.org/
[12] set in `Window → Preferences`

- JAX-WS related: `thirdparty/jax-ws/2.0/jaxws-tools.jar`, `jaxws-api.jar`, and `jaxws-rt.jar`

To build from the command line, you only need to have the mentioned JAX-WS related libraries in your classpath[13].

2.2.2.2 Build Overview

The main build file is `/build.xml`. To run a complete system build, use the `build.distribution` target. Other important targets are included in the list below.

```
build.distribution          complete WSMX system build (to /dist)

clean                       deletes the /dist directory

run                         runs WSMX from the /dist directory
shutdown                    shuts WSMX down (using MBean via HTTP access)

output.xxx                  build component xxx (as .wsmx JAR to /dist)

core.create.*               rebuild libraries from core sources that are
                              used across components (see below)

choreography.create.asm     build asm library used by choreography and
                              orchestration (to thirdparty/wsmx/…)
```

Listing 2: Main build targets

There are several common libraries used by the WSMX components (incl. the core) for which the source code is located in `/core`. These include:

- **annotations**: WSMX specific annotations (binary in `/thirdparty/wsmx`)
- **commons**: WSMX commons library (binary in `/thirdparty/wsmx`)
- **environment**: WSMX environment, includes `Environment` and execution semantics classes (binary in `/thirdparty/wsmx`)
- **infomodel**: WSMX Integration API (binary in `/commons/lib`)

If any of the related source code changes, the according `core.create.*` target needs to be executed before running a system build.

2.2.2.3 How to call a target of a component's build file

With the current build system it may not be possible to call a target in a component's build file directly because the classpath to thirdparty files will not be resolved correctly.

Especially when using Ant inside Eclipse, you need to add a target in the main file just calls the desired target the component's build file, e.g.:

```
<ant antfile="build.xml" dir="componentDirectory" target="theTarget"/>
```

From the command line, you may call the target directly if your current directory is the WSMX root directory. In that case, you would need to specify the build file, of course, e.g.:

```
ant –f componentDirectory/build.xml theTarget
```

[13] e.g. by adding them to the `CLASSPATH` environment variable

2.2.2.4 External libraries integration and component build files

If a component requires any of the libraries in `/thirdparty` or `/commons`, they simply need to be referenced by file name in the component's build file:

```
<path id="component.classpath">
     <pathelement path="theThirdpartyLib.jar"/>
</path>
```

Libraries referenced in this way will be available at compile time and also be added to the component's `lib` directory when built. Any libraries defined in `complete.classpath` in the `/tools/init.ent` file do not need to be added to the `component.classpath`, since they will be provided automatically to all components.

Note: If a new library is added to the `complete.classpath` in `/tools/init.ent`, it also needs to be added to the component.classpath in the Core component's build file (this ensures that it is packed with the Core component and thus also available at runtime).

If writing a new component, use one of the other component's build files as a reference on how it should be structured. Some components still have a XML configuration file, which is not necessary when using the `@WSMXComponent` annotation.

2.2.2.5 Note for WSMX committers[14]

Please use the `build.distribution` target from the main build file as a judge before you check in anything, not only your components buildfile.

The reason for this is that even though we've pretty good isolated the build success of components from each other there are still situations where a modification for one component can break another.

With the current build system these situations should be limited to:

- if you remove or rename a library from `/thirdparty`;
- if you add a library with the same name as an existing library (this kind of ambiguity is resolved non-deterministically by the build system);
- if you modify your component's interface and the integration API.

Additionally, these things will probably not raise a compile error but will cause an exception at runtime:

- if you diverge from the build system's current default WSMO4j version.

2.2.3 Running and configuring WSMX

The binary distribution contains the following files and folders:

```
startc.bat                  startup script to start WSMX on Windows
wsmx.core                   WSMX core component, packed as JAR
*.wsmx                      the other WSMX components, packed as JARs
                            (currently including: choreography,
                            communicationmanager, datamediator, invoker,
                            orchestration, parser, resourcemanager,
                            serviceDiscovery, and webServiceDiscovery)
config.properties           the configuration file
resources/                  folder containing resources used by various
                              components, incl. several examples
axisWSMXEntryPoints.war     Apache Axis web service archive, providing
```

[14] based on a mailing list entry by Thomas Haselwanter

```
                                web services for WSMX access
policy.all                     java policy file
lib/                           folder containing some additional
                                 libraries
```

Listing 3: Binary distribution contents

2.2.3.1 Starting and stopping WSMX

Please use the provided startup script to start WSMX.

To shut down WSMX, send the process a kill signal (`Ctrl-C` in most operating systems). A shutdown option is also available via SSH or HTTP access (see section 2.3).

If you run from your own build or from inside Eclipse, you may also use the `run` and `shutdown` targets in the Ant build script.

For **previous versions** of WSMX it might be the case that there is no startup script in your specific distribution. You may create one yourself then or just start WSMX from the command line. Make sure to include the policy file, e.g.:

```
java –Djava.security.policy=policy.all –jar wsmx.core
```

Note: In some cases the default heap memory available to the JVM might not be sufficient during execution and OutOfMemoryErrors might be thrown. This might happen if you use services with large amounts of request or response data. The heap memory settings can be changed by adding the following switches to the command line: `-Xmx200m -Xms200m` (this example sets the maximum and minimum heap size to 200 MB, respectively).

2.2.3.2 Configuration Options

The configuration file, `config.properties`, resides in the same directory as the core component. It is well commented and self-descriptive. The following aspects can be configured:

1) **Network** configuration
    a. Assignment of ports for HTTP, SSH, and Web Services (Apache Axis); Tuplespace address[15]
    b. Proxy settings for Apache Axis
2) Password for **SSH access**
3) Optional setting of the **systemcodebase** (specifies where the core looks for WSMX components to load; by default it is the directory of the core component)
4) **Monitoring**:
    a. Visualizer: whether the Java SWING-based Visualizer (“WSMX Monior”) should be activated
    b. ActiveMQ message visualizer activation *(will not be futher described in this document)*
5) **Resource Manager** component:

[15] only needs to be changed for distributing WSMX components across multiple machines; must be the same for all WSMX instances participating in the distributed WSMX system

a. setting of **resource directories** from which WSMX will load WSML resources and .map mapping files for data mediation at startup (see actual configuration file for more details)

b. goals specified in (a) will only be loaded in server mode; this option allows to specify which goals are to be loaded in offline mode (e.g. JUnit tests)

6) **Service Selection (QoS Discovery)** component: whether to activate; (please refer to sections 2.2.1 and 4.1.9 for additional prerequisites and information regarding QoS Discovery)

7) **Data Mediator** component flags:

- `filter_mappings_based_on_input`: (should be `false`)
- `transform_only_connected_instances`: (should be `false`)
- `log.*`: logging configuration (if activated, the data mediator will use the `org.deri.wsmx.mediation.ooMediator.wsmx.RMLoader` log4j logger)
- `write_merged_ontology_to_file`: if activated, writes a file `wsml-rules.wsml.tmp` to the working directory which contains the merged ontology

8) **Logging** Ouput (Apache Log4j[16]; default log output goes to `/wsmx.log`)

```
# 1a) Set ports for the WSMX daemons and the space address
wsmx.spaceaddress=localhost
wsmx.httpport=8080
wsmx.axisport=8050
wsmx.sshport=8090

# 1b) Axis proxy settings
http.proxySet=false
#http.proxyHost=cache.nuigalway.ie
#http.proxyPort=8080

# 2) login root/dedication at the SSH console
wsmx.ssh.rootpassword=dedication

# 3) Let the kernel find out the systemcodebase itself
#wsmx.systemcodebase=/home/wsmx/systemcodebase

# 4a) flag for SWING based simple visualizer tool
wsmx.visualizer=false
# 4b) flag for ActiveMQ message visualizer
wsmx.monitoring=false

# 5a) references to the locations from where WSMO Entities
#      and mapping files for data mediation can be found
wsmx.resourcemanager.ontologies =  "$(resources)/Ontologies"
wsmx.resourcemanager.goals =  "$(resources)/Goals"
wsmx.resourcemanager.webservices = "$(resources)/Webservices"
wsmx.resourcemanager.mediators = "$(resources)/Mediators"
wsmx.resourcemanager.mappings = "$(resources)/mappings"
# 5b) load following goals only in offline (non-server) mode
wsmx.resourcemanager.goals.offline = "$(resources)/Goals/offline"

# 6)
```

[16] see http://logging.apache.org/log4j/1.2/manual.html for details on log4j configuration

```
wsmx.discovery.qosdiscovery = true
wsmx.discovery.qosdiscovery.createDB = false
```

```
# 7) datamediator flags
wsmx.datamediator.filter_mappings_based_on_input = false
wsmx.datamediator.transform_only_connected_instances = false
wsmx.datamediator.log.ontologies = false
wsmx.datamediator.log.queries = true
wsmx.datamediator.log.timing = true
wsmx.datamediator.write_merged_ontology_to_file = false
```

Listing 4: Sample config.properties configuration file

2.3 Access, monitoring and administration

2.3.1 WSMX entry points

The WSMX entry points represent the business functionality exposed by the system. They include:

- **achieveGoal**: starts the AchieveGoal execution semantics as described in section 1.4

  input: goal to achieve incl. input data (instances); returns result instances

- **discoverWebServices**: discover web services based on a goal

  input: goal; returns discovered web services

- **invokeWebService**: invoke a web service

  input: web service, goal with instances; returns result instances

- **store**: store a WSML entity

  input: wsml entity

The operations are available using either the Server Administration GUI (HTTP) or via web services; see below.

2.3.2 Access via web service

The WSDL of the web service exposing the WSMX entry points and additional operations (like storing or retrieving WSML entities) is available at http://localhost:8001/jaxws/WSMXEntryPoints?wsdl *(JAX-WS, port fixed)* or http://localhost:8050/axis/services/WSMXEntryPoints?wsdl *(Axis, port as specified in the configuration)*.

Invoking a web service operation may either be done programmatically or using tools such as StrikeIron[17], SoapUI[18] or WSMT (see below).

The other available web services (see http://localhost:8050/axis/services/listServices) are used for examples.

2.3.2.1 Access web services via WSMX SOAP Client

The WSMX SOAP Client is available in the source distribution (`/client`). It is a proxy to the String-based web services that adds the additional functionality to use WSMO4j objects directly.

[17] http://www.strikeiron.com/
[18] http://www.soapui.org/

2.3.3 Access via WSMT (Eclipse)

WSMT (see section 2.4.1) provides the possibility to view, upload and delete WSML artifacts from WSMX as well as sending SOAP messages to the WSMX entry points web service.

To set up WSMT for these tasks, switch to the SEE perspective and create a `New → WSMX Server`, specifying the ports as configured in WSMX. The SOAP Message View facilitates sending a message to the achieveGoal entry point and viewing the result message (see Figure 5 on p.23).

2.3.4 Server Administration GUI (HTTP)

The WSMX Management Console is available at http://localhost:8080/ *(port as specified in the configuration).* It contains general information about the running WSMX instance and facilitates basic administration tasks.

Especially note the *server view* (see Figure 2), where important functionality from different domains (e.g. Core services, Component services) is exposed via MBeans[19]. Examples of the functionality include:

- WSMX shutdown using the `core/WSMXKernel` MBean
- WSMX entry points via the `components/Communication Manager` MBean

Note that several of the components' operations are also exposed as web service (see above).

2.3.5 Server Administration TUI (SSH)

A Terminal User Interface (TUI) for the basic administration tasks as depicted in Figure 3 is available via SSH using

```
ssh root@localhost -p 8090
```

(port and password as specified in the configuration).

2.3.6 "Visualizer" WSMX monitor

The WSMX Monitor (SWING-based "Visualiser") can be used to trace service discovery, choreography, and WSML messages between service requester and provider. The Visualizer opens automatically as soon as it gets any input from the respective components, provided it has been activated in the configuration. See Figure 4.

[19] a JMX MBean (managed bean) is a Java object that represents a manageable resource, such as an application, a service, a component, or a device

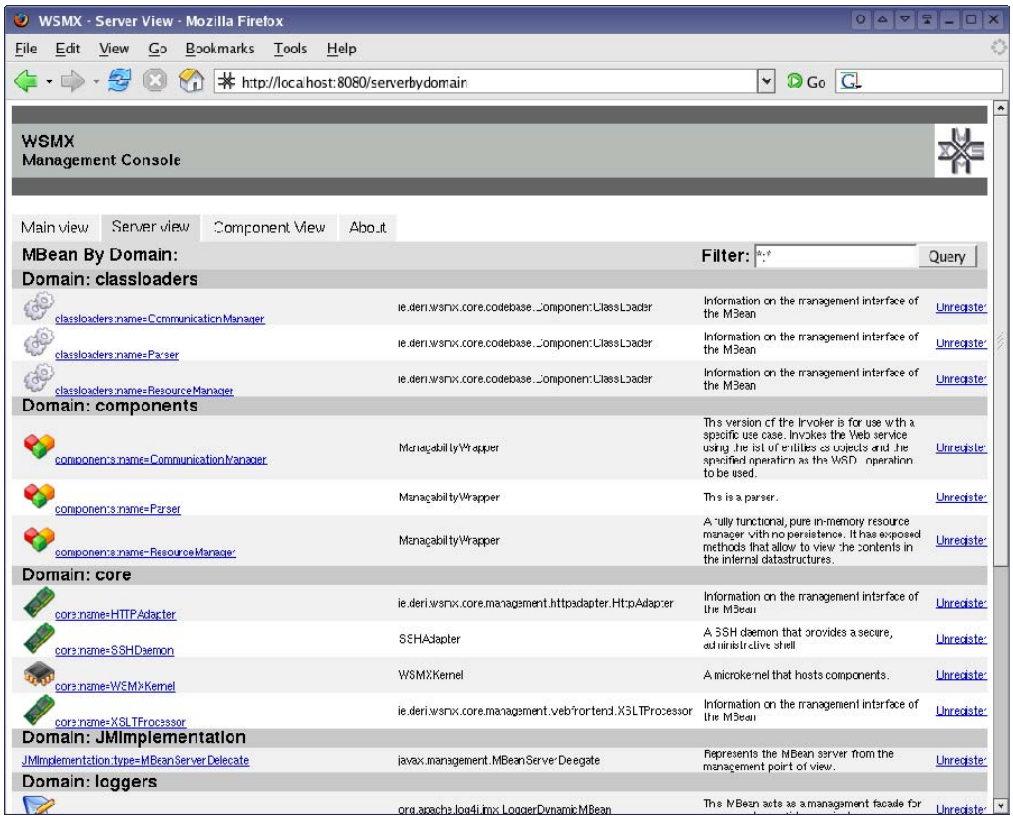

Figure 2: WSMX Management Console (GUI / HTTP) [2]

Figure 3: WSMX Management Console (TUI / SSH) [2]

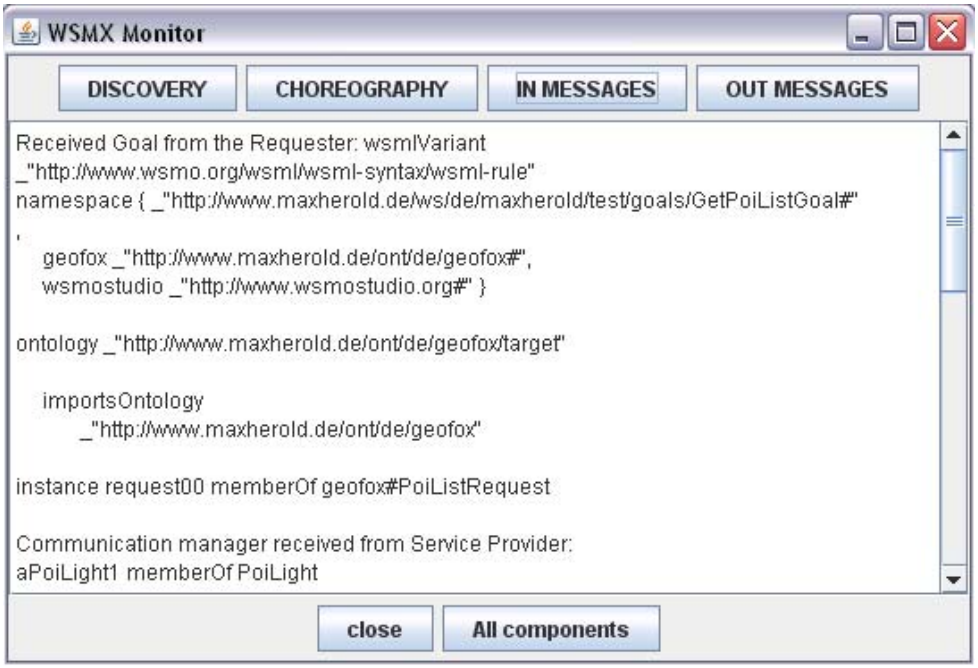

Figure 4: WSMX Monitor

2.4 Related tools

2.4.1 WSMT

The Web Service Modelling Toolkit (WSMT) provides:

- Design-time support:
  - **WSML editors** for creating and viewing WSMO artifacts (ontologies, goals, web services, mediators)
  - **WSML Discovery view, Cache view, and WSML-Reasoner** for testing web service discovery, viewing the internal WSML cache, and executing queries, respectively
  - Ontology Mapping Tool (OMT): **Mapping editor and MUnit** for creating and testing abstract mapping files (`.map`) between ontologies for data mediation
- WSMX run-time support:
  - **SEE perspective:** features a **Servers view** to connect to a WSMX instance for viewing and modifying stored WSML artifacts; and a **SOAP Message view** to send messages to the WSMX EntryPoints web service and view the results

2.4.1.1 Releases

WSMT is available at http://sourceforge.net/projects/wsmt/. Like WSMX, WSMT is under constant development, which may not be reflected in the official release cycle on sourceforge.net.

It is thus recommended to use a more recent unofficial intermediary build, which can currently be downloaded from one of the developers at http://see.deri.org/adrian/Resources/wsmt/, with more recent patches for the data mediation (mapping files) support available at http://see.deri.org/adrian/Resources/datamediator/patch/.

Note that these URLs are not officially supported, so maintenance may be paused or discontinued without notice.

Alternatively, you may check out the sources from sourceforge.net and build WSMT yourself.

2.4.1.2 Using WSMT sources

Note that detailled build instructions for WSMT are not in scope of this document. Currently, Eclipse 3.1 is required (only the plugin `org.deri.wsml.eclipse.formbasededitor` needs Eclipse 3.2). A migration to Eclipse 3.3 is planned[20].

2.4.1.3 Usage notes

Note that most of the views are context sensitive, i.e. they react to the currently opened and selected file. For instance, the MUnit view only provides the option to run a mapping test when (a) a mapping file is opened and selected and (b) there is a WSML file present containing a MUnit test case for that mapping.

Please also check the FAQ in the appendix, especially concerning WSMO4j.

2.4.2 WSMO Studio

WSMO Studio is another Eclipse-based tool that supports the modeling of WSMO entities. There is an overlap in functionality with WSMT. A feature of WSMO Studio not contained in WSMT is the BPMN editor. Since the BPMN editor is not needed in the SWS use cases for WSMX and WSMO Studio does not contain a mapping file editor like WSMT, it is recommended to use WSMT.

[20] easy way to check: if `/src/org/deri/wsmt/Application.java` in the plugin `org.deri.wsmt` still uses `org.eclipse.core.runtime.IPlatformRunnable`, the migration to Eclipse 3.3 has not yet happened

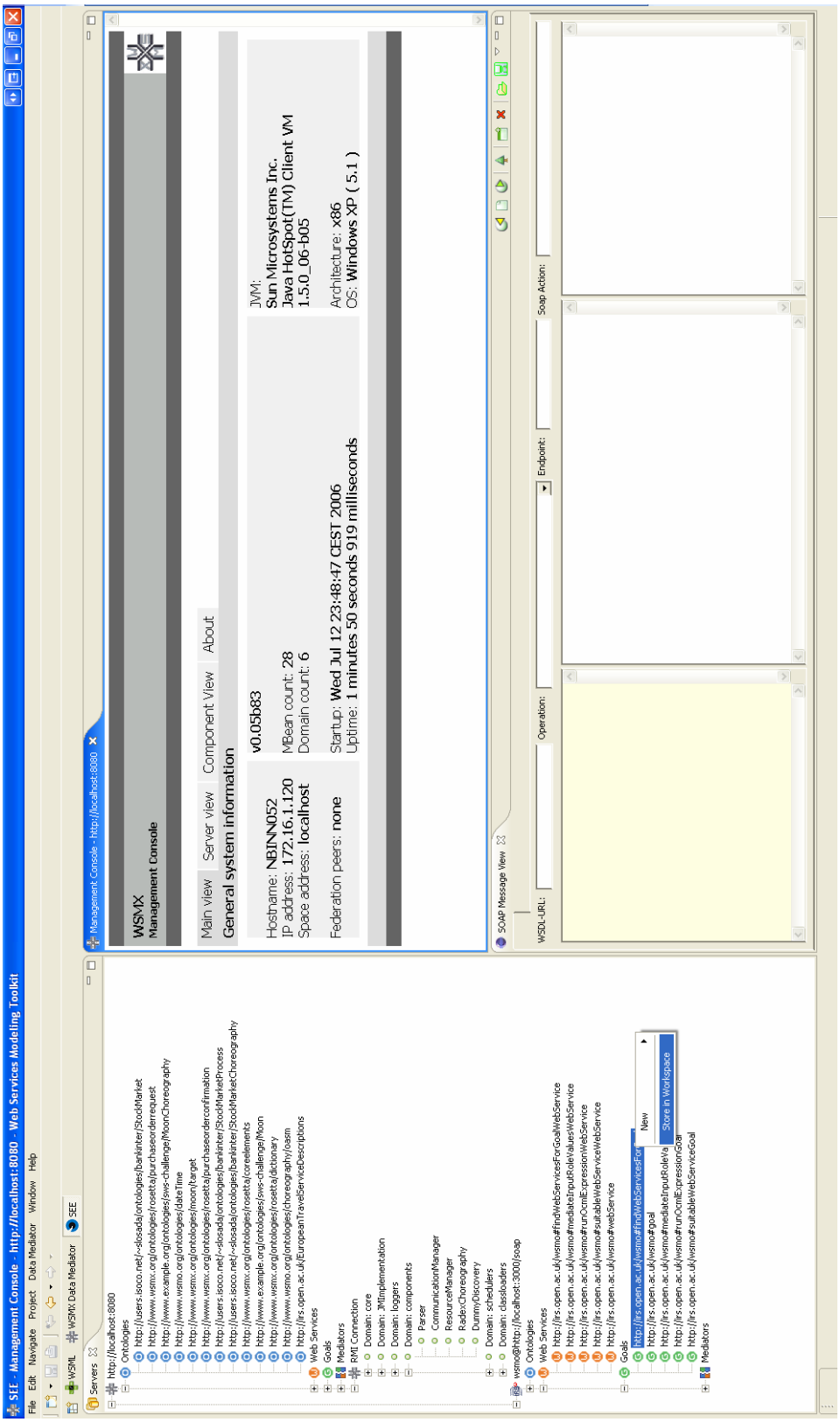

Figure 5: WSMT SEE perspective [2]

2.5 Related frameworks and libraries

2.5.1 WSMO4j

WSMO4j is an API and a reference implementation for creating and modifying WSMO artifacts. It provides an in-memory representation as well as serializers and parsers for the different WSML variants and formats.

WSMO4j is used by WSMX and other applications for processing WSML.

Please refer to the FAQ in the appendix of this document about some specific behaviour of WSMO4j (and in consequence possibly in the applications using it).

http://wsmo4j.sourceforge.net/

2.5.2 WSML2Reasoner framework and reasoning engines

The WSML2Reasoner framework is a highly modular architecture which combines various validation, normalization, and transformation functionalities essential to the translation of ontology descriptions in WSML to the appropriate syntax of several underlying reasoning engines.

The WSML2Reasoner framework is used by WSMX and other applications for reasoning purposes. The reasoning engines used by WSMX and its components include[21]:

- **IRIS** can be used for WSML-Flight, currently the fastest engine in this category
  http://iris-reasoner.org/
- **KAON2** can be used for WSML-Flight and WSML-DL
  http://kaon2.semanticweb.org/
- **MINS** can be used for WSML-Rule
  http://tools.sti-innsbruck.at/mins/
- **PELLET** can be used for WSML-DL
  http://pellet.owldl.com/

http://tools.sti-innsbruck.at/wsml2reasoner/ ,

http://sourceforge.net/projects/wsml2reasoner/

[21] they might not all be used (there are some changes under way because of the greatly improved performance of IRIS; in addition, there have been some problems with erratic behaviour of MINS in certain situations)

3 SWS DEVELOPMENT AND EXAMPLES

3.1 Creating and running SWS

3.1.1 Supported WSML variants

Currently, you can only use WSML-Flight variant if you want to use the SWS with WSMX. This is due to the fact that the IRIS reasoner is used by different components, which currently officially supports WSML-Flight only. Many examples declare WSML-Rule, which works as long as the expressivity used in the artifact remains in the bounds of the expressivity supported by the reasoner (which might already support part of WSML-Rule). Please refer to the examples.

3.1.2 Overview[22]

The following steps detail how to create a semantic web service with the current WSMX implementation.

For the sake of simplicity, it is assumed that both goal and web service use the same ontology to describe the data (i.e. no data mediation is necessary).

1. **Verify the WSDL-based web service:** Make sure the web service can be invoked as it is. Use a tool of your choice[23] to figure out what the SOAP request and response messages look like.
2. **Create the WSMO artifacts:**
    a. **Create an ontology based on the WSDL:** Get or create an ontology that defines the information that is being sent in the messages. Think of this like a database schema that defines the types of data that you might use in method invocations (e.g. customer data, mapping data...).
    b. **Create a web service** that describes the service.
    c. **Create a goal** that describes the desired goal.
    d. **Create an ontology containing input instances:** Make an instance ontology that contains the input data to go with the Goal. The `achieveGoal` entry point in WSMX takes as input a WSML document that (to keep things simple) contains both the Goal description and an ontology that contains only instances. This ontology imports the other ontology (or ontologies) used to define the data. Look at one of the goals from the SWS Challenge demo (see section 3.3.1) to see an example.
3. **Create lowering and lifting adapters:** You need to create lifting and lowering adapters that transform the ontology instances to XML for the SOAP message and the XML response back to ontology instances, respectively. For both ways code updates to the Invoker component are necessary.
    a. **Lowering adapter:** In the source code for the Invoker, look at the code for the `inlineWSMLtoXMLAdapter()` method and you'll see how its done for the other examples. Your services are going to need the same kind of adapter so, the code in that part of the Invoker will have to be extended.

[22] based on a mailing list entry by Maciej Zaremba using previous information from Matthew Moran
[23] e.g. SoapUI, see http://www.soapui.org/

b. **Lifting adapter:** This is usually done using XSLT or Ontology-XPath approach[24]. Look at how the response is handled `in syncInvoke()` in the Invoker and you'll see.

4. **Test / run it:** Use `AchieveGoalExecutionSemanticsTest` for testing, as described below; or build WSMX as a whole using Ant, start it, and access WSMX using the `achieveGoal` entry point.

3.1.3 Integration of data mediation

If goals use different ontologies than the web services, data mediation will be required. If you followed the steps described above, a goal could now be created that uses one or more ontologies unknown to the web service. Note that this only regards the exchanged messages[25]: The mediation between capability definitions of service descriptions is still an open issue, so for discovery functionality the goal must still use ontologies known to the web service in the pre- and postconditions.

Three steps are required for the integration of data mediation:

1. **Create a mapping file:** The current implementation of OO-Mediator functionality is available by means of mapping files. A mapping file (`.map`) containing abstract mappings between the source and target ontology can be created graphically using WSMT. Any imported ontologies are also considered. Like WSML artifacts, the mapping file needs to be placed in a directory where it will be loaded by WSMX (as specified in the configuration).

2. **Create an OO-Mediator** which specifies the source and target ontology for which a mapping file is available (must be the same source and target ontology as specified in the mapping file). Think of an OO-Mediator as an interface and of a corresponding mapping file as a concrete implementation[26]. You will also need to specify which of the input instances shall be considered the "root" instances for the mediation input, otherwise the system will consider all input instances as "root" instances (which will in the majority of cases lead to more result instances than expected). This is a parameter specific to the data mediator implementation and thus is specified via a NFP. The system will use all instances of the specified concept(s) as "root" instances. See Listing 5.

3. **Make sure you have importsOntology statements** on the state signature of the interface's choreography. This regards both the goal and the web service. The system will look for any registered OO-Mediators that can mediate between those ontologies during runtime.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://www.example.org/ooMediators/Rosetta_to_Moon#" }

ooMediator _"http://www.example.org/ooMediators/Rosetta_to_Moon"
  nfp
    _"http://www.wsmo.org/datamediation/rootConcepts" hasValue {
"http://www.wsmx.org/ontologies/rosetta/purchaseorderrequest#Pip3A4Purcha
seOrderRequest" }
  endnfp
  source _"http://www.wsmx.org/ontologies/rosetta/purchaseorderrequest"
```

[24] find references to examples for both approaches in section 3.3

[25] described by the choreography's state signature

[26] this way it is possible to easily integrate a different way for data mediation in parallel or instead of the currently used data mediator

```
target _"http://www.example.org/ontologies/sws-challenge/Moon"
```

Listing 5: OO-Mediator example

There are several aspects which influence the output of the data mediation (using the same mapping file and the same input instances):

- the specification which of the input instances should be considered the **root input instance(s)**:
  - **WSMX:** as specified in the OO-Mediator description, see above
  - a **MUnit** test case[27] specifies the root input instance(s) explicitly with `hasSourceInstance`
  - a **JUnit** test case (based on `DataMediatorBaseTest`) expects the root instances to contain the word `input` in their name (and any others not!)
  - the **stand-alone** Run Time Data Mediator application currently uses all the given instances as root instances
- there are **flags** that can be set in WSMT properties which also have to be considered in JUnit tests – in WSMX, these flags have corresponding properties in the configuration file
  - `TRANSFORM_ONLY_CONNECTED_INSTANCES` specifies whether to disregard any instances not connected to the root instance(s)
  - `FILTER_MAPPINGS_BASED_ON_INPUT`

You should be aware that there may only be one OO-Mediator (and thus also only one mapping file) for any given pair of source and target ontologies. Also, it is not (yet) implemented to

- explicitly specify an OO-Mediator using the `usesMediator` statement
- specify e.g. some web service as a `mediationService` in the OO-Mediator
- have sources or targets in OO-Mediators that are not ontologies

3.1.3.1 How to create a mapping file

A full discussion about how to create mappings is out of scope of this document. A reference to a document describing this process will be provided as soon as such a document is available.

Run WSMT, create a new *WSML Project*, and put the WSML files containing the source and target ontologies there (plus any necessary ontologies that are imported). Switch to the *Mapping* perspective. Create a new *Mapping* file and specify the source and target ontology identifiers in the dialog.

Opening the mapping file with the graphical editor, you will see the view of the source ontology on the left side and the view of the target ontology on the right side. You should be able to do most (or even all) of the mappings in the current "PartOf perspective" of the mapping tool, which focuses on the structure of concepts and their attributes. Select a concept or attribute from the source and target view respectively, and press the  button to create a mapping between those two. Observe the generated Concept2Concept / Concept2Attribute /

[27] see below for instructions on how to test mappings

Attribute2Concept and Attribute2Attribute mappings in the view below the main mapping window.

Creating your mappings top-down may be a good idea in many cases, i.e. start with the concept that your root input instance will be of, map it to a concept of the target ontology, and then continue with the attributes of that source concept. You can test your mapping file as described in section 3.2.2.

The *SWS Challenge: Purchase Order Mediation* example (Rosetta/Moon scenario, see section 3.3.1) uses data mediation and the corresponding mapping file might help. See Figure 6 for a schematic overview of the expected input message and the desired result after data mediation using the mapping file. If you look at the mapping file, notice that there are always mappings from the most specific attribute in the source hierarchy all the way up to the root concept (Pip3A4PurchaseOrderRequest).

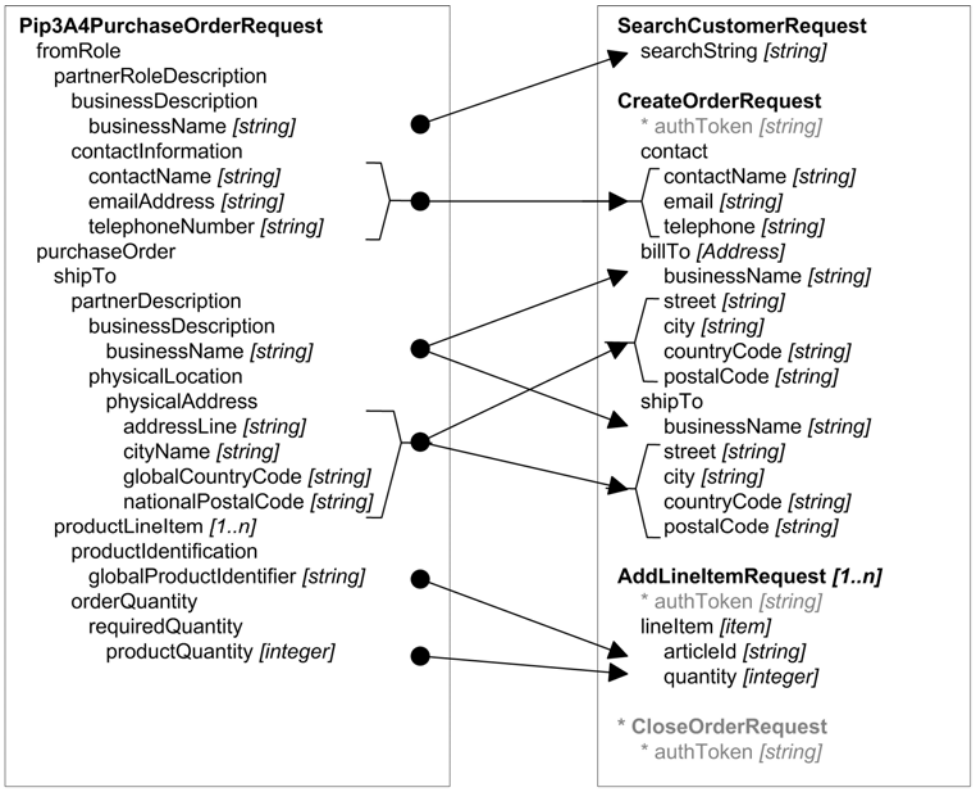

Figure 6: Mapping scenario

3.1.4 How to create a WSMO web service for WSMX

The web service should contain a **NFP** `_"http://owner"` which will later be used to select the appropriate lifting / lowering adapter. An ontology should have been created beforehand that can represent the data structures exchanged in the communication with the WSDL web service. It should be closely aligned with the XML Schema used in the WSDL so the lifting / lowering can be implemented in a straightforward way.

In the **capability**, define a postcondition that describes the service outcome. It will be used in the web service discovery process[28].

In the **interface**, define a **choreography**. The state signature should contain:

- any concept of which instances are used as input to call a WSDL operation as **`in` mode**, with **grounding** information linking it to the WSDL as described in [12][29]

[28] refer to sections 3.1.6, 4.1.9 and 4.1.10 if you want to use other than the default types of discovery

- any concept of which instances will carry the response of a WSDL call as **`out` mode**;
- any concept which satisfies both of the above as **`shared` mode**, incl. **grounding**;
- any other concepts of which instances will need to be created or modified by the choreography execution as specified in [7][30] (note that if there are concepts containing attributes with complex data types, their respective concepts will need to be explicitly declared in order to provide write access to the choreography execution; grounding information is only necessary for the top level concepts)

The **transition rules** are specified using abstract state machines (ASM). The defined rules are processed and may result in adding, updating or deleting data from the requester's or provider's processing memory. In case of add or update actions, the data may need to be obtained by service invocation. The choreography execution finishes when a provider processing step does not result in any further service invocations.

There are several possibilities where used ontologies can be imported in WSML. The current components require **`importsOntology`** statements for the used ontologies on the respective most specific level, i.e. for the choreography on the state signature element of the choreography interface (which is also used by data mediation) and for the discovery on the capability element.

3.1.4.1 Controlling the order and the end of execution of transition rules

If there is a necessity for controlling the order of execution of transition rules, one would have some internal ontology, use the *controlled* mode, and create or modify instances in the rules that can in turn be used as conditions in other rules. This works well after the first choreography step, but in order to control the beginning and end of the choreography explicitly you will need to use a specific ontology which is known to the choreography implementation. Such an ontology does exist and is listed in section 3.1.5. The usage is as follows:

1. Add `http://www.wsmo.org/ontologies/choreography/oasm#ControlState` as a *controlled* mode concept to the choreography to get read/write access to instances of this concept.
2. Create your own states ontology with your own states. Check the *value* attribute of the control state instance in each of the transition rules' conditions and remove / add the old / new state concept, respectively.
3. For the rule to be fired first, check the *value* attribute of the control state instance for `http://www.wsmo.org/ontologies/choreography/oasm#InitialState`. (A control state instance with such a *value* is created during the initialization of the choreography implementation.)
4. If, instead having the choreography execution stop like described above, you want to specify the end state explicitly, you can do that by changing the *value* attribute of the control state instance to `http://www.wsmo.org/ontologies/choreography/oasm#EndState` . (The choreography implementation checks for this state in memory and will end immediately if this state is found. This check is done after the provider update step, i.e. after executing any web service invocations triggered by the firing rule).

Several of the examples shipped with WSMX make use of these features.

---

[29] end of section 3.2.3

[30] section 2.1; diverging from the description given there, grounding information only needs to be provided for the cases explicitly mentioned above

3.1.4.2 Modifying responses from web services before the next evaluation of rules

In some cases during choreography execution, there might be the need to modify the response instances of the web service while you are still in context of the current rule execution step. E.g. in the SWING Use Case 1 example (section 3.3.4), an attribute needs to be added to the response instances (basically in order to be able to identify later from which request a specific response instance has resulted from). For this purpose, you may have an *add* statement in your transition rules which is actually performing a write operation on an instance of a concept that is declared as *out* mode (i.e. the response). If the choreography implementation finds such statements, it will execute them after the response is returned from the web service invocation and still in the context of the current rule execution step.

3.1.4.3 How to create a WSMO goal for WSMX

A goal basically looks like a web service, but less information is necessary. Look at the examples. The current implementation actually looks only at the capability part (for discovery) and at the state signature of the choreography (for determining the imported ontologies for data mediation).

3.1.5 Ontologies used by components

Choreo-graphy	**ontology _"http://www.wsmo.org/ontologies/choreography/oasm"** **concept ControlState** **value impliesType _concept** **concept InitialState** **concept EndState** **// /* Created by choreography during initialization: */** **// instance controlstate memberOf ControlState** **// value hasValue InitialState** **// /* Create your own custom states to control exec. order */** **// concept MyState1** **// concept MyState2** (see section 3.1.4.1 for a detailed explanation)

3.1.6 NFPs that configure the system's behaviour

There are several non-functional properties that configure the processing behaviour of WSMX:

Category	**NFP and description**
Lifting / lowering	**_"http://owner"** *Specified on element: webService* lifting/lowering adapters are currently selected based on this NFP; see `Invoker.java`
Web Service	*Note that only one of the NFPs in this category should be set to "true" in any given goal; if none of those is set to "true", Keyword and Lightweight discovery*

Discovery	*are used as a default.*
	_"http://www.wsmo.org/goal/discovery/rule" *Specified on element: capability (of a goal)* Lightweight Rule web service discovery is activated if this is "true". Keyword-based discovery will also be used.
	_"http://www.wsmo.org/goal/discovery/rule/extendedPlugin" *Specified on element: capability (of a goal)* Heavyweight Rule web service discovery is activated if this is "true". Keyword-based discovery will *not* be used.
Service Discovery	*Note that only one of the NFPs in this category should be set to "true" in any given goal*
	_"http://www.wsmo.org/goal/discovery/qos" *Specified on element: capability (of a goal)* QoS discovery is activated if this is "true"
	_"http://www.wsmo.org/goal/discovery/instancebased" *Specified on element: capability (of a goal)* service discovery is activated if this is "true"
	_"http://www.wsmo.org/goal/discovery/instancebased/composition" *Specified on element: capability (of a goal)* service discovery is activated if this is "true"; the discovery also regards if multiple different services can provide the requested functionality together
Service Discovery params	**_"http://www.wsmo.org/goal/discovery/instancebased/rankingcriteria"** *Specified on element: capability (of a goal)* specifies by which parameters the ranking should be performed
	_"http://www.wsmo.org/goal/discovery/instancebased/ontology" *Specified on element: capability (of a goal)* refers to the ontology of the messages related to the goal
	_"http://www.wsmo.org/goal/discovery/instancebased/mainElements" *Specified on element: capability (of a goal)* specifies the root elements that are the main parameters that should be regarded
Data Mediation	**_"http://www.wsmo.org/datamediation/rootConcepts"** *Specified on element: ooMediator* specifies which of the input instances should be regarded as root instances for data mediation (value: a list of String representations of full IRIs of concepts)

3.2 Testing

3.2.1 General

For viewing and basic testing any of the examples provided with the WSMX sources on the resource level, you can import the `/resources` directory as an *existing project* into your Eclipse or WSMT workspace and start from there.

For testing SWS (either the examples or SWS developed by youself) you should import the WSMX source folder as an *existing project* into your Eclipse workspace. Look at the JUnit test:

```
org.deri.wsmx.unittest.main.AchieveGoalExecutionSemanticsTest
```

which is located in `/unittest/src`. It mimicks the steps of the AchieveGoal execution semantics and thus simulates the behaviour of WSMX in server mode. Just add your SWS and/or comment/uncomment any lines that refer to the examples. Make sure your WSML artifacts are located in directories where WSMX can find them (see WSMX configuration). Then run as a JUnit test.

If you want to test the discovery only, the `discoverWebServices` entry point is a good starting point.

Naturally, it may also help a lot to set the logging to debug level, make use of the available monitoring options, and to use a tool of your choice to test the invocation of WSDL-based web services before using them as SWS.

3.2.2 Data Mediation

Mappings can be tested using the **MUnit** view (triggers when the mapping file is opened in the View Based Mapping Editor and there is a `.wsml` file containing a MUnit test case for this mapping). See the listing below for an example.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { munit _"wsmt://munit#" , o1 _"http://my-o1-iri/#" , ... }

ontology my_test_ontology
importsOntology { _"wsmt://munit" , _"http://my-o1-iri/" , ... }

instance my_test_suite memberOf munit#testsuite
     munit#sourceOntology hasValue myO1Reference
     munit#targetOntology hasValue myO2Reference
     munit#hasTest hasValue my_test_case

instance myO1Reference memberOf munit#ontologyreference
     munit#stringRepresentation hasValue "http://my-o1-iri/"

instance myO2Reference memberOf munit#ontologyreference
     munit#stringRepresentation hasValue "http://my-o2-iri/"

instance my_test_case memberOf munit#test
     munit#hasSourceInstance hasValue myRootInstance
     munit#hasTargetInstance hasValue { myInstance1, myInstance2 }

instance myRootInstance memberOf o1#...
```

Listing 6: MUnit test case example

If the WSMT or WSMX sources are available, mappings can also be tested using **JUnit** (see `org.deri.wsmx.mediation.tests.DataMediatorBaseTest` and subclasses).

Another option for testing is the run-time **standalone** data mediator (a SWING-based Java application). There is an ant target to run it from the sources.

3.3 Examples

There are several examples available in the WSMX binary and source distributions. They are very helpful in assessing the functionality of WSMX and also as a reference on how to write the necessary WSML artifacts. A selection of those examples is briefly introduced here.

The given JUnit test case methods refer to `AchieveGoalExecutionSemanticsTest.` Please see also the inline comments.

3.3.1 SWS Challenge: Purchase Order Mediation

Target functionality: Process and data mediation

Files in WSMX distribution: in `resources/resourcemanager`

```
./goals/
  MoonGoal.wsml
  MoonGoalWithInstances.wsml
  MoonGoal2.wsml
./webservices/
  WSMoon.wsml
  WSMoon2.wsml
./ontologies/
  dictionary.wsml                      Rosetta dictionary
  coreelements.wsml                    Rosetta core elements
  purchaseorderrequest.wsml            Rosetta purchase order request
  purchaseorderconfirmation.wsml       Rosetta purchase order confirm.
  MoonOntology.wsml                    Moon ontology
./mediators/
  OO_Rosetta_to_Moon.wsml              OO-Mediator
./datamediator/mappings/
  Rosetta_Pip3A4POR_to_Moon.map        data mediation mapping file
../communicationmanager/
  moon2WSML.xsl                        XSL for lifting Moon responses
../../mediator/src/test/files/
  Rosetta2Moon_ExpectedOutput.wsml     expected output for MUnit test
  Rosetta2Moon_MUnitTest.wsml          MUnit test
  Rosetta2Moon_TestInput.wsml          test input for MUnit test
```

Listing 7: Files for Moon example (SWS Challenge)

Use case summary: Customer “Blue” sends a RosettaNet PIP3A4 Purchase Order Request to the “Moon” manufacturer. In order for “Moon” to process the request, the message needs to be split and converted to several messages (data mediation) and sent as web service calls in a specific order to different “Moon” legacy systems (process mediation). In this process, some data returned from calls to the legacy systems is required as input for subsequent calls. The final response needs to be converted to a RosettaNet PIP3A4 Purchase Order Response (data mediation) [31].

Notes: Uses data mediator, `oasm#EndState` control state in the choreography; and XSLT for lifting.

[31] this last step is not (yet) implemented in this use case, but basically works the same as the data mediation step in the beginning

References: Full use case description and other related artifacts available at http://sws-challenge.org/wiki/index.php/Scenario:_Purchase_Order_Mediation . See also Figure 6 on p.28, which depicts the scenario for data mediation.

Running the example: Use the content of `MoonGoalWithInstances.wsml` as input to the `achieveGoal()` WSMX entry point.

JUnit Test: `testSWSChallengeMediation()`

3.3.2 SWS Challenge: Shipment Discovery

Target functionality: Web service and service discovery; simple composition

Files in WSMX distribution: in `resources/resourcemanager/sws-challenge`

```
./Goals/
  GoalA1.wsml
  ...
  GoalD1.wsml
./SWS/
  WSMuller.wsml
  WSRacer.wsml
  WSRunner.wsml
  WSWalker.wsml
  WSWeasel.wsml
./
  AdapterOntology.wsml                  used for lifting
  PurchaseOntology.wsml
  ShipmentOntology.wsml
  ShipmentOntologyInstances.wsml
  ShipmentOntologyProcess.wsml
```

Listing 8: Files for Shipment Discovery example (SWS Challenge)

Use case summary: Identify possibly relevant services for shipping taking into account several conditions (like destination, weight, price). The given services specify their functionality in different ways and granularity. In order to get the necessary data for deciding the relevance of the services, they may need to be invoked (→ service discovery, i.e. instance-based).

Notes: Uses Ontology-XPath approach for lifting.

References: Full use case description and other related artifacts available at http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery

Running the example: Use one of the given goals with the `discoverWebServices()` or `achieveGoal()` WSMX entry points.

JUnit Test: `testSWSChallengeDiscovery()`

3.3.3 SWS Challenge: Discovery II and Simple Composition

Target functionality: Web service and service discovery; simple composition

Files in WSMX distribution: in `resources/resourcemanager/sws-challenge-composition`

```
./Goals/
  GoalA1.wsml
  ...
  GoalC4.wsml
```

```
./SWSs/
  WSBargainer.wsml
  WSHawker.wsml
  WSRummage.wsml
./Ontologies/
  AdapterOntology.wsml                  used for lifting
  ProductOntology.wsml
```

Listing 9: Files for Discovery II example (SWS Challenge)

Use case summary: Find the most appropriate offer for a customer who wants to buy a computer. There are different (possibly competing) preferences, which are ranked. In addition, different parts may be aquired from different providers, while assuring compatibility. Global optimization goals and constraints are also regarded. (Like in the previous scenario, instance-based discovery is used.)

Notes: Uses Ontology-XPath approach for lifting.

References: Full use case description and other related artifacts available at http://sws-challenge.org/wiki/index.php/Scenario:_Discovery_II_and_Simple_Composition

Running the example: Use one of the given goals with the `discoverWebServices()` or `achieveGoal()` WSMX entry points.

JUnit Test: `testSWSChallengeDiscoveryWithComposition()`

3.3.4 SWING Use Case 1

Target functionality: Service discovery and composition.

Files in WSMX distribution: in `resources/resourcemanager/SWING/uc1`

```
./Goals/
  uc1goal.wsml
./SWSs/
  uc1ws.wsml
./Ontologies/
  AdminFTO.wsml                   BRGM's AdminWFS
  BrgmFTO.wsml                    BRGM's QuarryWFS
  filter.wsml                     Basic Filter Encoding, V1.0.0 (subset)
  GML.wsml                        GML 2.1.2 elements
  INSEE.wsml                      INSEE
  oasm.wsml
  SEC.wsml                        SocioEconomicConstants requ./resp.
  SupportOntology.wsml            support requests / responses
  SwingAdapter.wsml               mappings for the Swing adapter
  SwingAggregateOntology.wsml     aggregate request / response
  SwingFTO.wsml                   feature type depproductionconsumption
  WFS.wsml                        generic concepts of the OGC WFS Spec.
```

Listing 10: Files for SWING UC1 example

Use case summary: The SWING project aims at deploying SWS in the geospatial domain. In this use case, a "production-consumption" map of aggregate resources for the area around Paris is to be created.

Notes: Uses specific SwingAdapter for lifting/lowering (a NFP on each element specifies XML name in ontology). Uses `oasm#InitialState` in choreography.

References: Full use case description available in [16]. See also http://www.swing-project.org/ .

Running the example: Use the given goal with the `achieveGoal()` WSMX entry point.

JUnit Test: `testSWINGUC1()`

3.3.5 SUPER-Nexcom

Target functionality: Service selection (QoS discovery)

Files in WSMX distribution: in `resources/resourcemanager/SUPER-Nexcom`

```
./Goals/
  GoalGetVoIP1.wsml
  ...
  GoalGetVoIP4.wsml
./GoalsQoS/
  Goal-QoS1.wsml
./SWSs/
  WSProvideVoIP1.wsml
  ...
  WSProvideVoIP6.wsml
./SWSsQoS/
  WS-QoS1.wsml
./Ontologies/
  AdapterOntology.wsml                  used for lifting
  BusinessModule.wsml
  Nexcom.wsml
  QoSBase.wsml                          upper ontology for QoS
  Ranking.wsml                          ranking concepts
  VoIPQoSBase.wsml                      VoIP specific QoS ontology
```

Listing 11: Files for Nexcom example (SUPER)

Use case summary: Match VoIP requirements of customers with providers.

Notes: Uses Ontology-XPath approach for lifting. Requires a running WSMX instance due to the reliance on Web service hosted by WSMX.

References: http://www.ip-super.org/

Running the example: Use one of the given goals with the `discoverWebServices()` or `achieveGoal()` WSMX entry points.

JUnit Test: `testSUPER1stYearReviewNexcom()`

4 WSMX COMPONENTS AND DEVELOPMENT

This section provides an overview of the currently available WSMX components.

4.1 Components Overview

4.1.1 Core

Source directory:	/core	**JAR binary:**	wsmx.core
Function/Service:	microkernel; provides middelware framework functionality such as finding and loading components, handling the messaging between components, and defining paths of execution ("execution semantics"); also contains the WSMX Integration API		
Motivation:	vertical layer; integrates all other components		
Current status:	*stable;* (execution semantics may change in the future to incorporate new components / functionality)		
Contact:	Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org		
References:	[1], [2]		

4.1.2 Choreography

Source directory:	/choreography	**JAR binary:**	choreography.core
Function/Service:	resolves process heterogenity (in terms of communication mismatches) between service requester and provider		
Motivation:	there may be communication mismatches like a different order or granularity of messages between requester and provider		
Current status:	*stable*		
Contact:	Raluca Zaharia, DERI Galway, raluca.zaharia@deri.org		
References:	[6], [7]		

Choreography describes the behaviour of the service from the communication (client) perspectives (that is, how the service communicates with the client, in order to consume the functionality provided by the service) [6]. Conceptually, choreography is stateful, public process. The component resolves the communication mismatches and figures out what services to call by processing state transitions and related modes in goal (requester) and web service (provider) descriptions. A kind of conversation is initiated between those two until an end state is reached.

4.1.3 Communication Manager

Source directory:	/communicationmanager	**JAR binary:**	communicationmanager.wsmx
Function/Service:	entry point functionality to the WSMX system		

Motivation:	represents the entry point to the system for external entities that want to consume WSMX functionality
Current status:	*stable*
Contact:	Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org
References:	

4.1.4 Data Mediator

Source directory:	/mediator	**JAR binary:**	datamediator.wsmx
Function/Service:	transforms instances of the ontologies known to one of the involved parties to instances of the ontologies known to the respective other party, or vice versa, based on previously created abstract mappings between ontologies		
Motivation:	resolves data heterogenity that can appear during discovery, composition, selection or invocation of web services		
Current status:	both run-time (in WSMX and stand-alone) and design-time (as part of WSMT) are available; "*related-by"* mappings are not yet implemented		
Contact:	Adrian Mocan, STI Innsbruck, adrian.mocan@sti2.at		
References:	[14], [15]		

4.1.5 Invoker

Source directory:	/invoker	**JAR binary:**	invoker.wsmx
Function/Service:	handles communication between WSMX and external SOAP-based web services; includes lifting/lowering to/from WSML		
Motivation:	to consume functionality of a WSMO web service, the external implementation of that service must be called (currently, grounding to WSDL-based web services is supported); this also involves the conversion between WSML instances and XML-based SOAP messages in both directions		
Current status:	- generic grounding (lifting/lowering between WSML instances and XML-based SOAP messages), i.e. without the need for code changes, is being developed[32] - complete support for REST-based web services will be added		
Contact:	Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org		
References:	[17]		

4.1.6 Orchestration

Source directory:	/orchestration	**JAR binary:**	orchestration.wsmx
Function/Service:	resolves process heterogenity in terms of defining how the overall functionality of a service is achieved by the cooperation of other services		
Motivation:			

[32] see section 3.1 for current implementation details

Current status:	*Currently only used as stand-alone for tests.* In current WSMX execution semantics only the Choreography component is used - and sometimes covers functionality of the Orchestration component (e.g., by using multiple Web services instead of statefully interfacing single Web service or by manipulating data exchanged between these services).
Contact:	Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org
References:	[10]

4.1.7 Parser

Source directory:	/parser	**JAR binary:**	parser.wsmx
Function/Service:	performs syntactic validity checks of WSML documents and converts it to an in-memory representation		
Motivation:			
Current status:	This is a very light-weight component, usually other components use WSMO4j directly to parse documents when needed. The parser is used in the WSMX execution semantics, though.		
Contact:			
References:			

4.1.8 Resource Manager

Source directory:	/resourcemanager	**JAR binary:**	resourcemanager.wsmx
Function/Service:	provides storage services		
Motivation:			
Current status:	currently an in-memory implementation which uses the file system for initial loading at startup; there should be a version in the future with full persistency support (DB storage)		
Contact:	Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org		
References:			

4.1.9 Service Discovery, incl. service selection (QoS discovery)

Source directory:	/serviceDiscovery	**JAR binary:**	serviceDiscovery.wsmx
Function/Service:	**(1)** instance-based service discovery with service contracting **(2)** QoS discovery (service selection based on nonfunctional properties)		
Motivation:	**(1)** Discovery of a service based on a given goal: It may be necessary that web services need to be invoked with the actual instance data that was provided with a goal in order to determine whether a service can actually fulfill the concrete goal. *Example: A goal might contain a price criterion; a service might not expose rules to calculate the price, but provide means to retrieve it by invoking it with the concrete data.* **(2)** Further refine (rank) results according QoS criteria		

Current status:	**(1)** Service Discovery: relatively stable concerning externally visible behaviour; some more use cases and examples will be added **(2)** Service Selection (QoS Discovery): should at some time be refactored to a separate component (currently located in this component for organisational reasons)
Contact:	**(1)** Service Discovery: Maciej Zaremba, DERI Galway, maciej.zaremba@deri.org **(2)** Service Selection (QoS Discovery): Le-Hung Vu, EPFL, lehung.vu@epfl.ch
References:	**(1)** Service Discovery: [13] **(2)** Service Selection (QoS Discovery): [11] *(see Web Service Discovery component for other steps in the service discovery process)*

QoS Discovery contains a test-suite which runs with WSML-Flight reasoner. The approach to QoS-discovery is based on an upper level ontology which is inherited in domain specific ontologies. Additional functionality supporting ranking of Web services and updates of QoS parameters via user reports are also provided. The component uses Apache Derby DBMS. More details on the component can be found at:

http://lsirpeople.epfl.ch/lhvu/download/qosdisc.

4.1.10 Web Service Discovery

Source directory:	/discovery	**JAR binary:**	webServiceDiscovery.wsmx
Function/Service:	web service discovery based on web service and goal description		
Motivation:	Use a given goal definition to find web service definitions in the repository which can fulfill that goal.		
Current status:	**(1)** Keyword-based: *stable*; *enabled by default* **(2)** Lightweight (functional): *enabled by default;* is able to handle single conjunction in the logic formulas **(3)** Lightweight Rule: *can be enabled via NFP;* work in progress which will replace the previous Lightweight discovery (2) **(4)** Lightweight DL: *not integrated into WSMX* system at the moment, only used in standalone test cases **(5)** Rule (Heavyweight): *can be enabled via NFP* *(see Service Discovery component for other steps in the service discovery process)*		
Contact:	**(1)** Keyword-based: Ioan Toma, STI Innsbruck, ioan.toma@sti2.at **(2)** Lightweight (functional): Nathalie Steinmetz, STI Innsbruck, nathalie.steinmetz@sti2.at **(3)** Lightweight Rule: Nathalie Steinmetz, STI Innsbruck, nathalie.steinmetz@sti2.at **(4)** Lightweight DL: Holger Lausen, STI Innsbruck, holger.lausen@sti2.at **(5)** Rule (Heavyweight): Nathalie Steinmetz, STI Innsbruck, nathalie.steinmetz@sti2.at		
References:	*(see Service Discovery component for other steps in the service discovery process)*		

(1) Keyword-based: considers non-functional properties

(2) Lightweight (functional): uses WSML-Flight variant; considers only the postcondition part of the service capability

(3) Lightweight Rule: uses WSML-Flight/Rule variant; considers only the postcondition part of the service capability (will replace the previous Lightweight discovery (2)).

(4) Lightweight DL: Uses WSML-DL variant. A dedicated ontology is used to annotate the list of discovered Web services (i.e. type of match - set based DL, and degree of match like: exact, subsumes, plug-in, intersect). Background ontologies of both goal and web service are considered. Validation is performed on (a) expressions in goal, (b) expressions in Web service and (c) all referenced ontologies. Only valid WSML DL ontologies/expressions are accepted.

(5) Rule (Heavyweight): uses WSML-Rule variant; considers both pre- and postconditions

4.2 WSMX Development Notes

4.2.1 Useful classes

The `ie.deri.wsmx.scheduler.Environment` class (core, WSMX environment library) defines several helpful methods to access the environment. This includes access to the configuration and whether the current instance runs with the WSMX kernel (this is not the case e.g. when running a unit test from inside Eclipse).

The `ie.deri.wsmx.commons.Helper` class (core, WSMX commons library) also contains useful methods. *Note that, for production ready components, the methods related to resources storage and retrieval should not be used (other than by the Resource Manger itself): They will not work in the case where components are distributed across JVMs or physical machines. Nevertheless, they are a good starting point for development if a component needs resource access.*

4.2.2 Component deployment

Deployment of components is fairly simple: The component (packed as JAR with `.wsmx` extension) needs to be copied to the directory configured in the `wsmx.systemcodebase` option (if not configured, the same directory the core component resides in is used). This can also be done while WSMX is running.

4.2.3 Usage of components outside of WSMX

If you want to use the functionality of a component outside of WSMX, you can either:

- access the functionality via the WSMX entry point web service (which may be extended if it does not expose the required functionality); or
- access the component binary JAR (`*.wsmx`) like any other JAR as a library (note that some components might depend on Core functionalities).

5 CONCLUSION AND OUTLOOK

5.1 Current and future development efforts

The following features are currently under active development and may well be completed before this document is updated:

- support for REST-based web services
- generic grounding of WSMO web services to WSDL (in order to transform WSML instances to XML and back without the need for XSLT and hard-coded transformations), including generation of ontologies from WSDL
- usage of SAWSDL for service grounding

Other issues that might be addressed in the future include:

- Security
- Error handling (review error handling in components, WSDL fault handling).

See also to the *current status* descriptions of the individual components in section 4.1.

5.2 Acknowledgements

This work is supported by the Science Foundation Ireland Grant No. SFI/02/CE1/I131 and the EU project SUPER (FP6-026850).

REFERENCES

[1] Vitvar, T. et al. (2007): *Semantically-enabled service oriented architecture: concepts, technology and application.* In: *Service Oriented Computing and Applications*, Vol. 1 Number 2 / June 2007, pp. 129-154. Springer, London.
http://www.vitvar.com/!publications/SOCA2007-VitvarMKZZMCHF.pdf

[2] Moran, M.; Schreder, B.; Haselwanter, T.; Zaremba, M.; Sapkota, B.; Wahler, A. (2006): D6.17: *Architecture Prototype V4*. DIP Project, WP6 Interoperability and Architecture.

[3] Roman, D.; Lausen, H.; Keller, U. (eds.) (2006): *D2V1.3: Web Service Modeling Ontology (WSMO).* WSMO Final Draft.
http://www.wsmo.org/TR/d2/v1.3/

[4] de Bruijn, J. (ed.) (2005): *D16.1v0.21 : The Web Service Modeling Language WSML*. WSML Final Draft.
http://www.wsmo.org/TR/d16/d16.1/v0.21/

[5] Zaremba, M.; Moran, M.; Haselwanter, T.: *D13.4v0.2 WSMX Architecture.* WSMX Final Draft.
http://www.wsmo.org/TR/d13/d13.4/v0.2/

[6] Aiken, D.; Zaremba, M. (2005): *D22.0 v0.2 WSMX Documentation.* WSMO Working Draft.
http://www.wsmo.org/TR/d22/v0.2/

[7] Roman, D. (ed.) ; Scicluna, J. (ed.) ; Nitzsche, J. (ed.); Fensel, D.; Polleres, A.; de Bruijn, J. (2007): *D14v1.0 Ontology-based Choreography.* WSMO Final Draft.
http://www.wsmo.org/TR/d14/v1.0/

[8] Cimpian, E.; Mocan, A.; Scicluna, J.: *D13.7 v0.1: Process Mediation in WSMX*. WSMX Working Draft.
http://www.wsmo.org/TR/d13/d13.7/v0.1/

[9] Feier, C.; Domingue, J. (2005): *D3.1v0.1 WSMO Primer*. WSMO Final Draft.
http://www.wsmo.org/TR/d3/d3.1/v0.1/

[10] Roman, D. (ed.); Scicluna, J. (ed.); Fensel, D.; Haselwanter, T. (2007): *D15v0.1 Orchestration in WSMO*. WSMO Final Draft.
http://www.wsmo.org/2005/d15/v0.1/

[11] Hauswirth, M.; Porto, F.; Vu, L.-H. (2006): *D4.17 P2P & QoS-enabled Service Discovery Specification.* DIP Project Deliverable.
http://dip.semanticweb.org/documents/D4.17-Revised.pdf

[12] Kopecký, J.; Moran, M.; Vitvar, T.; Roman, D.; Mocan, A.: *D24.2v0.1 WSMO Grounding*. WSMO Working Draft.
http://www.wsmo.org/TR/d24/d24.2/v0.1/

[13] Zaremba, M.; Vitvar, T.; Moran, M.; Haselwanter, T. (2006): *WSMX Discovery for SWS Challenge*. In: *Proceedings of the Semantic Web Services Challenge - Phase III Workshop*, Workshop at 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia, USA.
http://see.deri.ie/maciej/papers/Workshop/SWS-Challenge2006-phase3-Discovery.pdf

[14] Mocan, A.; Cimpian, E. (2007): *An Ontology-Based Data Mediation Framework for Semantic Environments.* In: *International Journal on Semantic Web and Information Systems (IJSWIS)*, 3(2), April - June 2007

[15] Mocan, A.; Cimpian, E.; Kerrigan, M. (2006): *Formal Model for Ontology Mapping Creation.* In: *Proceedings of the 5th International Semantic Web Conference (ISWC-2006)*, pp. 459-472, Athens, Georgia, USA, November 2006. Springer-Verlag.
http://iswc2006.semanticweb.org/items/Mocan2006oq.pdf

[16] BRGM Bureau de Recherches Géologiques et Minières (2007): *D1.1 Use Case Definition and I&T Requirements.* SWING Project Deliverable.
http://www.swing-project.org/deliverables/document/76

[17] Kopecký, J.; Moran, M.; Vitvar, T.; Roman, D.; Mocan, A. (2007): *D24.2v0.1. WSMO Grounding.* WSMO Working Draft.
http://wsmo.org/TR/d24/d24.2/v0.1/

APPENDIX A: FAQ

This Appendix tries to give answers to the most common questions which were not incorporated in the main document. It is mainly based on questions asked on various mailing lists.

If you have any open questions, please check whether you might find the answer in the main document. If not, check the mailing lists for any more recent additions. If you can't find an answer, send a mail to the appropriate mailing list. Of course, you may also try to get a hold of the responsible component developer directly.

WSMO/WSML modeling

What is the difference between Logic Programming (LP) and Description Logics (DL)?

Logic Programming is one of the main logical formalism for Semantic web apart from Description Logics. They have different semantics and syntax and major difference is in closed world assumption taken in LP (i.e., things we don't know are assumed to be false) versus open world assumption in DL (i.e., things which are not stated to be neither true or false are assumed to be unknown). Some reading material on comparison of Logic Programming with DL in context of WSML can be found at:

http://www.wsmo.org/TR/d16/d16.1/v0.21/

http://www2005.org/cdrom/docs/p623.pdf

How to model WSML web service when there are multiple operations, each with a different set of preconditions?

In principle WSMO considers single service capability, if you have a Web service which has a number of operations you need to either consider them as a separate services or chain WSDL operations in Choreography part providing a high-level capability for the overall functionality offered in the Choreography.

Another possibility is more bottom-up approach using SAWSDL where you can attach WSMO Capability to WSDL operations. We however do not yet have any run-through WSMX example based on SAWSDL. However, the functionality offered by existing WSMX components could be used in SAWSDL case without any serious changes. *(Maciej Zaremba, Feb. 2008)*

How do I check the consistency of an ontology with instances, i.e. that they do not violate any of the defined axioms?

Go to the reasoning view of WSMT and enter the query:

```
?x memberOf ?y
```

If your ontology is okay it will execute the query, if not then you will receive a pop-up dialog with the consistency violation. *(Mick Kerrigan, Sep. 2007)*

WSMX

What does it mean if a component is 'blacklisted'?

It is related to WSMX development and some small inconsistencies between WSMX component manifest and actual classes packed inside. *(Maciej Zaremba, Oct. 2006)*

I get the following exception: org.wsml.reasoner.ExternalToolException: The ontology … is not registered.

It's a KAON2 bug and not harmful in any way. (This exception is thrown when you first register an ontology with the KAON2 reasoner. Unfortunately, it's thrown internally and cannot be intercepted.) *(Maciej Zaremba, Dec. 2007)*

How can I run the Amazon E-commerce service on WSMX? Where can I find a description of how to do that?

To the best of my knowledge this particular use case was never executed in it's entirety or updated to the latest language specification which WSMX is based on. *(Thomas Haselwanter, June 2006)*

WSMO4j and Other

Where can I find some basic WSMO4j examples?

See http://wsmo4j.sourceforge.net/examples.html which features basic examples of handling ontologies, web services, locators, SAWSDL, and parsing with WSMO4j.

What is the difference between orchestration and choreography? Why is orchestration not used in the AchieveGoal execution semantics?

WSMO Choreography is used to model stateful Web service expressed in ontologized ASM in to some extend similar way like WSMO Orchestration. There are some differences in terms of the syntax and semantics between Choreography and Orchestration, e.g., in WSMO Choreography Web Service invocations are implicit and can be figured out from the state transitions and related modes, while Orchestration is more explicit with Service invocation by using InvokeWebService orchestration construct and allowing to explicitly invoke WSMO Goals. In most of the WSMX use-cases it sufficed to use Choreography (sometimes slightly abused, closed to what Orchestration is supposed to do), while Orchestration was used more as a proof-of-concept in the standalone test-cases.

There is no correlation in the implementation between Choreography part and Orchestration. They are used separately, but it would definitely make a lot of sense to add some coordination between them and to use Orchestration component in WSMX Execution Semantics. Conceptually, Orchestration is something more like private process, while Choreography is stateful, public process. Currently, in execution semantics we only use Choreography which sometimes covers functionality of the Orchestration component (e.g., by using multiple Web services instead of statefully interfacing single Web service or by manipulating data exchanged between these services). *(Maciej Zaremba, Aug. 2007)*

Why are there doubled or even tripled postconditions (when storing a WSML entity in WSMX multiple times // OR // when using SEE→ store in WSMT and then want to view it again) ? Is there a solution?

This happens because postconditions are not named (anonymous id). Since WSMT does not clean the model before parsing it the old postconditons are still around. *(Holger Lausen, Oct. 2006)*

Solution: Using WSMX, just delete the entity before you store it again. For this problem in WSMT, there is currently no solution.

See also the next question for more details.

WSMO4j merges newly parsed ontologies with the old ones. Why? Is there a way to prevent that?

WSML parser has a static registry for WSMO entities in order to accumulate a total model as per open-world assumption. If you do not want the parsed specifications to accumulate, you pass a configuration to the parser that clears the previously parsed entities. You can do it this way:

```
public class CleanModelParserExample {

    public static void main(String[] args) throws IOException,
ParserException, InvalidModelException {
        Parser parser01 = Factory.createParser(null);
        Serializer serializer = Factory.createSerializer(null);

        InputStream is =
CleanModelParserExample.class.getResourceAsStream("/Ontology01.wsml");
        InputStreamReader isr = new InputStreamReader(is);
        TopEntity[]    topEntities = parser01.parse(isr);
        Ontology ontology01 = (Ontology) topEntities[0];

        serializer.serialize(new TopEntity[]{ontology01}, new
OutputStreamWriter(System.out));

       Map<String,Object> props = new HashMap<String, Object>();
       props.put(Parser.CLEAR_MODEL, true);
       Parser parser02 = Factory.createParser(props);
        is =
CleanModelParserExample.class.getResourceAsStream("/Ontology02.wsml");
        isr = new InputStreamReader(is);

        topEntities = parser02.parse(isr);
        Ontology ontology02 = (Ontology) topEntities[0];
        serializer.serialize(new TopEntity[]{ontology02}, new
OutputStreamWriter(System.out));
    }

}
```

Note: The `CLEAR_MODEL` solution solves the issue of sequentially loading the same file twice, but does not resolve the particular problem if you want to have two versions of the same ontology in memory at the same time as one another. There has been discussion about a "workspaces" model in WSMO4j for a long time. This issue will most likely be addressed in the near future.

(Mihail Konstantinov, Ontotext, and Mick Kerrigan, STI Innsbruck, Feb. 2008)

How do I convert between WSML syntaxes (WSML, WSML XML, WSML RDF)? And to OWL?

WSMO4j contains parsers and serializers for all of the mentioned. Use the code below as a starting point.

```
package test;

import java.io.FileReader;
import java.io.StringWriter;
import java.util.HashMap;
import java.util.Map;

import org.wsmo.common.TopEntity;
import org.wsmo.factory.DataFactory;
import org.wsmo.factory.Factory;
import org.wsmo.factory.LogicalExpressionFactory;
import org.wsmo.factory.WsmoFactory;
```

```
import org.wsmo.wsml.Parser;
import org.wsmo.wsml.Serializer;

import com.ontotext.WSMO4j.serializer.xml.WsmlXmlSerializer;

public class WSMOXMLTest {

     public static void main(String[] args) {
           Map props = new HashMap();

           // use default implementation for factory
           WsmoFactory factory = Factory.createWsmoFactory(null);
           LogicalExpressionFactory leFactory = Factory
                       .createLogicalExpressionFactory(null);
           DataFactory dataFactory = Factory.createDataFactory(null);

           props.put(Factory.WSMO_FACTORY, factory);
           props.put(Factory.LE_FACTORY, leFactory);
           props.put(Factory.DATA_FACTORY, dataFactory);

           Parser parser = Factory.createParser(props);

           props = new HashMap();
           props.put(Factory.PROVIDER_CLASS,
WsmlXmlSerializer.class.getName());
           Serializer serializer = Factory.createSerializer(props);

           try {
                 TopEntity[] entities = parser.parse(new
 FileReader("the-simpsons-ontology.wsml"));

                 StringWriter writer = new StringWriter();
                 serializer.serialize(entities, writer);
                 System.out.println(writer.getBuffer().toString());


           } catch (Exception e) {
                 e.printStackTrace();
           }
     }
}
```

(Mick Kerrigan, STI Innsbruck; Code by Tammo van Lessen, Uni Stuttgart; 2008)

How do I get the contents of a Choreography from WSMO4j? There are two different Choreography interfaces.

Due to some API changes, there are currently two different Choreography interfaces: `org.wsmo.service.Choreography` and `org.wsmo.service.choreography.Choreography`. The latter one actually has methods to access the contents of a choreography (i.e. rules and state signature), but the API usually returns the first. Since the actual object returned by the implementation implements both interfaces, you can just cast to `org.wsmo.service.choreography.Choreography` if you need to.