DIP
Data, Information and Process Integration with Semantic Web Services
FP6 – 507483

Deliverable

WP 4: Service Usage
D4.4
WSMO Studio Specification

Marin Dimitrov
Alex Simov
Marc Richardson

Mar 11th, 2005
SUMMARY

This deliverable presents the specification of an Integrated Service Environment for the Semantic Web Service domain, called WSMO Studio. The studio is based upon the Web Services Modelling Ontology and is concerned with the generation, description, management and use of the WSMO elements – Ontologies, Web Services, Mediators and Goals. The aim of WSMO studio is also to provide a framework for the easy integration of 3rd party components. This will improve its usability and extensibility. The deliverable first focuses on existing approaches for tools of this nature and considers their relative merits. Limitations of the current tools are also discussed with reference to how WSMO studio aims to address some of these. A detailed list of the functional and non-functional requirements for the tool is then provided. Finally a specification for WSMO Studio is presented based on the Eclipse plug-in architecture.

The Target audience for the specification of WSMO Studio is the component providers who aim to provide plug-ins, and also the partners contributing to the development of WSMO studio in deliverables D4b.5 and D4b.11.

Disclaimer: The DIP Consortium is proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>FP6 – 507483</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acronym</td>
<td>DIP</td>
</tr>
<tr>
<td>Full title</td>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
</tr>
<tr>
<td>Project URL</td>
<td><a href="http://dip.semanticweb.org">http://dip.semanticweb.org</a></td>
</tr>
<tr>
<td>Document URL</td>
<td></td>
</tr>
<tr>
<td>EU Project officer</td>
<td>Brian Macklin</td>
</tr>
<tr>
<td>Deliverable Number</td>
<td>4.4</td>
</tr>
<tr>
<td>Title</td>
<td>WSMO Studio Specification</td>
</tr>
<tr>
<td>Work package Number</td>
<td>4</td>
</tr>
<tr>
<td>Title</td>
<td>Service Usage</td>
</tr>
<tr>
<td>Date of delivery</td>
<td>Contractual M 12 Actual 20-Feb-04</td>
</tr>
<tr>
<td>Status</td>
<td>version. 0.6 final</td>
</tr>
<tr>
<td>Nature</td>
<td>Prototype ☑ Report ☐ Dissemination ☑</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public ☑ Consortium ☑</td>
</tr>
<tr>
<td>Authors (Partner)</td>
<td>Marin Dimitrov (Sirma), Alex Simov (Sirma), Marc Richardson (BT)</td>
</tr>
<tr>
<td>Responsible Author</td>
<td>Marc Richardson</td>
</tr>
<tr>
<td>Partner</td>
<td>BT</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:marc.richardson@bt.com">marc.richardson@bt.com</a></td>
</tr>
<tr>
<td>Phone</td>
<td>+441473609589</td>
</tr>
<tr>
<td>Abstract (for dissemination)</td>
<td></td>
</tr>
<tr>
<td>Keywords</td>
<td></td>
</tr>
<tr>
<td>Version Log</td>
<td></td>
</tr>
<tr>
<td>Issue Date</td>
<td>Rev No.</td>
</tr>
<tr>
<td>18-2-2005</td>
<td>001</td>
</tr>
<tr>
<td>21-2-2005</td>
<td>002</td>
</tr>
<tr>
<td>Date</td>
<td>Code</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>08-3-2005</td>
<td>003</td>
</tr>
<tr>
<td>11-3-2005</td>
<td>004</td>
</tr>
<tr>
<td>28-3-2005</td>
<td>005</td>
</tr>
<tr>
<td>29-3-2005</td>
<td>006</td>
</tr>
<tr>
<td>30-3-2005</td>
<td>007</td>
</tr>
</tbody>
</table>
## Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Ireland Galway        | NUIG    | Prof. Dr. Christoph Bussler  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
Email: chris.bussler@deri.org  
Tel: +353 91 512460 |
| Fundacion De La Innovacion.Bankinter         | Bankinter | Monica Martinez Montes  
Fundacion de la Innovation. Bankinter  
Paseo Castellana, 29  
28046 Madrid,  
Spain  
Email: mmmnez@bankinter.es  
Tel: 916234238 |
| Berlecon Research GmbH                       | Berlecon | Dr. Thorsten Wichmann  
Berlecon Research GmbH  
Oranienburger Str. 32  
10117 Berlin,  
Germany  
Email: tw@berlecon.de  
Tel: +49 30 2852960 |
| British Telecommunications Plc.              | BT      | Dr John Davies  
BT Exact (Orion Floor 5 pp12)  
Adastral Park Martlesham  
Ipswich IP5 3RE,  
United Kingdom  
Email: john.nj.davies@bt.com  
Tel: +44 1473 609583 |
| Swiss Federal Institute of Technology, Lausanne | EPFL   | Prof. Karl Aberer  
Distributed Information Systems Laboratory  
École Polytechnique Fédérale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
Email: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council                         | Essex   | Mary Rowiatt,  
Essex County Council  
PO Box 11, County Hall, Duke Street  
Chelmsford, Essex, CM1 1LX  
United Kingdom.  
Email: mary@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik                 | FZI     | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe  
Germany  
Email: abecker@fzi.de  
Tel: +49 721 9654 0 |
Institut für Informatik, Leopold-Franzens Universität Innsbruck

UIBK

Prof. Dieter Fensel
Institute of computer science
University of Innsbruck
Technikerstr. 25
A-6020 Innsbruck, Austria
Email: dieter.fensel@deri.org
Tel: +43 512 5076485

ILOG SA

ILOG

Christian de Sainte Marie
9 Rue de Verdun, 94253 Gentilly, France
Email: csma@ilog.fr
Tel: +33 1 49082981

inubit AG

inubit

Torsten Schmale
inubit AG
Lützowstraße 105-106
D-10785 Berlin
Germany
Email: ts@inubit.com
Tel: +49 30726112 0

Intelligent Software Components, S.A.

iSOCO

Dr. V. Richard Benjamins, Director R&D
Intelligent Software Components, S.A.
Pedro de Valdivia 10
28006 Madrid, Spain
Email: rtbenjamins@isoco.com
Tel. +34 913 349 797

The Open University

OU

Dr. John Domingue
Knowledge Media Institute
The Open University, Walton Hall
Milton Keynes, MK7 6AA
United Kingdom
Email: jb.domingue@open.ac.uk
Tel.: +44 1908 655014

SAP AG

SAP

Dr. Elmar Dorner
SAP Research, CEC Karlsruhe
SAP AG
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
Email: elmar.dorner@sap.com
Tel: +49 721 6902 31

Sirma AI Ltd.

Sirma

Atanas Kiyakov,
Ontotext Lab., - Sirma AI EAD
Office Express IT Centre, 3rd Floor
135 Tzarigradsko Chausse
Sofia 1784, Bulgaria
Email: atanas.kiyakov@eirma.bg
Tel.: +359 2 9768 303

Tiscali Österreich GmbH

Tiscali

Dieter Haacker
Tiscali Österreich GmbH.
Diefenbachgasse 35
A-1150 Vienna
Austria
Email: Dieter.Haacker@at.tiscali.com
<table>
<thead>
<tr>
<th>Unicorn Solution Ltd.</th>
<th>Unicorn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tel:</strong> +43 1 899 33 160</td>
<td></td>
</tr>
<tr>
<td><strong>Jeff Eisenberg</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Unicorn Solutions Ltd,</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Malcha Technology Park 1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Jerusalem 96951</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Israel</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Email:</strong> <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a></td>
<td></td>
</tr>
<tr>
<td><strong>Tel.:</strong> +972 2 6491111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vrije Universiteit Brussel</th>
<th>VUB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Carlo Wouters</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Starlab- VUB</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Vrije Universiteit Brussel</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Pleinlaan 2, G-10</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1050 Brussel ,Belgium</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Email:</strong> <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a></td>
<td></td>
</tr>
<tr>
<td><strong>Tel.:</strong> +32 (0) 2 629 3719</td>
<td></td>
</tr>
</tbody>
</table>
## List of Key Words/Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EPL</td>
<td>Eclipse Public Licence</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ISE</td>
<td>Integrated Service Environment</td>
</tr>
<tr>
<td>J2SE</td>
<td>Java 2 Platform Standard Edition</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSGi</td>
<td>Open Services Gateway Initiative</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PSM</td>
<td>Problem Solving Methods</td>
</tr>
<tr>
<td>LGPL</td>
<td>GNU Lesser General Public Licence</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit</td>
</tr>
<tr>
<td>SWS</td>
<td>Semantic Web Services</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Discovery, Description and Integration</td>
</tr>
<tr>
<td>UPML</td>
<td>Universal Problem-solving Modeling Language</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>WSFL</td>
<td>Web Services Flow Language</td>
</tr>
<tr>
<td>WSML</td>
<td>Web Service Modeling Language</td>
</tr>
<tr>
<td>WSMO</td>
<td>Web Service Modeling Ontology</td>
</tr>
<tr>
<td>WSMX</td>
<td>Web Service Execution Environment</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUMMARY</td>
<td>2</td>
</tr>
<tr>
<td>LIST OF KEY WORDS/ABBREVIATIONS</td>
<td>8</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>9</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>2 STATE OF THE ART</td>
<td>11</td>
</tr>
<tr>
<td>2.1 IRS III</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 Architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 User Interface</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 Key Features</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4 Summary</td>
<td>14</td>
</tr>
<tr>
<td>2.2 SWWS Studio</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2 User Interface</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3 Key Features</td>
<td>16</td>
</tr>
<tr>
<td>2.2.4 Summary</td>
<td>16</td>
</tr>
<tr>
<td>2.3 ODE SWS</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Architecture</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 User Interface</td>
<td>18</td>
</tr>
<tr>
<td>2.3.3 Key Features</td>
<td>19</td>
</tr>
<tr>
<td>2.3.4 Summary</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Overview and Comparison</td>
<td>20</td>
</tr>
<tr>
<td>3 REQUIREMENTS</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Functional requirements</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Non-functional requirements</td>
<td>22</td>
</tr>
<tr>
<td>4 WSMO STUDIO GOALS</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Shortcomings of existing tools</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Requirements resolution in WSMO Studio</td>
<td>24</td>
</tr>
<tr>
<td>4.2.1 Functional requirements</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2 Non-functional requirements</td>
<td>25</td>
</tr>
<tr>
<td>5 ECLIPSE PLATFORM</td>
<td>26</td>
</tr>
<tr>
<td>5.1 Plug-in based architecture</td>
<td>26</td>
</tr>
<tr>
<td>5.1.1 Runtime kernel</td>
<td>26</td>
</tr>
<tr>
<td>5.1.2 Plug-ins</td>
<td>27</td>
</tr>
</tbody>
</table>
5.2 Workspace ............................................................................................................ 29
5.3 User Interface ....................................................................................................... 30

6 WSMO STUDIO SPECIFICATION ................................................................................. 31
   6.1 High-level architecture ......................................................................................... 31
   6.2 Components .......................................................................................................... 32
      6.2.1 Ontology manager ......................................................................................... 32
      6.2.2 WSMO editor ................................................................................................ 34
      6.2.3 Repository browser ........................................................................................ 35
   6.3 Specification refinement ....................................................................................... 35
      6.3.1 Product mapping ............................................................................................ 35
      6.3.2 Plug-ins .......................................................................................................... 36
      6.3.3 Extension points ............................................................................................ 36
      6.3.4 Perspectives ................................................................................................... 37
      6.3.5 Views and Editors ......................................................................................... 38
      6.3.6 GUI guideline ................................................................................................ 41

7 CONCLUSION ............................................................................................................... 41

REFERENCES .................................................................................................................. 42

LIST OF FIGURES
   Figure 1. IRS III Architecture ...................................................................................... 12
   Figure 2. Defining a SWS in IRS-3 ................................................................................ 13
   Figure 3. Defining a Goal in SWWS Studio ................................................................. 15
   Figure 4. The ODE-SWS Architecture ......................................................................... 17
   Figure 5. Defining a Service in ODE SWS ................................................................. 19
   Figure 6 Eclipse plug-in architecture .......................................................................... 28
   Figure 7 Eclipse Workbench (as appears in [9]) ........................................................ 31
   Figure 8 WSMO Studio architecture .......................................................................... 32
   Figure 9 WSMO Studio within the Eclipse platform .................................................. 36
   Figure 10 WSMO Studio - Ontology perspective ...................................................... 40
   Figure 11 WSMO Studio – WSMO perspective ........................................................... 41

LIST OF TABLES
   Table 1: WSMO Studio perspectives ........................................................................... 38
1 INTRODUCTION

This deliverable is concerned with the specification of a Semantic Web Services Browser. The remit of the Browser is broad enough to merit the use of a more appropriate name. The tool developed will be known as the WSMO Studio since it will be based upon the Web Services Modelling Ontology and is concerned with the generation, description, management and use of the WSMO elements – Ontologies, Web Services, Mediators and Goals.

The role of Workpackage 4 in DIP is to create a set of tools / modules for using and exploiting semantically described Web Services. The deliverable will first focus on existing approaches for tools of this nature and consider their relative merits. Following this a detailed presentation of the functional requirements is given. This focuses on requirements related to the WSMO elements and to management and general issues. Since the extensibility of the tool is a key requirement, architectural issues are then examined in detail with a description of existing approaches and a selection of the most appropriate of these for the Studio. Following this, a specification of the Studio is presented which describes how the requirements will be satisfied.

2 STATE OF THE ART

There are a number of tools currently available relating to SWS, including tools to describe, compose, invoke and discover web services as well as many tools related to ontology creation and management. This section will present some of the tools which have most in common with the aims of WSMO studio and discusses their relative merits. A more comprehensive state of the art is provided in [12].

2.1 IRS III

The Internet Reasoning Service (IRS) supports the provision of semantic reasoning services within the context of the Semantic Web. It allows the publication, composition, execution of different and heterogeneous web services. IRS-III is based on WSMO [29] and the UPML [25] (Unified Problem Solving Method Development Language) framework developed within the IBROW[34] (Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web) project. The UPML framework partitions knowledge in the system into several components: ontologies, domain models, task models, and problem solving methods (PSMs) These components are connected via bridges. The UPML framework has been extended to allow support for WSMO by including goal models, mediator models and web service models.
2.1.1 Architecture

The IRS-III architecture is composed by the three main components: the IRS-III Server, the IRS-III Publisher and the IRS-III Client. As shown in Figure 1.

![Figure 1. IRS III Architecture](image)

The IRS-III Server is based on a Lisp HTTP server which has been extended to include a SOAP handler to send and receive messages within the system. A concrete web service can be published with IRS-III by associating it with a WSMO web service description.

2.1.2 User Interface

The IRS-III browser provides an interface to support the creation of WSMO classes, link the WSMO descriptions to concrete web services and then to invoke the services.

- Creating WSMO descriptions - The user can select the properties button and bring up a dialog for defining the non functional properties. The other properties associated with a web service are contained in four panels: Inputs and Output, Capability, Interface and Used Mediators. A roles panel is also provided with ‘Add Input’ and ‘Delete Input’ buttons that can be used to add and delete input roles, as shown in Figure 2.
• **Discovery/Invocation** - IRS-III contains a simple goal discovery mechanism. A user can find goals whose input or output roles have a type which match a defined filter. The IRS discovery mechanism is based on Goal-WebService (GW) mediators which is not a standard mediator as defined by WSMO. This is because IRS-III supports capability driven web service invocation and bases discovery on GW-mediators and assumptions. Goals are in this context are defined to have inputs and outputs meaning they can be invoked directly (which is not the case in WSMO 1.0). When a goal is invoked the IRS broker creates a set of possible contender web services using the gw-mediators. A specific web service out of the possible web services is then selected using an applicability function within the assumption slot of the web service’s associated capability.

![Figure 2. Defining a SWS in IRS-3](image)

2.1.3 **Key Features**

- **Ease of publishing** – By filling in a simple form, a piece of standalone Java or Lisp code, a web application accessible through a browser or a web service can be attached to a semantic description.

- **Open** - Parts of the IRS-III brokering mechanism have been implemented as semantic web services. IRS-III users are free to publish their own services to implement these components if they desire.

- **Capability based invocation** – Web services can be invoked by asking for a WSMO goal to be solved.

- **OWL-S support** – IRS-III provides support to import OWL-S descriptions.
2.1.4 Summary
IRIS-III is a good example of a SWS platform supporting WSMO and provides the ability to define WSMO ontologies and descriptions. It supports the import of OWL-S descriptions but at this stage does not support WSML[8]. It implements a simple yet effective discovery mechanism. However this discovery mechanism may not be suited to all SWS applications and there are scenarios where a goal based invocation model may not work.

2.2 SWWS Studio
The SWWS Studio1 is the client side integrated development environment for modelling Semantic Web Service descriptions. It was developed as part of the EU 5th Framework Project Semantically Enabled Web Services2 (SWWS). It provides a User Interface for defining and managing WSMO Services, Ontologies, Goals and Mediators.

2.2.1 Architecture
SWWS Studio is comprised of two components: a graphical designer for modelling WSML specifications (Service Editor) and a graphical tool for modelling compositions of services (Service Composer). As the tool is proprietary and not intended for further development outside of the SWWS project there is no detailed information about the architecture.

2.2.2 User Interface

Service Editor
The Service editor provide support for describing and managing the four elements associated with WSMO descriptions; Ontologies, Web Services, Goals and Mediators.

- **Ontology Management** - The Ontology Editor can be used to define concepts, instances, axioms and relations in the ontology together with any mediators used by the ontology. The non-functional properties of the ontology can also be defined. Instances can be defined using the Instance Editor.

- **Service Management** -Service definitions are limited to non-functional and quality-of-service properties, used mediators and capabilities. The Capability definition for a Service includes properties, used mediators, preconditions, postconditions, assumptions and effects.

1 http://swws.ontotext.com
2 http://swws.semanticweb.org
• **Goal Management** - The Goal Editor can be used to define goals, which are specified in terms of their post-conditions, effects and mediators. This is shown in Figure 3.

• **Mediator Management** - Mediator definitions are created by selecting the Source and Target component(s) and then specifying used mediators. A Mediation service can be specified for ooMediators (used for mediating between Ontologies), and reduction axioms can be defined for wgMediators (used for mediating between Service Capabilities and Goals) and ggMediators (used for mediating between Goals).

![Figure 3. Defining a Goal in SWWS Studio](image)

**Service Composer**

The studio also provides a service Composer, and is used for defining service orchestrations, e.g. the interaction with other services in order to achieve the service functionality. The control flow constructs supported by the Service Composer include: Sequence, Parallel split, Exclusive choice and Iteration.
2.2.3 Key Features

- **Early WSMO support** – This tool was one of the first to provide support for creating WSMO descriptions and up until recently was the best tool for early WSMO adopters.

- **Import/Export** – There are several import and export options available allowing users flexibility to use in conjunction with other tools.

- **Composition** – The ability to create graphical composition based on WSMO descriptions.

2.2.4 Summary

The SWWS Studio is the tool with has most in common with the WSMO Studio, and are both based on the WSMO specification. On the surface the SWWS Studio seems to achieve most of the requirements set out for WSMO Studio. There is however a number of drawbacks to the design and implementation of the SWWS Studio that WSMO Studio aims to overcome:

- **Monolithic Architecture** - The architecture of the SWWS Studio was not designed to be extendable, and hence there is no way of adding plug-ins or extensions to the tool. One of the key aims of WSMO Studio is to provide on open architecture and allow third party developers to plug-in modules to add or replace functionality of the Studio.

- **Outdated WSMO support** - The Studio does not support the current latest version of WSMO. The WSMO parser built into the studio is proprietary and it does not use the Open Source wsmo4j API [11] that has been developed. Also the WSMO export options available are of a deprecated format, withdrawn by the WSMO working group.

- **OWL-S process model for composition** - At the time of development there was no formal specification for Choreography/Orchestration in WSMO so compositions created in the studio are exported as OWL-S process models. The WSMO Studio aims to fully support the WSMO Choreography/Orchestration model.
2.3 ODE SWS

ODE-SWS is a tool which allows users to design SWSs on the basis of PSM [15] modelling, enabling its description and composition at a conceptual level. This environment also performs verification about the consistency and completeness of the design created by the user. Once verified the user has a choice of which SWS language to export to, enabling the tool to be independent of a specific SWS specification language.

2.3.1 Architecture

The Architecture of ODE-SWS is based a 3 layers

- **Presentation Layer** – This layer manages the interaction between the user and the software system (i.e. the user interface). This is known as the ODE SWSDesigner.

- **Domain Layer** – This layer contains all the components that achieve the functions offered by the user interface. The SWSDesigner will directly invoke components when needed.

- **Data Source Layer** – In this layer provides a means for other applications to provide functionality not supported by ODE-SWS. At present this is composed of the WebODE\(^3\) platform which provides services for the management and access to ontologies.

![Figure 4. The ODE-SWS Architecture](http://webode.dia.fi.upm.es/)

---

\(^3\) [http://webode.dia.fi.upm.es/](http://webode.dia.fi.upm.es/)
2.3.2 User Interface

The ODE-SWS interface known as the ODE-SWSDesigner, provides features which allow the users to develop SWSs at a conceptual level; and a set of services (or tools), called ODE-SWS services, which support the linkage to the described ontologies and generates instance sets. Once a service has been defined the tool can be used to (1) verify the consistency and completeness of the SWS conceptual model; or (2) translate this model from its graphical description into a specific language.

The design of the interface is based round the classical modelling of problem solving methods, including hierarchies of task methods as well as input/output interaction diagrams & control flow of sub-tasks that compose a method.

The ODE-SWSDesigner consists of a number of views and graphical components that perform a variety of functions:

- **Task (method) trees** – Allows users to create the tasks associated with the functional features of a service (e.g. book plane ticket)
- **Ontology trees** – Shows the concepts and attributes of the ontologies used to specify the service input/output roles. Ontologies can not be created but can be imported.
- **Service definition** – Used to specify the functional and non-functional properties of a service including input/output roles, pre/post conditions, providers, commercial classification and quality rating
- **Decomposition** – Allows users to specify the decomposition of a method into sub-tasks (which are in turn other methods)
- **Knowledge flow** – Allows users to define the input/output interactions among the sub-tasks of a method.
- **Control flow** – Enables users to describe the control flow of a method.
2.3.3 Key Features

- **Independent of a specific SWS language** – ODE-SWS allows users to design services at the knowledge level, independent of specific SWS languages (such as WSMO/OWL-S). This has the advantage of reducing inconsistencies in the way services are described as well as enabling reuse by allowing the descriptions to be exported to multiple SWS languages.

- **Strong User Interface** – The graphical interface provided by ODE-SWS enables the description of a SWS to be constructed in an intuitive and easy to use manner.

2.3.4 Summary

ODE-SWS provides a novel approach to describing SWSs by completely decoupling the design from any concrete SWS language. This gives it some unique advantages over tools which are tied to a specific SWS language. It provides a clear and easy to use interface to allow service to be modelled. Although it is claimed that ontology management functions are not included to so that inconsistencies and errors are resolved in an ontology development environment, there may be applications where editing is required.
2.4 Overview and Comparison

This section provides a table comparing the three tools presented.

<table>
<thead>
<tr>
<th></th>
<th>IRS-III</th>
<th>SWWS Studio</th>
<th>ODE-SWS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SWS Features</strong></td>
<td>Describing</td>
<td>Describing</td>
<td>Describing</td>
</tr>
<tr>
<td></td>
<td>Publishing</td>
<td>Composing</td>
<td>Composing</td>
</tr>
<tr>
<td></td>
<td>Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Distributed SOAP-based</td>
<td>Client based, two components</td>
<td>Client based, three layer</td>
</tr>
<tr>
<td></td>
<td>Server-Publisher-Client</td>
<td>(Service editor, Composer)</td>
<td>architecture (Presentation,</td>
</tr>
<tr>
<td></td>
<td>architecture</td>
<td></td>
<td>Domain &amp; DataSource)</td>
</tr>
<tr>
<td><strong>Service Ontology</strong></td>
<td>TASK/PSM/WSMO</td>
<td>WSMO/OWL-S</td>
<td>TASK/PSM</td>
</tr>
<tr>
<td><strong>Key Features</strong></td>
<td>Early WSMO support.</td>
<td>Early WSMO support</td>
<td>Independent of a specific SWS</td>
</tr>
<tr>
<td></td>
<td>Ease of publishing.</td>
<td>Good Import/export options</td>
<td>specification language.</td>
</tr>
<tr>
<td></td>
<td>Open (can easily replace</td>
<td>Allows composition of WSMO</td>
<td>Enables reuse of knowledge level</td>
</tr>
<tr>
<td></td>
<td>certain functionality).</td>
<td>described service</td>
<td>descriptions with different SWS</td>
</tr>
<tr>
<td></td>
<td>Capability based</td>
<td></td>
<td>languages.</td>
</tr>
<tr>
<td></td>
<td>invocation.</td>
<td></td>
<td>Strong user interface.</td>
</tr>
<tr>
<td></td>
<td>OWL-S import</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Limitations</strong></td>
<td>Discovery/Invocation mechanism</td>
<td>Monolithic Architecture.</td>
<td>Some SWS applications may</td>
</tr>
<tr>
<td></td>
<td>may not suite some SWS applications.</td>
<td>Outdated WSMO support.</td>
<td>require built in ontology editor.</td>
</tr>
<tr>
<td></td>
<td>Does not support WSML</td>
<td>OWL-S process model for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>composition.</td>
<td></td>
</tr>
</tbody>
</table>
3 REQUIREMENTS

A detailed analysis of requirements for tools for Semantic Web Services (SWS) has been presented in [12]. In summary, SWS tools should assist potential users for ontology creation, service description, discovery and composition. More specific functional requirements for the client GUI tools include:

- Means for effective interaction with ontologies. The UI should provide functionality for creating, versioning, browsing ontologies (including large ones)
- Means for effective interaction with repositories - publishing and browsing (of ontologies, goal descriptions, service descriptions, mediators, etc.). The user interface should provide means for easy interaction with repositories.
- End-user interface for service discovery (e.g. finding services that achieve a specified goal). Such User Interface will assist the user for discovery of services that can be used for service description and for manual service composition
- Means for describing the capabilities of services, formulating goals and specifying mediators. The User Interface should allow even users with no good knowledge of logic to generate such descriptions (the tools should provide the underlying translation between logical formalisms if necessary).
- Means for manual service composition and workflow specification

3.1 Functional requirements

In the context of WSMO [29] and the specific requirements of DIP, the high-level requirements may be re-formulated and detailed to cover various aspects of the SWS lifecycle:

- Working with ontologies (including creating / modifying ontologies for SWS, mediation between ontologies, translation between different formalisms, etc.)
- Creating SWS descriptions in a WSMO centric way: goals, mediators, web services
- Describing service orchestrations and choreographies ([12], [30], [31])
- Import and export of the SWS descriptions in various formats as specified in [7] (other formats may be supported too)
- Interaction with SWS repositories (such as [17]) and ontology datastores (for example Sesame\(^4\))
- Interaction with SWS runtime environments, such as WSMX ([26], [34])

\(^4\) http://www.openrdf.org/
• Working with existing Web Service standards that may be indirectly relevant to the semantic description of a service (such as WS-MetadataExchange [3], WS-Policy [4], UDDI [32], WSDL [33])

Note that this list presents the functional requirements at this phase of the project but it is far from static - as the SWS domain evolves it is likely that additional requirements will emerge as well.

3.2 Non-functional requirements

The main goal of integrated environments such as WSMO Studio is to make users more productive\(^5\). Tool productivity can usually be increased in two ways:

• *by adding more features related to a specific task* (e.g. “vertical” extension), for example better ontology editors, ontology viewers capable of visualising huge ontologies, service composers supporting more powerful workflow primitives, etc.

• *or, by providing features covering more tasks* (e.g. “horizontal extension”), for example providing functionality for new tasks / perspectives, integration with already existing tools, etc.

As noted in [9], IDE users rarely focus on only one aspect or task when they work; instead users usually play different roles and perform different related tasks in their everyday work, so the environment should provide functionality covering various perspectives and tasks.

In the scope of SWS there are several roles such as:

• *ontology engineers* performing tasks such as creating / modifying / versioning / storing ontologies

• *service annotators* performing the actual semantic annotation of web services, e.g. translating a standard WSDL service (described in terms of ports, operations, inputs and outputs) into a WSMO service (described in terms of capabilities, pre/post conditions, assumptions, effects and mediators)

• *service users* querying for services that can be integrated into their business processes

• *service administrators* responsible for managing service repositories, datastores and execution environments

An integrated environment for SWS should provide sufficient functionality in relation to each of these perspectives, but at the same time should provide means for using these perspectives in an integrated way, since it is unlikely that a particular user working in the domain of SWS focuses only on a single task, e.g. the functionality should be

\(^5\) Note that different types of users may have different expectations (both functional and non-functional) from a tool.
provided in the form of a cohesive and flexible Integrated Service Environment (ISE, [5])

Another important non-functional requirement can be stated with respect to the extensibility of a tool. As stated in [9] “...open-ended extensibility is essential in the commercial IDE arena because no IDE vendor could possibly provide a sufficient set of useful tools to satisfy all customer needs. Which third party tool will be bundled as an add-in for a particular IDE is determined by market forces.” This statement, based on commercial IDE experience, can equally well be applied for ISE tools for the Semantic Web Services domain. Lack of extensibility is one of the main drawbacks of current SWS tools such as SWWS Studio [10] and IRS [21] at present.

Based on the above observations, we may define the following non-functional requirements for an Integrated Service Environment for the SWS domain:

- **Role-oriented development** – the ISE should allow for the end user to customise its functionality in a way that maximizes its productivity for a specific goal. For example, specific functionality may be added (removed) if it is relevant (irrelevant) to a specific user task or perspective. If a sufficient level of flexibility is provided then the ISE may satisfy the needs of a broader target audience (from SWS domain experts to more naïve users)

- **Extensibility** – since it is difficult to envision and specify a stable set of requirements for an emerging domain as SWS, it is crucial that the tools built for the domain are highly extensible. This way when the domain evolves in new directions the tools will be able to follow this evolution and provide the relevant functionality for the respective users (the extensions may be provided not only from the original group or community that build the tool but also from 3rd party contributors).

- **Open standards** – It is desirable that tools are designed and built in accordance with open standards and based on an open and extensible architecture, so that the cost of adopting and extending the tools is low for 3rd parties

- **Flexible licensing** – an open source licensing of a product improves its adoption rate and increases both its quality and active community base (contributing to and extending the product). Nonetheless, important differences exist between various swww.semanticweb.org open source licences in terms of copyright, compatibility with proprietary licences. For an open ISE, which relies on 3rd party contributions and extensions, it is important that the licence does not prevent such contributions.

- **Usability** – it is desirable that the tools provide the best UI experience for the end user. The UI should provide a convenient and easy-to-use abstraction of the domain being modelled in a way that maximises the end user productivity.
4 WSMO Studio Goals

Robust, mature and easy-to-use tools play crucial role for the easy adoption of any new technology. In fact the overall value of a technological innovation can be severely undermined by the lack of proper tools that support it. The Web Services Modeling Ontology (WSMO) provides a unique, highly innovative perspective onto Semantic Web and Web Service technologies.

The goal of the WSMO Studio effort is to provide a prototype that supports and elaborates that innovative perspective, making the SWS technology easy to use and transparent for the end user.

In particular, the WSMO Studio effort will provide an Integrated Service Environment (ISE) for the Semantic Web Services domain in the form of:

- An open architecture and a specification for a component (plug-in) based integrated environment for SWS
- A concrete implementation of this architecture in terms of a common runtime (based on wsmo4j [11]) and a set of plug-ins (that can further be extended by 3rd parties)

The following sections present more details on how WSMO Studio will cover the functional and non-functional requirements outlined in section 3.

4.1 Shortcomings of existing tools

A summary of tools relevant to the SWS domain has already been presented within the DIP project [12]. The two most prominent tools – SWWS Studio [10] and IRS [21] at this point in time fail to effectively cover some of the functional requirements and virtually all of the non-functional requirements presented in section 3.

4.2 Requirements resolution in WSMO Studio

In our opinion it is important that the functionality relevant to the SWS domain should be presented in a form that maximises its provided value, in other words both the functional and non-functional requirements should be handled by tool providers since focusing only on one of the aspects (f.e. providing functionality in a non-extensible, proprietary way) is unlikely to provide the desired results in the long term.

---

6 This statement is not intended to be a criticism to the SWWS Studio and IRS tools, but just to stress the point that we already presented – a single organization rarely has the resources to provide all the required functionality (especially in a still evolving domain such as SWS). Only tools open to 3rd party contributions can satisfy all customer needs in the long term.
4.2.1 Functional requirements

The main benefit of the WSMO Studio will be that it will provide an Integrated Service Environment for describing and using SWS in a WSMO centric way. The functionality should cover various aspects related to SWS (as outlined in section 3.1).

This functionality will be provided by means of:

- **Design** of a framework for an integrated service environment compatible with the WSMO approach of describing SWS. The architecture will be based on the Eclipse architecture.
- **Development** of a set of components (plug-ins) that add specific functionality to the service environment.
- **Integration** of already existing components into the service environment (for example existing components for WSDL)
- **Contributions** (from 3rd parties) of plug-ins compatible with the Studio

This way the WSMO Studio will provide a specification and implementation of a service environment which at the same time offers sufficient functionality for working with SWS and is open for continuous extension so that new functionality can be added when such requirements emerge.

4.2.2 Non-functional requirements

Section 3.2 outlined some important non-functional requirements for an ISE such as WSMO Studio: flexibility, extensibility, adherence to open standards, non-restrictive licensing, usability, etc. Details on how each of the non-functional requirements will be taken into consideration when designing and developing WSMO Studio follow:

- **Role-oriented development** – WSMO Studio is based on the Eclipse platform and its component (plug-in) based model, which is highly flexible and customisable – plug-ins may be added / removed / customised by the end user in a declarative manner, without any new development effort. The same base functionality (e.g. runtime + a set of plug-ins) may be extensively customised to cover the perspective of a specific end-user audience.

- **Extensibility** – the Eclipse component model presents a declarative specification of ways to extend the platform, called extension points (more details on the Eclipse platform are presented in section 5). New plug-ins may extend existing plug-ins and new plug-ins may be seamlessly integrated into the platform at any time.

- **Open standards** – WSMO Studio will be based on industry proven open standards such as Eclipse\(^7\) ([9], [24]) and OSGi\(^8\) ([27], [28]) as well as emerging

---

\(^7\) [http://www.eclipse.org](http://www.eclipse.org)

\(^8\) [http://www.osgi.org](http://www.osgi.org)
standard proposals as WSMO\textsuperscript{9} [29] (which is in the process of a W3C submission at present)

- Flexible licensing – our recommendation is that an open source licence, specifically LGPL \textsuperscript{10} [16], is used for the architecture and the common runtime, while the plug-in contributors are free to release their contributions under a licence most suitable for them. According to this scenario, an end user deployment may be comprised of both open source (WSMO Studio runtime + core plug-ins) and proprietary components (distributed under a licence chosen by the respective author\textsuperscript{10})

- Usability – the Eclipse platform has a highly configurable and portable UI, which offers a native and high performance user interface for a variety of window systems for Windows, Linux, OSX and QNX\textsuperscript{11}. The Eclipse 3 release has targeted especially responsive and scalable user interface [21]. The Eclipse platform has defined a set of user interface guidelines that aim at providing the best GUI experience for the end user [13].

5 Eclipse Platform

This section will provide a brief overview of the Eclipse platform, so that concepts and terms that will be used throughout the document are properly explained. A more detailed overview of Eclipse is available in [9], [24] and [7].

5.1 Plug-in based architecture

There are two components in the Eclipse architecture – a small runtime kernel and a set of plug-ins. Plug-ins represent the smallest unit of functionality within Eclipse and the runtime kernel is responsible for the plug-in lifecycle management.

5.1.1 Runtime kernel

The Eclipse runtime is a small, OSGi microkernel (based on the Equinox project\textsuperscript{12}). The OSGi Service Platform presents a standardised environment for managing the lifecycle of software components. With the adoption of OSGi in Eclipse v3.0, the Eclipse

\textsuperscript{9} http://www.wsmo.org
\textsuperscript{10} The LGPL licence allows that non-GPL components are bundled in a distribution as well.
\textsuperscript{11} See http://download.eclipse.org/downloads/drops/R-3.0.1-200409161125/index.php for the list of supported platforms in release 3.0.1
\textsuperscript{12} http://www.eclipse.org/equinox/
Platform now presents a dynamic plug-in model (e.g. plug-ins can be added, removed or updated at runtime, without restarting the Eclipse system).

When the Eclipse system (e.g. an Eclipse based application) is started, the microkernel discovers the plug-ins deployed within the system and builds a dynamic registry of the available plug-ins. Based on the information in the plug-in registry, individual plug-ins may be activated (instantiated) or deactivated at runtime.

### 5.1.2 Plug-ins

Plug-ins encapsulate the smallest unit of functionality within Eclipse. Apart from the microkernel, everything else in Eclipse is a plug-in (including the GUI, the tools, etc.). In other words, with the exception of the microkernel, which is responsible only for plug-in management, all the functionality provided to the end user is in the form of plug-ins (usually a small functional feature can be delivered as a single plug-in, but complex functionality is delivered in the form of a set of contributing plug-ins).

A plug-in is usually comprised of:

- A **plug-in descriptor** (described in section 5.1.2.1),
- The core plug-in functionality in the form of one or more Java library archives
- 3rd party Java libraries used by the plug-in
- Other resources required by the plug-in (images, read-only files, etc.)

Plug-ins may depend on other plug-ins (section 5.1.2.2), extend other plug-ins (section 5.1.2.3), be bundled together with other plug-ins or be composed from a set of fragments (section 5.1.2.4).

#### 5.1.2.1 Plug-in descriptor

The plug-in descriptor (or **manifest**) is an XML file (called **plugin.xml**), which presents the declarative description of the plug-in. The descriptor contains information such as the plug-in name, ID, version, dependencies on other plug-ins, exposed functionality, extension points and extensions. The plug-in descriptor describes how the plug-in will be integrated in the runtime environment.

---

13 A plug-in should be deployed in the **plugins** directory of the Eclipse installation in order to be discovered by the system

14 Apart from a small set of **core plug-ins** (which are activated each time when the platform starts) the actual plug-ins are not being loaded / activated at this step. A non-core plug-in will be dynamically loaded / activated only on-demand, when a request for its specific functionality is made.

15 Some plug-ins cannot be deactivated once loaded – dynamic deactivation depends on the plug-in design.

16 Based on the DTD from [18]
5.1.2.2 Dependencies
A plug-in may reuse functionality from other plug-ins in the system. Such a dependency is declared in the plug-in descriptor so that the Eclipse runtime may ensure that all required (prerequisite) plug-ins are available before activating a particular (dependent) plug-in.

5.1.2.3 Extension points
A plug-in may allow other plug-ins to extend its functionality (the former is usually called a *host plug-in* and the latter are *extender plug-ins*). The host plug-in declares one or more *extension points* in its plug-in descriptor, while extender plug-ins specify a set of *extensions* in their descriptors (e.g. which extension points of a host plug-in are being extended). An extension point contains a reference to an extension specific XML schema that specifies the format of information exchange between the host and extender plug-ins. This is a very flexible approach which allows extension points of arbitrary complexity to be declared.

Figure 6 presents the Eclipse plug-in architecture and relationships between individual plug-ins (extension or dependency).

The extension mechanism in Eclipse allows for loose coupling between components (the host plug-ins are unaware of the eventual extenders, apart from the fact that extenders should adhere to the extension point specification). In addition, the declarative dependency specification improves the modularisation of the platform.
The set of predefined extension points from the core plug-ins of the Eclipse platform is described in [19].

5.1.2.4 Fragments
Although logically a plug-ins is the smallest unit of functionality in Eclipse, a plug-in may be physically composed of several components called fragments. A fragment is also associated with a descriptor (called fragment.xml). At runtime all the fragments of a plug-in are logically merged into one component.

Fragments allow that parts of the plug-in be updated / installed / uninstalled independently from the plug-in. Fragments are most useful when parts of a plug-in are optional, for example localisation resources, language translations, environment specific resources, etc.

5.1.2.5 Plug-in activation
When the Eclipse platform is started the Eclipse runtime (microkernel) inspects the set of plug-ins deployed within the system and builds a plug-in registry. At this point plug-ins are still not activated. A plug-in is activated only when its functionality is required (implicitly by other plug-ins or explicitly by the user). With such a load-on-demand model the Eclipse platform offers optimal resource consumption, even if there are thousands of plug-ins deployed within the system.

5.2 Workspace
The Workspace sub-system of the Eclipse platform provides various abstractions and an API for resources that the platform operates on. Resources in Eclipse include:

- Files, Folders and Storage objects
- Projects that group resources into build-able units
- The Workspace that provides a context for all resources (there is only one workspace for the platform)
- Project natures that can be associated with a project to add specific behaviour to a project. Natures perform specific configuration when a project is opened.
- Markers that can attach metadata to a resource
- Builders (commands) that operate on the resources of a project and produce new resources. There are several types of builders: full, auto-build and incremental.

---

17 To be precise, some of the core plug-ins, part of the Eclipse platform, may be loaded at this point.
18 Provided by the org.eclipse.core.resources core plug-in
19 For example, a Java builder will operate on the .java files of the project and produce .class files
5.3 User Interface

There are two UI toolkits which part of the Eclipse platform:

- **Standard Widget Toolkit (SWT)** – an OS independent API for graphical widgets (lists, text fields, buttons, menus, etc.). The SWT implementation is integrated with the underlying windowing system (Windows, Motif, GTK, Carbon, etc.). Unlike Java Swing, the look-and-feel of SWT is not emulated but entirely integrated with the underlying native system, which offers optimal UI performance and experience.

- **JFace** – a high-level API built over SWT that offers high-level UI components (dialogs, wizards, editors, viewers, etc.). The JFace implementation is independent of the native windowing system.

The Eclipse UI is based on the Eclipse Workbench (the GUI of Eclipse) and its associated Workbench API. The latter is implemented over SWT and JFace.

The main Eclipse UI concepts are:

- **Views** and **editors** – containers for logically and functionally related UI sub-components (text areas, buttons, etc.).

- **Perspectives** – containers for logically and functionally related views and editors. Only one perspective is visible at any moment but several perspectives may be active. A perspective specifies the layout and visibility of the enclosed views and editors. It is recommended that a perspective provides all the functionality related to a specific task, so that the user won’t have to switch between different perspectives to accomplish his task.

Plug-ins with a UI part can be integrated within the Eclipse Workbench by extending the relevant extension points\(^\text{21}\) declared in the `org.eclipse.ui` core plug-in. Figure 7 presents a screenshot from the Workbench.

\(^{20}\) More details about builders are provided in [2]

\(^{21}\) The extension points for perspectives, views and editors are `org.eclipse.ui.perspectives`, `org.eclipse.ui.views` or `org.eclipse.ui.editors` respectively, as defined in [19]
6 WSMO STUDIO SPECIFICATION

This chapter presents the WSMO Studio specification. Sections 6.1 and 6.2 present the high-level specification of the WSMO Studio based only on the functional requirements. Section 6.3 contains a detailed specification with concrete product mappings, non-functional requirements resolution and UI guideline.

6.1 High-level architecture

The high-level architecture of the WSMO Studio is comprised of the following layers (see Figure 8):

- **Runtime** – a layer providing common functionality across all components of the WSMO Studio. Such functionality includes: creating WSMO models, export and import from various formats (as defined in [8]) and basic serialization. The runtime layer is based on the wsmo4j library [11] and its extensions. 3rd parties may extend this layer to provide specific functionality required by some of the UI components from the next layers (for example an export / import from a new format or a new serialization mechanism)

- **UI layer** – comprised of various UI components for working with ontologies, WSMO descriptions (goals, services, mediators) and goal/service repositories. 3rd parties may further extend this layer with new components (for example an orchestration and choreography editor).
• **Extensions** – various extensions and customisations of existing components from the UI layer may be provided within this layer, for example, a new axiom editor that may replace the default axiom editor part of the Ontology Manager component (see sections 5.1.2.3 and 6.3.3 for details on the extension mechanism)

![Figure 8 WSMO Studio architecture](image)

**6.2 Components**

The following components are described in this section: ontology manager, WSMO editor and Repository browser.

**6.2.1 Ontology manager**

The ontology manager can be used for browsing an ontology repository, visualising and creating / editing ontologies.

The main sub-components forming the Ontology Manger are:

- Ontology repository browser
- Ontology viewer
- Ontology editor

Next paragraphs present more details on the specific functionality of each sub-component:
1. Ontology repository browser – the repository browser presents the list of ontologies available in a specified ontology repository. The user can select / edit / delete an ontology from the list. Other ontology specific actions may be provided as well (f.e. exporting an ontology in a specified format such as WSML, OWL, F-Logic, etc. as specified in [8])

2. Ontology editor – this component provides means for creating and modifying descriptions of ontology components – concepts, instances, relations and axioms. Specific functionality includes:
   - Specifying the non-functional properties of an ontology: Contributor, Coverage, Creator, Date, Description, Format, Identifier, Language, Owner, Publisher, Relation, Rights. Source, Subject, Title, Type and Version (as defined in [29])
   - Specifying the imported ontologies
   - Specifying the used mediators
   - Concept Editor
     - Defining concepts (non-functional properties, super-concepts, attributes and the defining logical expression)
     - Defining attributes (non-functional properties, range)
   - Instance Editor (non-functional properties, concept, attribute values)
   - Relation editor
     - Defining relations (non-functional properties, super-relations, parameters, defining logical expression)
     - Defining parameters (non-functional properties, range)
   - Relation instance editor (non-functional properties, relation, parameter values)
   - Axiom editor (non-functional properties, defining logical expression)

3. Ontology viewer – although specific visualisation approaches (f.e. hyperbolic trees) may be useful for ontologies with huge number of components (concepts, instances, relations, etc.) no such functionality will be provided with the first versions of the WSMO Studio. Instead, the Ontology editor can be used for viewing the definitions of the ontology components (e.g. both editing and viewing functionality will be provided by the Ontology editor sub-component).
6.2.2 WSMO editor

The WSMO editor can be used for creating WSMO descriptions (goals, mediators, web services) in accordance with the Web Services Modelling Ontology [29].

The main sub-components forming the WSMO editor are:

- Mediator editor
- Goal editor
- Service editor

Next paragraphs present more details on the specific functionality of each sub-component:

1. Mediator editor – the mediator editor offers functionality for creating descriptions of WSMO mediators (ooMediator, ggMediator, wwMediator and wgMediator) and their components:
   - non-functional properties,
   - source component,
   - target component,
   - imported ontologies,
   - mediation service,
   - used mediator

2. Goal editor – the goal editor offers functionality to define goals:
   - non-functional properties,
   - used mediators,
   - post-conditions,
   - effects

3. Service editor – the service editor can be used for creating Web Service descriptions in accordance with WSMO:
   - Non-functional properties
   - Capability (non-functional properties, pre-conditions, assumptions, post-conditions, effects, used mediators)
   - Interface definition (non-functional properties, used mediators, orchestration and choreography definitions)\(^{22}\)

\(^{22}\) The choreography and orchestration interfaces in WSMO are not fully specified at present so a detailed specification for these components cannot be presented. The implementation should be guided by the latest drafts of [30][30] and [31]
6.2.3 Repository browser

The Repository Browser provides a centralised view of the ontology datastores and service/goal repositories the end-user may use elements from when annotating services (e.g. creating WSMO descriptions)

The main functionality of the repository component is:

1. Repository browsing – navigate an ontology/service/goal repository and select/view elements
2. Query a repository – navigation is not always sufficient for locating the elements (ontologies, services, goals) of interest. The repository browser should provide sufficient means for querying the repository
3. Import/export of the elements into/from the repository

6.3 Specification refinement

This section presents a refinement of the high-level specification from sections 6.1 and 6.2 according to the concrete product mappings.

6.3.1 Product mapping

The following product mappings can be identified for the initial version of the WSMO Studio23:

1. Java platform24 - WSMO Studio will be a J2SE standalone application. The first version will be based on J2SE 1.4.x since J2SE 5.0 is still not widely adopted at this time. Subsequent versions may be based entirely on J2SE 5.0
2. Eclipse platform - the Eclipse 3 platform will be the basis for WSMO Studio. WSMO Studio functionality will be provided as an Eclipse application (set of plug-ins) using the Eclipse specific UI toolkits.
3. wsmo4j library25 - the wsmo4j library provides low-level functionality for creating WSMO descriptions (ontologies, goals, mediators, services), import/export from various representation formats and basic persistence functionality.

23 Next versions and 3rd party contributions will extend the list of product mappings.
24 http://java.sun.com
25 http://wsmo4j.sourceforge.net
6.3.2 Plug-ins

Within the Eclipse platform, all the functionality (with the exception of the micro-kernel) is provided by means of plug-ins. Following the Eclipse approach we may distinguish the following plug-ins in WSMO Studio:

- WSMO runtime plug-in
- Ontology plug-in
- WSMO plug-in
- Repository plug-in

Figure 9 shows the relation between WSMO Studio and the Eclipse platform. All the functionality of WSMO Studio is split across a set of Eclipse plug-ins. The WSMO Studio plug-ins extend core/non-core Eclipse plug-ins and the 3rd party contributions (new plug-ins) to WSMO Studio may extend existing WSMO Studio plug-ins.

Figure 9 WSMO Studio within the Eclipse platform

6.3.3 Extension points

Each plug-in in WSMO Studio should define specific extension points so that other plug-ins may extend / replace its functionality. Extension points for the first version will
be fully specified (e.g. extension point schema and Java interfaces) and finalised throughout the development of the first prototype. Useful extension points may include:

- **Ontology plug-in:**
  - Browser – for plugging an ontology viewer / navigator which is different than the default tree-based navigator (f.e. ontology navigators based on hyperbolic trees are also available at present)
  - Concept/instance editor – for plugging alternative editors of concepts and instances
  - Axiom editor – for plugging alternative axiom editors (the axiom editor may be specific to the underlying logical formalism, f.e. Description Logic, F-Logic, etc.)

- **WSMO plug-in:**
  - Axiom editor – same as above
  - Choreography – an editor for service choreography interface based on [30]
  - Orchestration – an editor for service orchestration interface based on [31]
  - Import / export – plug-ins for exporting/importing the service definitions in various formats in addition to the ones defined in [8] (OWL, Flora, WSML, etc.)

- **Repository plug-in:**
  - Browsers – for plugging repository specific viewers and navigators
  - Query interface – for plugging alternative UI components for querying the repositories (depending on the query language supported by the repository)

### 6.3.4 Perspectives

Perspectives in Eclipse allow that functionality related to a specific task is grouped in the same visual container. This way all the visual elements (views, editors, etc.) that the user may need for accomplishing the task will be presented in the same perspective and the user won’t have to switch between the active perspectives.

Perspectives are a very useful way for achieving role-oriented development with the WSMO Studio, e.g. presenting only the functionality specific to a specific task (role) and abstracting from features irrelevant to the user’s goal. Such a role-oriented customisation of the tool provides maximum usability to the end-user and lowers the overall tool complexity.
The perspectives in WSMO Studio are closely related to the functionality provided by the different components (plug-ins) and so we can distinguish the following perspectives for the first version of WSMO Studio presented in Table 1.

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Target users</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology</td>
<td>Ontology engineers</td>
<td>Create / modify / manage ontologies</td>
</tr>
<tr>
<td>WSMO</td>
<td>Service annotators</td>
<td>Create WSMO descriptions for existing web services</td>
</tr>
<tr>
<td>Repository</td>
<td>Service users, service annotators, service administrators</td>
<td>Browse / query / manage a repository of services and goals</td>
</tr>
</tbody>
</table>

Perspectives will be properly declared in the respective plug-in descriptors so that at runtime they can be seamlessly integrated within the Eclipse GUI.

### 6.3.5 Views and Editors

The perspectives identified in the previous section will be comprised of the following visual components (views and editors):

- **Ontology perspective**
  - Ontology repository view – a tree-view navigator that lists the ontologies available in a specified ontology repository. The user may add / remove / edit the selected ontology
  - Ontology view – a view that presents the textual definition of the ontology (according to its format, e.g. WSML, OWL, F-Logic, etc.)
  - Ontology view – a tree-view navigator that shows the ontology concept/instance hierarchy as a tree
  - Ontology editor – an editor that allows modifications of the ontology components (e.g. its non-functional properties, concepts, instances, relations, relation instances, axioms). The editor is comprised of the following sub-components:
    - Non-functional properties editor – a form editor that allows non-functional properties of the ontology to be specified
    - Concepts & instances editor – a form editor that allows specification of concepts and instances (non-functional properties, super-concepts / concepts, attributes / attribute values)

---

26 3rd party contributions may define new, contribution specific perspectives as well
- Relation & relation instance editor – a form editor that allows specification of relations and their instances (non-functional properties, super-relations / relations, parameters / parameter values)
- Axiom editor that allows specification of the non-functional properties and the logical expression for the axiom

- WSMO perspective
  - WSMO repository view – a tree-view navigator that lists the goals/services available in a specified goal/service repository. The user may add / remove / edit the selected elements
  - Goal editor – a form editor that allows specification of goals (non-functional properties, used mediators, post-conditions, effects). Figure 11 presents a sample UI for such an editor.
  - Mediator editor – a form editor that allows specification of the four types of mediators and their components (non-functional properties, source component, target component, imported ontologies, mediation service, used mediators)
  - Service editor – a form editor that allows specification of a WSMO service (non-functional properties, capability, choreography and orchestration interface definition)

- Repository perspective
  - Repository view – provides the combined functionality of the ‘Ontology repository view’ from the Ontology perspective and the ‘WSMO repository view’ from the WSMO perspective
  - Query view – comprised of a form editor for specifying the queries (for goals, services, ontologies) in accordance to the respective query formalism of the underlying repository, and a tree view for presenting the list of results

Figure 10 and Figure 11 present sample UI for the ontology perspective (hierarchy viewer) and the WSMO perspective (goal definition)

27 The latter will only be defined as an extension point so that functionality can be plugged in the future when the orchestration and choreography specifications in WSMO are complete.
Figure 10 WSMO Studio - Ontology perspective
6.3.6 GUI guideline

We will not specify a custom GUI guideline for WSMO Studio. Instead, we will adopt the recommended Eclipse UI Guideline from [13] which contains recommendations and best practices for visual component design and consistency within the Eclipse platform.

7 CONCLUSION

This deliverable presents the specification of a n Integrated Service Environment for the Semantic Web Service domain, called WSMO Studio. A detailed list of the functional and non-functional requirements for the tool was provided, and the concrete means to achieve these requirements were outlined. This deliverable forms the input basis for two more deliverables within DIP – D4b.5 “WSMO Studio v.1” (due M18) and D4b.11 “WSMO Studio v.2” (due M24).
REFERENCES


