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

Deliverable

WP6: Architecture and Interoperability
D6.15
DIP Semantic Web Services Architecture and Information Model

Matthew Moran, Michal Zaremba
Maciej Zaremba, Thomas Haselwanter, Carlos Pedrinaci
Adrian Mocan, Tomas Vitvar

7th February 2007
SUMMARY

This deliverable provides a description for the DIP Semantic Web services architecture following the guidelines of the IEEE Recommended Practice for Architectural Descriptions of Software-intensive systems (IEEE 1471) [9]. This standard proposes the description of an architecture identifying concerns that should be addressed, stakeholders in the system and viewpoints describing different perspectives on the architecture. Five viewpoints are presented in this description. Within the DIP consortium, the intended audience for this deliverable are the members of the technical work packages (WP1 – WP5), as well as the project’s technical committee. Outside DIP, we hope this deliverable is useful for IT professionals interested in the architectural description of an execution environment for Semantic Web services including the components the architecture contains and how they relate to each other.

A global overview places the architecture in context of distributed service-oriented systems, relates the architecture to the DIP Description of Work (DIP Technical Annex), describes the relationship with the Web Service Modeling Ontology family of specifications (WSMO/L/X), and introduces the underlying principles of the architecture. The structural viewpoint identifies the constituent services of the architecture and their interfaces. The process view describes the behaviour of the architecture in terms of the defined processes it supports. The usage viewpoint relates how the architecture is used in the case study prototypes and how it relates to the IRS-III system. Finally, the technology viewpoint describes the technology used in the design of the Core middleware service offered by the architecture as well as describing background to the use of wrappers in the implementation of the execution semantics.

The architecture presents a vision of a potentially distributed architecture, where semantically described components can be plugged in and out, and where the description of the invocation order of the components is separated from their implementation. While we assume that for the Semantic Web services (SWS) infrastructure, a minimal set of components for discovery, mediation, composition, invocation etc. are required, the system should support the hosting of additional components, whose functionality may be unknown at system design-time. The architecture is designed to allow new or upgraded components to be plugged in and out without necessarily requiring a rebuild or restart the system.

The prototype for the DIP architecture, WSMX, is provided as open-source software at SourceForge.net and formed an integral part of the W3C member submission for Semantic Web services submitted by DERI, Galway (NUIG) and DERI Innsbruck (Leopold Franzens Universitaet, Innsbruck). The DIP architecture and associated execution semantics also provide the basis for the newly formed OASIS Technical Committee for specifying a Semantic Execution Environment (OASIS SEE TC). The focus of this OASIS committee is to further develop the Semantic Web services architecture and description of execution semantics initiated through WSMX.

The architecture described here is relevant for all deliverables in DIP responsible for providing a functional component for the DIP Semantic Web services execution environment prototype.

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>Acronym</th>
<th>DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP6 – 507483</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Title</th>
<th>Project URL</th>
<th>Document URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
<td><a href="http://dip.semanticweb.org/">http://dip.semanticweb.org/</a></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EU Project Officer</th>
<th>Werner Janusch</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Title</th>
<th>Work Package Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.15</td>
<td>DIP Semantic Web Services Architecture and Information Model</td>
<td>6</td>
<td>Architecture and Interoperability</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of Delivery</th>
<th>Status</th>
<th>Nature</th>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contractual</td>
<td>M36</td>
<td>prototype</td>
<td>public</td>
</tr>
<tr>
<td>Actual</td>
<td>7-Feb-07</td>
<td>report</td>
<td>consortium</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th>Misc</th>
<th>E-mail</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>M. Moran (NUIG), Mi. Zaremba (UIBK)</td>
<td></td>
<td><a href="mailto:matthew.moran@deri.org">matthew.moran@deri.org</a></td>
<td>+353 (91) 495-017</td>
</tr>
<tr>
<td>Ma. Zaremba (NUIG), T. Haselwanter (UIBK), C. Pedrinaci (OU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A. Mocan (UIBK), T. Vitvar (NUIG)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resp. Author</td>
<td>Matthew Moran</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partner</td>
<td>NUIG</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| Abstract (for dissemination) | This deliverable describes the Architecture for the DIP project. The document provides a number of views from the perspectives of different stakeholders, addressing various concerns. The views include a global overview, services view, process view, usage view and technology view. The deliverable provides a blueprint for the development of an instance of the architecture and additionally describes how the WSMX prototype implementation was used to satisfy a selected case study application. |
| Keywords | Execution environment, architecture, Semantic Web, Semantic Web services, execution semantics, SOA |</p>
<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 Jun 05</td>
<td>1</td>
<td>M Moran, Michal Zaremba</td>
<td>Draft for D6.5 (internal architecture deliverable)</td>
</tr>
<tr>
<td>27 Jun 05</td>
<td>2</td>
<td>M Moran</td>
<td>Updated based on comments of Edward Kilgarriff</td>
</tr>
<tr>
<td>28 Jun 05</td>
<td>3</td>
<td>M Moran</td>
<td>Updated based on comments of Laurentiu Vasiu</td>
</tr>
<tr>
<td>15 Sep 05</td>
<td>4</td>
<td>M Moran</td>
<td>Draft for D6.8</td>
</tr>
<tr>
<td>1 Dec 05</td>
<td>5</td>
<td>M Moran</td>
<td>Updated summary. Updated section 3 to include newer architecture diagram and slightly modified component descriptions. Updated section 4 to provide revised descriptions and UNL activity diagrams for WSMX execution semantics. Revised section 5 slightly to remove redundant paragraphs. Revised Appendix A to ensure API matches revised DIP architecture API (D6.9).</td>
</tr>
<tr>
<td>5 Dec 05</td>
<td>6</td>
<td>M Moran</td>
<td>Final updates to draft for review</td>
</tr>
<tr>
<td>15 Dec 05</td>
<td>7</td>
<td>M Moran</td>
<td>Updates after review</td>
</tr>
<tr>
<td>13 Jun 06</td>
<td>8</td>
<td>M Moran</td>
<td>Response to M24 DIP review comments and consequent modifications to the deliverable</td>
</tr>
<tr>
<td>17 Jun 06</td>
<td>9</td>
<td>M Moran</td>
<td>Draft of D6.11 Revision of the Architecture submitted for review.</td>
</tr>
<tr>
<td>28 Jun 06</td>
<td>10</td>
<td>M Moran</td>
<td>Review comments addressed and returned to reviewers.</td>
</tr>
<tr>
<td>31 Aug 06</td>
<td>11</td>
<td>M Moran, Maciej Zaremba, T Vitvar</td>
<td>Added usage viewpoint based on participation in SWS Challenge.</td>
</tr>
<tr>
<td>25 Sep 06</td>
<td>12</td>
<td>M Moran, Maciej Zaremba, T Vitvar</td>
<td>Updated usage viewpoint to include details of eBanking case study.</td>
</tr>
<tr>
<td>25 Sep 06</td>
<td>13</td>
<td>M Moran, Maciej Zaremba, T Vitvar</td>
<td>Modified behaviour viewpoint - renamed to process viewpoint.</td>
</tr>
<tr>
<td>25 Sep 06</td>
<td>14</td>
<td>M Moran</td>
<td>Remove p2p viewpoint and incorporated into Global overview.</td>
</tr>
<tr>
<td>Dec 06</td>
<td>15</td>
<td>M Moran</td>
<td>Renamed structural viewpoint to services viewpoint and revised section.</td>
</tr>
<tr>
<td>Jan 07</td>
<td>16</td>
<td>M Moran</td>
<td>Updated the deliverable to final state in response to the reviewer comments.</td>
</tr>
</tbody>
</table>

**Reviewers**

<table>
<thead>
<tr>
<th>Partner</th>
<th>E-mail</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ozelin Lóopez</td>
<td><a href="mailto:ozelin@isoco.com">ozelin@isoco.com</a></td>
<td></td>
</tr>
<tr>
<td>Michael Altenhofen</td>
<td><a href="mailto:michael.altenhofen@sap.com">michael.altenhofen@sap.com</a></td>
<td></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 Galway | NUIG | Dr. Sigurd Harand  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
E-mail: sigurd.harand@deri.org  
Tel: +353 91 495112 |
| Fundacion De La Innovacion.Bankinter | Bankinter | Monica Martinez Montes  
Fundacion de la Innovation. BankInter,  
Paseo Castellana, 29  
28046 Madrid,  
Spain  
Email: mmtnez@bankinter.es  
Tel: 916234238 |
| British Telecommunications Plc. | BT | Dr. John Davies  
BT (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  
E-mail: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlatt,  
Essex County Council,  
PO Box 11, County Hall, Duke Street,  
Chelmsford, Essex, CM1 1LX,  
United Kingdom.  
E-mail: maryr@essexcc.gov.uk  
Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe  
Germany  
E-mail: abecker@fzi.de  
Tel: +49 721 96540 |
| 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 |
<table>
<thead>
<tr>
<th>Company Name</th>
<th>Contact/Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILOG SA</td>
<td>Christian de Sainte Marie&lt;br&gt;9 Rue de Verdun, 94253,&lt;br&gt;Gentilly, France&lt;br&gt;E-mail: <a href="mailto:csma@ilog.fr">csma@ilog.fr</a>&lt;br&gt;Tel: +33 1 49082981</td>
</tr>
<tr>
<td>inubit AG</td>
<td>Torsten Schmale, inubit AG,&lt;br&gt;Lützowstraße 105-106&lt;br&gt;D-10785 Berlin,&lt;br&gt;Germany&lt;br&gt;E-mail: <a href="mailto:ts@inubit.com">ts@inubit.com</a>&lt;br&gt;Tel: +49 30726112 0</td>
</tr>
<tr>
<td>Intelligent Software Components, S.A.</td>
<td>Dr. V. Richard Benjamins, Director R&amp;D&lt;br&gt;Intelligent Software Components, S.A.&lt;br&gt;Pedro de Valdivia 10&lt;br&gt;28006 Madrid, Spain&lt;br&gt;E-mail: <a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a>&lt;br&gt;Tel: +34 913 349 797</td>
</tr>
<tr>
<td>Hanival WEB Solutions</td>
<td>Alexander Wahler&lt;br&gt;Hanival Internet Services GmbH&lt;br&gt;Niederacher &amp; Wahler OEG,&lt;br&gt;Kirchengasse 13/1a&lt;br&gt;A-1070 Wien&lt;br&gt;E-mail: <a href="mailto:wahler@hanival.net">wahler@hanival.net</a>&lt;br&gt;Tel.: +43 131 95843 11</td>
</tr>
<tr>
<td>MDR Partners</td>
<td>Rob Davies&lt;br&gt;MDR Partners&lt;br&gt;8 St. Andrew Street &amp; Hertford, Herts,&lt;br&gt;United Kingdom, SG14 1JA,&lt;br&gt;E-mail: <a href="mailto:rob.davies@mdrpartners.com">rob.davies@mdrpartners.com</a>&lt;br&gt;Tel.: +44 (0)208 8763121</td>
</tr>
<tr>
<td>The Open University</td>
<td>Dr. John Domingue&lt;br&gt;Knowledge Media Institute,&lt;br&gt;The Open University, Walton Hall,&lt;br&gt;Milton Keynes, MK7 6AA, UK&lt;br&gt;E-mail: <a href="mailto:j.b.domingue@open.ac.uk">j.b.domingue@open.ac.uk</a>&lt;br&gt;Tel.: +44 1908 655014</td>
</tr>
<tr>
<td>SAP AG</td>
<td>Dr. Elmar Dorner&lt;br&gt;SAP Research, CEC Karlsruhe&lt;br&gt;SAP AG&lt;br&gt;Vincenz-Priessnitz-Str. 1&lt;br&gt;76131 Karlsruhe, Germany&lt;br&gt;E-mail: <a href="mailto:elmar.dorner@sap.com">elmar.dorner@sap.com</a>&lt;br&gt;Tel: +49 721 6902 31</td>
</tr>
<tr>
<td>Sirma AI Ltd.</td>
<td>Atanas Kiryakov,&lt;br&gt;Ontotext Lab, - Sirma AI EAD,&lt;br&gt;Office Express IT Centre, 3rd Floor&lt;br&gt;135 Tzarigradsko Chausse,&lt;br&gt;Sofia 1784, Bulgaria&lt;br&gt;E-mail: <a href="mailto:atanas.kiryakov@sirma.bg">atanas.kiryakov@sirma.bg</a>&lt;br&gt;Tel.: +359 2 9768 303</td>
</tr>
</tbody>
</table>
| Unicorn Solution Ltd. | Unicorn Solutions Ltd, Malcha Technology Park 1, Jerusalem 96951, Israel  
E-mail: Jeff.Eisenberg@unicorn.com 
Tel.: +972 2 6491111 |
|-----------------------|---------------------------------------------------------------------|
| Vrije Universiteit Brussel | Pieter De Leenheer, Starlab- VUB  
Vrije Universiteit Brussel  
Pleinlaan 2, G-10  
1050 Brussel, Belgium  
E-mail: Pieter.De.Leenheer@vub.ac.be  
Tel.: +32 (0) 2 629 3749 |
## CONTENTS

1 INTRODUCTION .......................................................... 1
   1.1 Description Methodology ....................................... 1
   1.2 Motivation .......................................................... 2
   1.3 Organization of the Document ............................... 3
   1.4 Change History .................................................... 3
   1.5 Acknowledgment ................................................... 4

2 OVERVIEW AND UNDERLYING PRINCIPLES ......................... 5
   2.1 Global View ....................................................... 5
   2.2 Principles of the Architecture ............................... 7
      2.2.1 Middleware .................................................. 7
      2.2.2 Service Oriented Architectures (SOAs) ............... 9
      2.2.3 Event-driven Messaging .................................. 9
      2.2.4 Logical Reasoner Driven Functionality ........... 11
   2.3 Relationship to WSMO, WSML and WSMX .................. 11
   2.4 Requirements for Open Source and Standardization ...... 11
      2.4.1 Open Source ................................................ 11
      2.4.2 Standardisation ............................................ 12

3 SERVICES VIEWPOINT .................................................. 13
   3.1 Client Services .................................................. 13
   3.2 Business Services ............................................... 14
   3.3 Middleware Services ............................................. 14
      3.3.1 Core Component ............................................ 16
      3.3.2 WSMLReasoner .............................................. 17
      3.3.3 Resource Manager ......................................... 17
      3.3.4 Service Discovery ......................................... 17
      3.3.5 QoS Service Discovery ................................. 18
      3.3.6 Data Mediator .............................................. 19
      3.3.7 Process Mediator .......................................... 19
      3.3.8 Communication Manager ................................. 19
      3.3.9 Choreography Engine .................................... 21
      3.3.10 Parser ..................................................... 22
      3.3.11 Orchestration .............................................. 22
   3.4 DIP API ............................................................ 22

4 PROCESS VIEW .......................................................... 23
   4.1 Discovery-only Process ...................................... 23
   4.2 Invocation-only Process .................................... 23
   4.3 Combined Goal-based Discovery and Invocation Process .. 25

5 USAGE VIEWPOINT ...................................................... 27
   5.1 Usage of the architecture in the DIP Case Study Prototype . 27
      5.1.1 1 – Sending Request ..................................... 28
      5.1.2 2 – Discovery and Conversation Setup ........... 30
      5.1.3 3 – Conversation with Requester ........................ 31
5.1.4 Conversation with Provider (buying stock) ........................................ 31
5.2 IRS-III and the DIP Architecture ............................................................. 33
5.2.1 Integration Points for IRS-III with the DIP Architecture Prototype (WSMX) .......................................................... 33

6 TECHNOLOGY VIEWPOINT ........................................................................ 36
6.1 Core Design ............................................................................................... 36
  6.1.1 Management ......................................................................................... 36
  6.1.2 Communication and Coordination ....................................................... 38
6.2 Execution Semantics .................................................................................. 39

7 SUMMARY ..................................................................................................... 43
1 INTRODUCTION

This deliverable describes the DIP architecture for a Semantic Web service execution environment. The DIP work packages 1-5 provide the basic conceptual and technical building blocks required to set up DIP’s infrastructure while this deliverable describes how these building blocks are consolidated into a coherent software architecture that can be used as a blueprint for implementation.

Both the terms Web service and Semantic Web service are used throughout this document. The intent is that Web service corresponds to a service described using WSDL, available at a URI, that can be invoked on the Web using a combination of SOAP, HTTP and XML. Semantic Web services are Web services for which a semantic annotation has been created allowing for richer possibilities for discovery, composition and mediation of Web services.

1.1 Description Methodology

The IEEE Recommended Practice for Architectural Descriptions of Software-intensive systems (IEEE 1471) [9] recommends that software architecture descriptions be organized into one or more constituents called architectural views. Each view is used to represent the concerns of a particular viewpoint of one or more stakeholders in the architecture. In this context, stakeholders have an interest in or concerns relative to the the system. Concerns include system considerations such as performance, reliability, scalability, security etc.

A system stakeholder is defined as ”an individual, team or organization (or classes thereof) with interests in or concerns relative to a system”. The purpose of describing a system architecture is to record the architecture in such a way that it addresses the concerns and interest of these various perspectives. Identifying architecture stakeholders enables multiple different, sometimes overlapping, perspectives to be taken into account. In this revision of the architecture we identify five categories of stakeholders and their primary interest in the architecture description provided by this deliverable.

- Owner of the system
- Acquirer of the system (e.g. a project considering the adoption of the DIP architecture as a starting point for their work)
- Designer or developer of a system based on the the architecture
- Component owner (e.g. responsible for design of the Discovery component)
- Case study partner

The concerns that we aim to address in the context of this architecture description are:

- Does the architecture achieve what is required of it as described in the DIP Project Description of Work (internal project document)?
- What is the relationship to the Web Services Modeling Ontology/Language/Execution Environment (WSMO/L/X)?
• What are the underlying principles for the architecture?

• How can components in the DIP architecture be distributed across a network?

• Is there sufficient information and detail in the description to progress to a detailed technical design?

• What are the components that are required for the architecture?

• What are the processes that the architecture provides to enable the system functionality to be achieved?

• How are individual technical components (e.g. prototypes from the technical work packages 1-5) plugged into the architecture?

• How have case study partners used the architecture to build their prototypes?

IEEE 1471 defines views as "a representation of a whole system from the perspective of a related set of concerns". Viewpoints then are described as a "pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis." We will define four viewpoints that correspond to the perspectives of each of the stakeholders. The following four viewpoints are described in this deliverable:

• **Overview and Underlying Principles.** An overview of the architecture including its relationship to WSMO and WSMX and how the architecture addresses the requirements in the DIP Project Description of Work.

• **Services.** The structure of the architecture in terms of functional components, or services as the architecture is designed as a service-oriented architecture, and their interfaces.

• **Process.** The processes supported by the architecture to enable the required behaviour (in terms of interactions between components).

• **Usage.** How the components of the architecture are used, using one of the DIP case studies as an example.

• **Technology.** The technology used in the design of the prototype for the architecture

1.2 Motivation

In general where advances in computer and information science occur their uptake is greatly assisted by having tools and frameworks that allow both software engineers and potential users to try out the new technology. The DIP architecture aims to address both these interest groups. It provides a software engineering blueprint for the design of an execution environment for Semantic Web services. This execution environment layers on top of existing Web and Web service standards including URIs, XML, HTTP, SOAP and WSDL. It can be stand-alone or considered as a semantic-enabling layer for Web services in a distributed computing environment.
For designers and users of Semantic Web services, the DIP architecture prototype allows the focus to be directed on learning how to maximize the benefits offered by the formalised machine-understandable descriptions of the functional and non-functional aspects of Web services. The various components of the architecture encapsulate how these benefits are achieved leaving Semantic Web service users the freedom to focus on what problems they wish to solve rather than how the can devise a framework to make that possible. The DIP architecture describes the glue that brings the features promised by Semantic Web service technology to life. In the same way as middleware systems hide the detail of how they link autonomous heterogeneous applications and systems together, a Semantic Web services environment such as that of DIP hides the detail and complexity of how the requester and provider of Semantic Web services can communicate with each other even where they do not know each other in advance, speak different languages and expose different behaviours at their service interfaces.

1.3 Organization of the Document

The rest of this document is organized as follows. Chapter 2 describes a global overview, placing the architecture in context of distributed service-oriented systems, relates the architecture to the DIP Description of Work (DIP Technical Annex), describes the relationship with the Web Service Modeling Ontology family of specifications (WSMO/L/X), and introduces the underlying principles of the architecture. The services viewpoint of chapter 3 identifies the constituent services of the architecture and their interfaces. The process view in chapter 4 describes the behaviour of the architecture in terms of the defined processes it supports. The usage viewpoint of chapter 5 relates how the architecture is used in the case study prototypes and how it relates to the IRS-III system. Chapter 6, the technology viewpoint describes the technology used in the design of the Core middleware service offered by the architecture as well as describing background to the use of wrappers in the implementation of the execution semantics. Finally, chapter 7 provides a summary.

1.4 Change History

The first version of the architecture (M12) represented a high level approach identifying the main components required for a Semantic Web services architecture. As shown in figure 1.1, the second and the third versions of the architecture (M18 and M24) refined the architecture and focused on the design of the Core component and the integration of the functional component prototypes coming from work packages 1-5. In this version, we follow up on the recommendations from the second DIP review and align the architecture description with the recommendations of the IEEE 1471 Recommended Practice for the Architectural Description of Software Intensive Systems. In the context, we have introduced viewpoints on the architecture and have both revised earlier and created new UML models to help describe these viewpoints. This final version of the DIP Architecture (M36) brings include further refinements of the model descriptions, going into further details where necessary, with the aim of providing a comprehensive architectural description of the DIP system which can be used as a blueprint for further research in the field of Semantic Web service execution environments.
1.5 Acknowledgment

The work described in this deliverable was carried out jointly under the auspices of the DIP project funded by European Union’s IST programme (no. FP6 - 507483) and the DERI-Lión project funded by Science Foundation Ireland (SFI grant no. SFI/02/CE1/I131). The editors would like to thank the members of the WSMO, WSML, and WSMX working groups for their advice and input into this document.
2 Overview and Underlying Principles

This viewpoint describes a global view on the DIP architecture abstracting from the technical viewpoints on the structure, behaviour and prototype technology. It is intended to motivate and place the DIP architecture in context and to describe how it addresses the requirements set out in the DIP project description of work. The fundamental principles adopted by the architecture are introduced and the relationship of the architecture to WSMO, WSML and WSMX is explained. All stakeholders listed in section 1 are addressed by this viewpoint. In particular the owner and acquirer stakeholders may be interested in the motivation and guiding principles behind the architecture.

The rest of the chapter is organized as follows. Section 2.1 describes a global view on where the DIP architecture fits in terms of service-oriented architectures. Section 2.2 provides background material on the principles on which the architecture is based. The relationship of the architecture to WSMO, WSML and WSMX is explained in section 2.3. Finally, section 2.4 discusses how the architecture meets the requirements for open-source software and standardisation.

2.1 Global View

The DIP architecture does not exist in a vacuum. Instead it provides a layer of semantic middleware between service clients and service providers using the Web as their common network. The architecture diagram of figure 2.1 shows a global overview on the semantic enabled middleware provided by the DIP architecture. The figure is divided into two sides corresponding to two instances of the architecture connected over the Web. On each side there are three levels.

The top level represents the client entities that interact with the architecture. These can be client services (service requesters), system administrators (managing the services deployed in the architecture) and semantic modelers (domain experts creating WSMO descriptions ontologies, Web services, goals and mediators).

The middle level contains the middleware services defined for the DIP architecture. These include services for managing the operation (messaging, service management), discovery, composition, interoperability (mediation at data and process levels), resource management through registries and repositories etc. Additionally, on the left side the middleware is shown as existing on two separate servers connected using a shared message space. Shared spaces provide a messaging abstraction for distributed systems, either in the same network domain or across networks. This enables middleware services deployed on different servers to be considered as a single instance of a DIP architecture. The bottom level of figure 2.1 shows business services representing back-end systems that are exposed to the Web through their semantic descriptions.

Each instance of the DIP architecture can be considered as a node in a network of semantic middleware systems. The architecture itself can be described as a Web service. Interacting with an instance of DIP on the Web will follow the same pattern as for any other Web service invocation. Figure 2.2 shows how a simple distributed network of instances of DIP architectures, Web services and back-end systems could look. In the diagram, each DIP architecture acts as middleware responsible for handling messages with semantic content (WSML). One assumption is that the Web services
shown have semantic descriptions that have been registered with the DIP architecture instances. Another assumption is that the back-end systems (e.g., the ERP system)
has an adapter that enables it to send and receive WSML messages. Finally, the DIP architecture instances are capable of sending and receiving SOAP messages. The processes made available through the DIP architecture (described in section 4) are assumed to be exposed as WSDL Web service operations.

2.2 Principles of the Architecture

We describe four areas from which the architecture draws its underlying principles. These are

- Middleware
- Service oriented architecture
- Event-driven messaging
- Logical reasoners

2.2.1 Middleware

Middleware is the invisible software that allows independent applications using heterogeneous definitions for the various information items that they require, consume or provide to be able to communicate with each other. Ideally, from the perspective of each partner in a point-to-point communication, the middleware is transparent. It appears that, as each partner communicates using their own hardware, operating systems, information descriptions and rules, any differences are automatically reconciled. Middleware provides a very useful programming abstraction over different hardware and software platforms and the research on this topic is a mature field of study.

Despite their strengths, existing middleware systems do not provide sufficient support for applications to be linked together at runtime. The attempts to make this
possible are based on the pattern of encapsulating data and functionality behind a public interface. The interface is described in a language independent way with standardized bindings to major programming languages. An application can query the interface at run-time to determine how to interact with the application encapsulated behind the interface. There are two problems with the approach, one scientific and one technical. The first problem is that although the interfaces are defined using standardized types, there is no way for an application at run-time to determine the meaning of the information expressed in the interface. The assumption is that both the client and service developers have an implicit agreed shared understanding of the semantics of the information. The second problem is more mundane. For each middleware technology, there are a number of vendors. Implementations do not match and consequently, even for a prominent middleware technology like CORBA, heterogeneities can occur.

Web service technology provides a language and platform independent layer for middleware providing RPC-like functionality using Web-friendly specifications such as URIs, XML and WSDL. Although the perception is that Web services open up the Web to on-demand distributed service oriented computing, in fact they suffer from some of the same disadvantages as RMI-based technologies and are a much less mature technology. While TP monitors, such as IBM CICS, and middleware systems, such as CORBA, provide sophisticated services for messaging management, transactionality and security, Web services provide no advance on these features. Likewise, although the description of Web services using the Web Service Description Language provides a Web-friendly separation of interface from implementation, the semantics of the information described in the WSDL documents can not be determined by a computer without the intervention of a human user. As with CORBA and RPC-based integrations, Web service integration usually involves the creation of a client and server stub at design-time by software developers.

The DIP architecture provides a design for semantically enriched Web services middleware. It builds on the abstractions originally used for RPC and RMI in distributed computing and aligns them to the emerging specifications for the Semantic Web and Semantic Web services. For example, the Interface Definition Language (IDL) is used to define CORBA interfaces. Standardised mappings are available between IDL and common programming languages such as Java, C, C++ and Python. The mappings used by the DIP Data Mediation component provide the basis for a richer mapping between ontologies representing the information models of the communicating parties. The mappings are executed at run-time using a logical reasoning engine that ground the mappings to the actual types involved in the message exchange. This abstracts from the level of programming language types, allowing the information modeler to control the mappings, rather than leaving it as a task for the developers. Additionally, the meaning of the types in the exchanged messages are defined in a formal machine understandable language. The principle of relying on standardised message types is invisible to the service requester and provider. They are free to describe their message using their own independent semantic models. The middleware principle applies to the external view of the architecture. Ideally, once the semantic descriptions of the Web services, ontologies, mediators and goals are provided, an instance of the architecture acts as a black box providing the mediation between independent heterogeneous systems.
2.2.2 Service Oriented Architectures (SOAs)

The DIP architecture adopts fundamental principles for service oriented architecture, including two (service decoupling and strong mediation) described by the Web services Modeling Framework (WSMF) [4], a primary input into the design of the Web Services Modeling Ontology.

- **Service decoupling**
  The independence of the design and implementation of services is emphasised. Making each service self-contained supports a clear separation of concerns where each service has a well defined (and described) functionality that can be reused. Services make their functionality accessible through interfaces that can be understood by the service-oriented architecture infrastructure.

- **Service description**
  Service capability and the means for accessing that capability must be described in a way that is understandable to potential clients of each service. It must be possible to discover and invoke services based on the information provided by their descriptions. In DIP, the capability and interface of each service is defined semantically using a machine-understandable language.

- **Strong mediation**
  Although mediation is not necessarily a term that is commonly used for SOA, any such architecture must provide a means to enable services using different conceptual models (and syntax) for data or process to be able to communicate with each other. In the Semantic Web services world, this is usually called mediation, and is a top level principle of WSMO, the underlying conceptual model for DIP.

- **Service composability**
  It must be possible to compose simple services into more complex ones depending on requirements. There are two aspects to service composition in DIP. The first is the design of WSMO service orchestrations [17] that are executable by an orchestration engine. The second is how the services that make up the DIP architecture itself can be composed to provide the processes required for the architecture. We call the explicit declaration of a process supported by the DIP architecture an execution semantics. In section [4] we look at this in more detail. In [18], the execution semantics for the DIP architecture are defined.

The SOA principle relates both to the internal and external view on the architecture. Internally each component, providing functionality required for the operation of the architecture, is defined as a service and communicates with other services through the exchange of messages. Externally, the architecture facilitates service-oriented architectures to be defined where requesters and providers can be bound at run-time, even where they use different data and/or process models.

2.2.3 Event-driven Messaging

The Event-Driven Architecture [19] is an architectural style that prescribes a particular communication paradigm. It establishes that communication between components
has to be performed on the basis of event notifications, where events are basically understood as changes in the state of something relevant for the system, e.g. a purchase order has been received, the invoice has been dispatched, etc. At runtime, an agent (internal or external software component or a human) may generate an event notification. This notification is subsequently populated to the rest of the system through some communication channel, so that interested components can take the event and consume it according to their internal business logic.

Communication between system components solely takes place through event notifications which are generated by so-called event producers. These events are populated by the communication middleware, often referred to as Message-Oriented Middleware, and may trigger reactions by so-called event consumers. See for example the Java Messaging System [11]. Therefore, as opposed to the loose coupling provided by SOA, event producers and event consumers are completely decoupled, i.e., producers are not aware of consumers and vice versa.

Instead, the communication middleware is in charge of appropriately distributing the events over the different software components. Event notification is typically supported by the publish-subscribe paradigm, which can be formalised as the Observer Design Pattern [9]. Several consumers may subscribe as observers to an event type and, consequently, raising an event may trigger reactions in several components. EDA therefore supports one-to-many communications in an efficient manner as the event producer must not generate an event per consumer (producers may not even be aware of consumers). In fact, since several components might be running at the same time, nothing prevents them from generating events concurrently. Thus, EDA actually supports many-to-many communications. This contrasts with SOA implementations which are based on one-to-one communications.

Finally, EDA being based on a messaging mechanism, communication between event producer and event consumer is essentially asynchronous. Once an event has been generated, the event consumer is not required to wait until the execution of some processes executed by a set of event consumers finish. Still, if one requires synchronous communication, i.e., if the event producer requires some result back, this can be implemented by pausing the execution until an event with the result is received.

To take advantage of the characteristics in the preceding paragraphs, the DIP architecture adopts an event-driven mechanism. Events are defined for state changes during the execution of the system. For example, an incoming messaging received results in the creation of an event for that message to be parsed and transformed into an internal system representation. The successful completion of the discovery operation may result in an event being raised for data mediation to be performed if there is a semantic mismatch between the data provided by the service requester and expected by the service provider.

The event-driven principle applies to the internal view on the architecture. It enables asynchronous communication between the architecture’s components and provides flexibility for distribution of the components and scalability where multiple components for the same functionality are introduced to accept events of a particular type.
2.2.4 Logical Reasoner Driven Functionality

A logical reasoning engine is at the heart of the architecture. It makes possible the design of the components for discovery, mediation and those for handling the definitions of the process models that form part of the Semantic Web service descriptions. There are a variety of reasoning engines available, each designed to maximise the benefits of a particular logical formalism. For the DIP architecture, we provide a layer of abstraction to link the architecture with pre-existing reasoning engines. This layer provides a common API to access any reasoner used and a transformation service to transform between the logical language representation used natively by the architecture and the reasoner-specific input representation. In particular the architecture takes advantage of the WSML Flight reasoner developed as part of DIP work package 1\[8\]. This principle applies internally to the design of the various components and is reinforced by the architecture as the language used to define the messages sent and received to each component is WSML, a formal logical language, providing a knowledge base over which an inference engine can reason.

2.3 Relationship to WSMO, WSML and WSMX

The Web Service Modeling Ontology (WSMO) [20] defines the conceptual model for Semantic Web services used within the DIP project. While WSMO provides the model, the Web Service Modeling Language (WSML) [1] provides a family of languages to express the elements of WSMO. Messages exchanges at the interfaces of the functional components belonging to the DIP architecture are defined in terms of WSMO concepts expressed using WSML.

The Web Service Execution Environment (WSMX) [10] is a prototype implementation of the architecture described in this document. Both the architectural style implemented by WSMX and the architectural style of applications executed through WSMX are that of a service oriented architecture (SOA).

2.4 Requirements for Open Source and Standardization

2.4.1 Open Source

Chapter 1 of the DIP statement of work states that ”One of the key deliverables of DIP is an open source Semantic Web service architecture that is made public on a world-wide basis. The goal of the open source approach is to make the architecture easily available for other organizations (research as well as industrial). Through this approach a fast uptake of the work of DIP will be achieved”. Again in section 4.1.1 it is stated that the ”first main achievement of DIP will be the Semantic Web service architecture that will define a scalable platform for mechanized Web Service discovery, composition, and invocation. We will develop significant parts of this infrastructure in open source format in order to speed up the dissemination process of our efforts, ensuring that it will be widely if not globally used”.

The description of the DIP architecture, the definition of the DIP architecture API and the implementation of WSMX as an open-source prototype implementation of the DIP architecture address these stated requirements. The open-source nature
of the DIP architecture prototype and its use of WSMO to define the domain models for all messages exchanged between components of the architectures have also already encouraged other EU FP6 projects such as COCOON e-Health\textsuperscript{1} and the Integrated Project, SUPER\textsuperscript{2} to pick up this work as foundational input for their research.

2.4.2 Standardisation

The requirement for the DIP architecture to drive standardisation activities is raised in the DIP Project Description of Work: "the openness of eCommerce cannot be achieved without standardisation. This lesson can be learned from the success of the Web. But the standardisation requirements for eCommerce will be much more stringent. This will require standardisation of the actual content and business logics that are exchanged, which goes far beyond the requirement of standardising protocols and document layouts. DIP will develop significant contributions to this area (see also standardisation)" and again in section 5.1: "standardization of Web Services technology, e.g. W3C Web Services Activity, OASIS, WS-I. The project will closely monitor these standardization efforts, will provide reference implementations for some of the emerging standards in this area, and will participate in interoperability events”.

The related efforts of WSMO/WSML and the SWS architecture, developed through DIP, have been accepted as a member submission to the W3C activity on standardising Semantic Web services\textsuperscript{3}. Additionally, at the beginning of November 2005, a new Technical Committee (TC) was formed at the OASIS eBusiness-standards consortium. The new committee is called the Semantic Execution Environment TC (SEE TC) and has the mission to develop guidelines, justifications, and implementation directions for deploying Semantic Web services in service-oriented architectures. The two technical deliverables of the SEE TC are based on the work of DIP architecture and execution semanticsrespectively.

\textsuperscript{1}http://www.cocoon-health.com/
\textsuperscript{2}http://super.semanticweb.org/
\textsuperscript{3}http://www.w3.org/Submission/2005/06/
3 Services Viewpoint

This section describes the services viewpoint of the DIP architecture. Services are the basic building blocks of the architecture enabling the functionality, the architecture is designed to provide. There are three categories of services associated with the DIP architecture: client services, business services and middleware services, as illustrated in figure 3.1. In this section, we describe the conceptual definition of these services and their interfaces. The stakeholders for this viewpoints are software engineers and computer science researchers concerned with the functional services required to take full advantage of semantic annotations of Web service descriptions. The concerns addresses are what are the services that are required to make up the architecture, what are their interfaces and what dependencies exist between them?

The section is organized as follows. Section 3.1 and 3.2 describes what is meant, conceptually by client services and business services respectively, with respect to the DIP architecture. Sections 3.3 describes the middleware services that are provided by the architecture, including their interfaces and dependence on other services where appropriate. Section 3.4 provides the links to the online description of the Java implementation of the DIP API, derived from the DIP services description.

![Figure 3.1: Services View](image)

3.1 Client Services

The architecture provides the functionality it offers as processes that can be initiated by a client service. Generally, this occurs when a client sends a WSML message to an instance of the architecture. There are two types of process, (1) business related, and (2) Management related. These are described further in the following bullets:

- Business-related client services are those which use the DIP architecture as a broker for Semantic Web services. These services initiate the sending of WSMO goals to the architecture that lead to the discovery and/or invocation of business services. Figure 3.1 indicates that where a client service does not support the WSML language, an adapter is necessary as a bridge. Messages between the client services and the adapter are in a format supported by the client service.
Messages between the adapter and the DIP architecture are in WSML. Specifically, where a service request is being sent from the client service, the adapter takes the responsibility of transforming this request into a WSML goal. An example of this, from the eBanking case study, is described in detail in section 5.1.

- Management-related client services are those that are used for management or administrative tasks e.g. the management of middleware services or the content of a WSMO repository. The approach to the management of middleware services is described in detail in section 6.1.1. Access to the WSMO repository is facilitated through the interfaces defined for the ResourceManager service in the DIP API, described in section 3.3.3.

3.2 Business Services

Business services, in the context of the DIP architecture are those services that have been described using WSMO and that have a grounding to a concrete transportation protocol, (e.g. to WSDL as defined in [16]). There is no limitation on the domain of interest for the business services. The three case studies elaborated on by DIP are in the areas of eGovernment, Telecoms and eBanking.

Adaptation is also required between the data-definition language used by the business services, and WSML used by the DIP architecture. The majority of existing Web services are implemented using XML Schema to describe the message elements, and SOAP as the messaging protocol. The need, then, is for an adapter that lowers from WSML to XML (messages from DIP to business service) and that lifts from XML to WSML (messages from business service to DIP). Lifting and lowering is included in the functionality required of the DIP CommunicationManager service (section 3.3.8). The current approach to lifting and lowering for WSMO is available in the technical report at [15].

3.3 Middleware Services

Middleware is software that allows independent applications, using heterogeneous definitions for the various information items that they require, to be able to communicate with each other, in an apparently transparent manner. For the DIP project the middleware services, provided by the architecture, enable it to act as a broker between client services, issuing requests for services, and business services who are capable of satisfying those requests. The purpose of the middleware services is to abstract from the client and business services any heterogeneities that may exist between their respective information models, enabling them to communicate with each other. Ultimately, through discovery, composition, mediation and invocation based on semantic service annotation, the DIP architecture aims for flexible goal-driven service invocation with service requesters and providers only bound together at run-time (rather than up-front during the design phase).

Figure 3.2 shows a UML component model (DIP middleware services are represented as UML components), showing each service’s interfaces and dependencies on the interfaces of other services. Interfaces are represented as lines terminated with a
circle (sometime referred to as lolly-pop sticks). Where one service requires an interface provided by another, a line with a cup at the end is drawn from the dependent service to the desired interface. An example, is that the data mediator service requires the reasoner interface. With the exception of the Core, each service is depicted with a wrapper interface. Conceptually, wrappers are used to isolate the middleware services from the management and messaging functionality of the architecture. They are responsible for indicating that a middleware service is available to the architecture, handle the creation, handling and exchange of events with the messaging infrastructure, and make the invocation of the middleware service functionality, through the methods defined by the DIP API. The usage and implementation views (chapters 5 and 6) describe how the prototype generates the wrappers for a middleware service when it is plugged into the DIP architecture, either at start-up or while the system is running.

Events, handled between the wrappers and the event passing system, have the following structure:

- A context identifier (for the instance of the execution semantics).
- A type (to which middleware services must subscribe if they wish to consume the event).
- A means to locate any WSML content that is associated with the event (e.g. this may be by value or by reference).

Figure 3.2: UML Component Diagram for the Middleware Services

The following sections briefly describe each middleware service. They provide short descriptions of the interfaces required of each service. Full definitions of the interfaces, written in Java and presented using the JavaDoc html utility, are available in the DIP API (see section 3.4).
3.3.1 Core Component

The Core Component takes the role of service co-ordinator for the DIP architecture. It manages interaction between the other services through the exchange of messages expressed in terms of WSMO concepts expressed in WSML. The Core service provides the microkernel and messaging infrastructure for the architecture.

The Core Component has three interfaces -

**ManageSystem.** This is a management and monitoring interface to start and stop the system and allow its status to be monitored. Minimally, the interface offers the following functionality:

- `startSystem()` - start the DIP system
- `stopSystem()` - stop the DIP system
- `startService(service identifier)` - start a specific service
- `stopService(service identifier)` - stop a specific service
- `getServiceStatus(service identifier):service status` - retrieve the status of a middleware service

**ManageExecutionSemantics.** In the DIP architecture, an execution semantics is the executable description of one process that is offered by the architecture itself. In section 4, three such processes are identified. Each instance of an execution semantics gets a unique identifier when it is instantiated. Any events generated by middleware services that correspond to a particular instance of an execution semantics must contain the unique identifier for that execution semantics. Multiple instances of each execution semantics may run concurrently, as the events handled by each are uniquely identified.

- `startExecutionSemantics(executionSemantics type):executionSemanticsIdentifier` - start the execution semantics and return an identifier.
- `stopExecutionSemantics(executionSemantics identifier)` - stop the identified execution semantics.

Event types are listed in section 5. The ongoing work on the DIP architecture, as adopted for the EU SUPER project, will investigate how execution semantics can be defined as orchestrations which can be loaded, updated and removed for the architecture dynamically.

**ManageMessaging.** The Core acts as a message bus which accepts messages and routes them to a suitable target service. Multiple concurrent message exchanges may be maintained. This interface must support two operations:

- `sendEvent(typed event)` - send event to the messaging transport mechanism
- `getEvent(typed event)` - receive event from the messaging system

The typed event corresponds to an event type and an execution semantics identifier. The messaging mechanisms should be polled regularly to check if new events have been published.

3.3.2 WSMLReasoner

The WSML Reasoner, specified in [8], is at the heart of the architecture, providing a logical reasoning capability that is required by many of the other middleware services, notably discovery, process and data mediation, choreography and orchestration. The benefit of using formal ontologies for the semantic annotation of Web services and goals can only be realized through the reasoner. It must allow the knowledge base on which it operates, to be made available and must accept queries against this knowledge base. The WSML reasoner, used in the DIP architecture, based on KAON2[^2] allows for hierarchical queries on concepts, such as requests for subconcepts or superconcepts, entailment and, some support for queries against the knowledge base available to the Reasoner.

The required interface for this service is -

**WSMLReasoner.** This includes operations for retrieving subconcepts, superconcepts, for querying on concept subsumption, for logical entailment, for retrieval of all instances of a particular concept, and more. The full functionality offered by the implementation of the KAON2 reasoner is described in specification provided at [8].

3.3.3 Resource Manager

The ResourceManager is the service providing access to persistent storage for the DIP architecture. Persistence at the system level is required for all WSMO entities used during operation of the DIP system as well as the nonWSMO entities e.g. those representing messages and their states used during the execution of processes provided by the architecture (through execution semantics). Individual middleware services are responsible for any specific persistence required by their own design.

There are two interfaces for this service -

**WSMORespository.** This manages persistence for descriptions of WSMO descriptions of Web service, Goals, Ontologies and Mediators. The DIP API (section 3.4) defines an interface for each of the top level elements of WSMO. WSMO Studio[^3] implements the DIP API in this regard, providing a possible implementation for this part of the Resource Manager.

**NonWSMORespository.** This manages persistence for entities required by the operation of the architecture. The requirements for this interface are to record the life cycle of events and messages so that this information can be made available through the ManageSystem interface of the Core.

3.3.4 Service Discovery

The Discovery service is concerned with finding Web service descriptions that match the goal specified by the service requester. A service requester provides the WSMO

[^2]: http://kaon2.semanticweb.org  
[^3]: www.wsmostudio.org
description of a goal they wish to achieve (described in terms of a desired capability with preconditions, assumptions, effects and postconditions). This is matched to the WSMO description of Web services known to DIP (described in terms of offered capabilities).

A specification for a lightweight Semantic Web service discovery component for DIP is provided in [5]. The authors explain how there are two primary schools of thought in modeling Semantic Web services. The first models services as transitions of states describing the service before after it executes. The second characterizes a service by means of the objects it can deliver. The term object, in this sense, means something of value the service delivers in its domain of interest. For example, a service might deliver a train ticket or a shipment-order-confirmation document. In the specification of [5], services are modeled in terms of the objects they provide through the definition of preconditions, postcondition, assumptions and effects. The discovery service returns a (possibly empty) list of Web service descriptions. The conceptual description of the interface is:

There is one interface - **Discovery**, with the following conceptual method description:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>discoverService(WSMO Goal)</td>
<td>A (possibly empty) set of WSMO service descriptions</td>
</tr>
</tbody>
</table>

### 3.3.5 QoS Service Discovery

A complementary effort for Semantic Web Services discovery is based around the use of quality-of-service attributes as described in [13]. The focus here is on providing upper level ontologies for describing various domains for quality-of-service attributes that may be relevant to service descriptions. Non-functional properties are introduced to the Semantic Web Service descriptions whose values are defined in these QoS ontologies. A suitable WSML reasoner is able to compare required QoS versus offered QoS characteristics and return a list of matching services ordered by criteria specified by the service requester. For example, an ontological definition for response-times may be shared by both the Goal and the Service descriptions and a Goal may be specified to include a requirement on response-times along with a request to return any matching service descriptions in descending order of how quickly they promise to respond.

There is one interface, **QoSDiscovery**, with the following conceptual method description:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>qosDiscoverService(WSMO goal, WSMO ranking ontology)</td>
<td>A (possibly empty) set of matching services, ranked in order of the QoS parameters specified in the ranking ontology</td>
</tr>
</tbody>
</table>
3.3.6 Data Mediator

The DIP Data Mediator service has the role of reconciling the data heterogeneity problems that can appear during discovery, composition, selection or invocation of Web services. It is expected that this service should use inter-ontology mappings described in a suitable mapping language (defined at design-time) to implement the required mediation. Since the mappings will need to be stored one of the dependencies for the service is to the NonWSMOInterface of the ResourceManager. Accessing WSMO entities means a dependency on the WSMOInterface of the ResourceManager. The mappings required by the mediator will need to executed by a suitable logical reasoner. Hence there is a further dependency to the Reasoner service. The DIP Data Mediation module specification is available to the consortium only in the deliverable D5.2 [14]

The Data Mediator has one interface, DataMediator, with the following conceptual description:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataMediate(source WSMO ontology, target WSMO ontology, source-WSMO-ontology instances)</td>
<td>target WSMO ontology instances</td>
</tr>
</tbody>
</table>

3.3.7 Process Mediator

The DIP Process Mediator has the role of reconciling the public process heterogeneity that can occur during the invocation of Web services. That is, ensuring that the public processes of the client (represented by a Goal) and the matching service match. Since both the client and the Web service publish their public processes as choreographies, and the public processes are executed by sending/receiving messages, the process mediation service will deal with reconciliation of message exchange patterns based on choreography. Where data mediation is necessary, it is expected that it takes advantage of the data mediation service. The DIP Process Mediation module specification is available to the consortium only in the deliverable D5.3a [2]

The Process Mediator has one interface, ProcessMediator, with the following conceptual description:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>processMediate(source WSMO process ontology, target WSMO process ontology, source-WSMO-process-ontology messages)</td>
<td>target-WSMO-process-ontology messages</td>
</tr>
</tbody>
</table>

3.3.8 Communication Manager

The Communication Manager is responsible for dealing with the protocols for sending and receiving messages to and from DIP. As most existing Web services use SOAP over
HTTP for message exchange (and described using WSDL), this is the default that the CommunicationManager must support. The service should be open to supporting as many transport and messaging protocols as possible. This should be transparent, both to the client service making the request, and the business service providing the capability. As mentioned earlier, lifting and lowering between XML and WSML is a necessary default adaptation requirement. In this version of the architecture, handling this task is the responsibility of the CommunicationManager where the Invoker interface is used and the message exchange is synchronous. Where the message exchange between the CommunicationManager and the Web Service is asynchronous, the Receiver interface is used and expects the contents of the messages it receives to be already available as WSML. The four required interfaces are described in the following paragraphs.

**EntryPoint.** This interface provides the entry points for client services to initiate instances of the execution semantics made available by DIP. For example, a client service may send the DIP system a goal they wish to achieve as well as the input data (ontology instances) corresponding to that goal. The goal and input data can be sent together or in separate messages. The Receiver expects the contents of all messages it receives to be expressed in WSML. If the client uses a different language, then an adapter is required between the client service and DIP (as illustrated in figure 3.1).

The conceptual description of the operations of the EntryPoint interface are:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieveGoal(WSML goal)</td>
<td>context uniquely identifying the DIP conversation between the client and business service</td>
</tr>
<tr>
<td>getWebService(WSML goal)</td>
<td>Set of WSML Web service descriptions</td>
</tr>
<tr>
<td>getRankedWebService(WSML goal, WSML ontology for ranking)</td>
<td>Set of WSML Web service descriptions ordered as specified in the ranking ontology</td>
</tr>
<tr>
<td>invokeWebService(WSML document containing the Web service along with an ontology containing instance data for input)</td>
<td>context uniquely identifying the DIP conversation between the client and business service</td>
</tr>
</tbody>
</table>

**Invoker.** The Invoker interface is used whenever a Web service needs to be invoked by the DIP system. The invoker receives the WSMO description of the service, the endpoint for the invocation and the data that that particular endpoint expects to receive. It is responsible for making the actual invocation of an operation on a service including the lowering and lifting operations (described above).

The conceptual description of the operation of the Invoker interface is:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokeService(WSMO Web service, WSMO list of instances for data to send to the service, optional grounding information for lifting/lowering)</td>
<td>list of WSMO instance data, received from the business service (possibly after lifting</td>
</tr>
</tbody>
</table>

**Receiver.** The receiver interface is used for asynchronous communications, accepting messages that correspond to conversations that have already been initiated between
a client service and a business service via the DIP system. Either the client, or the 
business service, sends a message to this interface, including a context that indicates 
to the system the sending party and an identifier to the conversation.

The conceptual description of the operation is:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive(WSML document containing message, context to identify the conversation)</td>
<td>context identifier for the conversation</td>
</tr>
</tbody>
</table>

**WSMORegistry.** The WSMORegistry interface exposes the methods for the DIP 
system to act as a repository of WSMO entities. Conceptually, the operations pro-
vided are for the creating, updating or deleting WSMO elements. They are not listed 
individually here but can be found in the DIP API (section 3.4.

### 3.3.9 Choreography Engine

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

After discovering a Web service description, one has to know the observable be-
aviour of the Web Service in order to achieve the desired functionality. Choreogra-
phy and Orchestration comprise part of the interface definition of a WSMO service 
description. Choreography describes how to communicate with the service such that 
the service will provide its capability. Orchestration describes how the service collabor-
ates with other WSMO services to achieve its capability. The *Choreography Engine* is 
responsible for using the choreography descriptions of both the service requester and 
provider to drive the conversation between them. It is the responsibility of the engine 
to maintain the state of a conversation and to take the correct action when that state 
is updated. For example, the update of the state of a choreography instance may be 
the result of a message received from a service provider. The consequent action, as 
described in the choreography instance, could be to forward the unchanged message 
to the service requester. The DIP ontology for describing Web service choreographies 
is available to the DIP consortium in the deliverable D3.8 [17].

One required interface is exposed for registering choreographies and updating chore-
ography instances with messages - *ChoreographyEngine*.

The conceptual operations of the interface are:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerChoreography(WSMO Web service)</td>
<td>void</td>
</tr>
<tr>
<td>registerChoreography(WSMO Goal)</td>
<td>void</td>
</tr>
<tr>
<td>updateState(Identifier of goal or Web service, WSMO instance data)</td>
<td>void</td>
</tr>
</tbody>
</table>
3.3.10 Parser

The Parser parses incoming WSML messages to convert them into the internal data representation for the DIP architecture. The architecture prototype uses the WSMO4j Java class model for its internal data representation so in this case, the parser validates the WSML message and, if everything is ok, creates the corresponding WSMO4j objects.

The required interface for this service is - Parser.

The conceptual operation of the interface is:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse(WSML document containing WSMO element(s))</td>
<td>Set of WSMO4j entities</td>
</tr>
</tbody>
</table>

3.3.11 Orchestration

The Orchestration component provides the means for executing orchestrations of Semantic Web services as defined in terms of the DIP ontology for Web service orchestration, available in the deliverable D3.8 [17]. The final version of this ontology will be available in the deliverable D3.9 at the DIP web site [4].

The required interface for this service is - Orchestration.

The conceptual operation of the interface is:

<table>
<thead>
<tr>
<th>Method and Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerOrchestration(WSMO4j service object)</td>
<td>identifier for an orchestration instance</td>
</tr>
<tr>
<td>updateState(Instance data for an orchestration instance, identifier of the orchestration instance)</td>
<td>void</td>
</tr>
</tbody>
</table>

3.4 DIP API

The DIP API specifies the actual methods in the interfaces that each of the functional services comprising the DIP architecture must implement. It is defined using Java and is implemented using Java and published at the open-source SourceForge WSMX project. The HTML description of the API, created using JavaDoc, is available at the DIP deliverables Web page: http://dip.semanticweb.org/deliverables.html.

4 Process View

This section describes a process viewpoint on the DIP architecture. The architecture uses the execution semantics mechanism to provide the process behaviour. Summaries of three processes are presented in this chapter using UML Activity Diagrams. The DIP deliverable 6.16 on execution semantics specifies the processes in detail, including data and control flow, and explains how the three-layer DIP orchestration model may be applied to provide a formal semantics to the UML. The three processes described here are:

- Goal based service discovery
- Invocation of a specified service (no discovery)
- Goal-based service discovery and invocation

Additionally, chapter describes in detail Goal-based service discovery and invocation for the eBanking case study.

This viewpoint is targeted at the following stakeholders: acquirer of the DIP system, developers, component owners. It may also be of interest to the case-study partners. The viewpoint addresses the following concerns:

- What are the behavioural processes defined for the DIP architecture?
- Which middleware services are required for each process?

The section is organised as follow. Section provides the UML activity diagram and brief description for the discovery-only process. Section of the one-way goal execution scenario. Section provides the diagram and description for the invocation-only process. And section provides the diagram and description for goal-based discovery and invocation.

4.1 Discovery-only Process

In figure, a WSML goal is received by the CommunicationManager service. The Parser takes care of the transformation into the WSMO4j representation. Service discovery may need to make an invocation on the Data Mediation service. This call is made as part of the discovery activity and is not shown here. QoSDiscovery may be used to sort any discovered services, in an order determined by a provided ranking ontology, if one was provided. Once the discovery activities complete, a choreography instance is created in the Choreography Engine and this is updated with the information corresponding to the discovered services. The Choreography Engine then raises an event for the Invoker interface of the Communication Manager to send the list of discovered Web service descriptions to the sender of the goal (client service in the context of services viewpoint).

4.2 Invocation-only Process

In figure, a WSML Web service description with WSML instance data (input for the Web service) is received by the CommunicationManager service. As in the previous
Figure 4.1: Discovery-only Activity Diagram

Figure 4.2: Invocation-only Activity Diagram
example, the Parser takes care of the transformation into the WSMO4j representation. There is no need for Service discovery, so the control moves to the Data mediation activity which checks if this mediation is required (the input data may be in an ontology not recognised by the Web service). Next comes the creation of a choreography instance for the Web service in the Choreography Engine, which is updated with the instance data received with the Web service. Following the rules defined for the Web service choreography, the Choreography Engine raises an event for the Invoker interface of the Communication Manager to send the input data to the Web service. The Communication Manager is responsible to ensure that the correct message and transport protocols are used.

There are two possibilities when a Web service responds to an invocation. The first is the synchronous case where the CommunicationManager sends the invocation message and blocks until the response is received. As explained in section 3.3.8 in this case the lifting and lowering between XML and WSML is carried out by the CommunicationManager itself. The second case is where the Web service asynchronously responds to the invocation from the CommunicationManager. The Receiver interface of the CommunicationManager is used and expects the content of the incoming message to be in WSML. That means that the Web service must be able to support the creation of WSML messages (either directly or through an adapter) for asynchronous communication to take place.

4.3 Combined Goal-based Discovery and Invocation Process

Figure 4.3: Goal-based Discovery and Invocation Activity Diagram

Figure 4.3 shows a UML Activity Diagram for handling the conversation between a client service (providing the goal) and a business service (offered as a Web service). As in the other diagrams, a WSML message is received by the Communication Manager and then parsed. The execution semantics for this process then determines if the message contains a WSML goal. If this is the case, discovery is required and the goal is handled by the service and QoS discovery activities respectively. The Process Mediator is provided with the goal and Web service choreographies (from their
WSMO descriptions) and any instance data provided with the initial WSML message. It determines if process and data mediation is required before instructing that the Choreography Engine create and update the choreography instances for the goal and Web service (for this instance of the execution semantics). The Choreography Engine uses its rules to determine if any message needs to be sent to either the client service (goal owner) or the business service (Web service provider).

The control branch after the Parse activity is for the case where the WSML message received is part of an existing conversation. It may have come from either the client service or the business service. In both cases, there is no need for the discovery activities and so control moves to the process and data mediation activity.
5 Usage Viewpoint

The usage viewpoint is intended to provide detail on how the architecture is used in the context of the DIP case studies. Additionally it provides a brief overview of IRS-III as a second implementation of the DIP architecture and describes the integration points between the WSMX prototype and IRS-III.

This chapter addresses the following concerns:

- What parts of the architecture are used by the case studies.
- What is the invocation sequence for components when the prototype is used to address the case studies.
- Overview of IRS-III as second implementation of the DIP architecture and its integration points with the WSMX prototype.

In particular, the viewpoint addresses the needs of three of the stakeholder groups. System owners and System acquirers gain information on the operation of the prototype implementation of the architecture and how it uses the various functional components. Designers of systems that require a semantic middleware layer, to enable the usage of Semantic Web services, get a walk through example of how the description of Goals, Mediators and Web Services can be used. They also see how the architectural components can be used to decouple service requesters from providers in projects that require partners to be integrated over the Web.

5.1 Usage of the architecture in the DIP Case Study Prototype

As the usage of the architecture is similar across the case studies, despite the different domains of interest for each, we pick the stock trading scenario from the eBanking Case Study as a representative example. The following subsections provide an overview of the case study, specify which components of the architecture are used and then provides a UML sequence diagram to step through how the architecture is used to enable the required functionality.

The DIP prototype is also a participant in the Semantic Web Services Challenge. Its application to the challenge’s B2B case-study has been published in [?].

![Figure 5.1: eBanking Case Study Overview](image-url)
Figure 5.1 shows an overview of the DIP eBanking prototype for a stock broking application. To remain competitive in their markets, banks require the means to be adaptive to the changing needs of their clients. The hard-wired nature of many distributed computing systems makes it difficult to provide this flexibility at the business-to-customer (B2C) interface of the bank. The stock broking scenario allows bank clients to specify their request, in their own language, at the bank’s website (1). This request is analysed by the WSML adapter which attempts to match it with a WSML Goal from a pre-defined goal repository (2). The Goal is dispatched to the DIP prototype which carries out reasoner-based discovery, QoS-based selection, mediation, choreography, (possibly orchestration, if the Goal consists of multiple sub-goals), and invocation (3)4. Assuming a suitable service is identified, it is invoked and the result returned via the DIP prototype and the WSML adapter to the Stock broker website(5)(6)(7).

Note that, in this implementation, the eBanking adapter carries out control and data flow in addition to the transformation to and from WSML. This is a specific implementation required for this case study. It contains significant additional code, for practical reasons, that would be usually fulfilled by the orchestration component of the DIP architecture.

![Figure 5.2: Components Used for eBanking](image)

The architecture components used by the eBanking prototype are indicated in 5.2 with a red background colour. A mockup orchestration component was used that did not follow the DIP API, hence the light-red shading.

Note that the WSMORegistry interface (see 3.3.8) exposes methods that allow instance of the DIP architecture to act as a repository of WSMO description. The WSMO Studio\(^1\) is an ontology management tool that uses this interface to manage WSMO descriptions in this way.

Figure 5.3 shows a UML sequence diagram of calls to DIP components used by the eBanking prototype for the scenario of buying stock when a certain recommendation is received. The business rules to buy stock (or not) are implemented by the WSML Adapter.

### 5.1.1 Sending Request.

A Web page front end allows bank clients to enter that they wish to purchase a certain stock if the expert recommendation is 'Buy'. The adapter transforms the

\(^1\)http://www.wsmostudio.org/
natural language request into a WSML Goal using the repository of predefined Goals as a guide. The inputs for the Goal are provided by the client’s inputs on the Web page and the information gathered from a call to a Stock Adviser which returns a stockRecommendation. Note that the acquisition of a recommendation is achieved through Goal-based invocation but is not shown because the diagram would become too cluttered. The inputs, for buying stock, are lifted to WSML by the adapter using XSLT, a fragment of which is shown in listing 5.1. With the WSML Goal available, the adapter invokes the achieveGoal() entry point of the DIP architecture using the SOAP document-style message protocol. In return, a context is received containing the identification of the conversation as well as an identification of the role of the sender (i.e. requester or provider). This information is used in subsequent asynchronous calls from the requester.

```
1 <xsl:template match = "STOCK" />
2 <![CDATA[instance]]><xsl:apply-templates select = "CODIGOJSIN" />
3 <![CDATA[memberOf sm#Stock]]>
4 <xsl:apply-templates select = "NOMBRE_VALOR" />
5 <![CDATA[]]>
6 <xsl:apply-templates select = "CODIGOJSIN" />
7 <![CDATA[]]>
8 <xsl:apply-templates select = "COD_MERCADO" />
9 <xsl:apply-templates select = "FECHA_COTIZACION_COMPACCIONES" />
```
5.1.2 Discovery and Conversation Setup.

The AchieveGoal entrypoint is implemented by the Communication Manager – the middleware service, which facilitates the inbound and outbound communication with the DIP environment. After receipt of the goal, the Communication Manager initiates the execution semantics which manages the whole integration process. The Communication Manager sends the WSMO goal to the instance of the execution semantics, which in turn invokes the Parser returning the Goal parsed into an internal system object.

The next step is to invoke the discovery middleware service in order to match the requested capability of the Goal with the capabilities of services registered in the Repository. Services matching the requested capability are returned from to the execution semantics. For space reasons, we show a simplified discovery without selection. For the purposes of the diagram, we assume that exactly one service is discovered that matches the Goal capability.

Once a service is discovered, the execution semantics registers both the requester’s as well as the provider’s choreography with the Choreography Engine (these choreographies are part of the goal and service descriptions respectively). Both choreographies
are set to a state where they wait for incoming messages that could fire a transition rule. This completes the conversation setup.

5.1.3 3 – Conversation with Requester.

The instance data for the goal is sent from the BankInter adapter to WSMX asynchronously by invoking the receiveData entrypoint. Along with the instance data, the context is also sent to WSMX in order to identify the sender and the conversation (the context has been previously obtained as a result of invocation of the achieveGoal entrypoint).

The data in WSML (WSMLmsg) is passed through the Communication Manager to the execution semantics which again first parses the data into internal system objects using the Parser. In general, multiple independent conversations can be running inside WSMX, thus information carried by the context is used to identify the specific conversation to which the message belongs. The execution semantics then passes obtained data to the Process Mediator.

The first task of the Process Mediator is to decide, which data will be added to which choreography, i.e. requester’s or provider’s choreography. This decision is based on analysis of both choreographies and concepts used by these choreographies and is described in detail in [3]. In our scenario, Process Mediator first updates the memory of the requester’s choreography with the information that the Purchase Order request has been sent. The Process Mediator then evaluates how that data should be added to the memory of the provider’s choreography. In the use case, data mediation must be first performed to the ontology used by the provider (service ontology). For this purpose, the source ontology of the requester, target ontology of the provider and the instance data are passed to the Data Mediator. Data mediation is performed by execution of mapping rules between both ontologies (these mapping rules are stored within WSMX and have been created and registered during the integration setup phase). More information on the design and implementation of the Data Mediator can be found in [?]. Once mediation is complete, the mediated data is added to the provider’s choreography.

5.1.4 4 – Conversation with Provider (buying stock).

Once the requester’s and provider’s choreographies have been updated, the Choreography Engine processes each to evaluate if any transition rules have been fired. In our scenario, the requester’s choreography remains in the waiting state as no rule can be evaluated at this stage. For the provider’s choreography, the Choreography Engine finds the rule shown in the listing 5.2 (lines 14-21). Here, the Choreography Engine matches the data in the memory with the the antecedent of the rule and performs the action of the rule’s consequent (i.e. update/delete of the memory). The rule says that the message buyStock with data, stockId, volume, should be sent to the service provider (this data has been previously added to the choreography memory after the

\[\text{Note that the choreographies of WSMO services are modeled as Abstract State Machines}\] and are processed following the semantics of Abstract State Machines. Each choreography has its own memory containing instance data of ontological concepts. A choreography rule (ASM rule) whose antecedent matches available data in the memory is selected from the rule base and by execution of the rule’s consequent, the memory is modified (data in the memory is updated, deleted or removed).
mediation. The Choreography Engine then waits for the `buyStockResponse` message to be sent as a response from the provider. Sending the message to the service provider is carried out by Choreography Engine passing the message to the Communication Manager which, according to the grounding defined in the choreography, further passes the message to the `purchaseOrderConfirmation` entrypoint of the BankInter Adapter.

Listing 5.2: Provider’s Service Choreography

```plaintext
... choreography BuyStockChoreography
    stateSignature _"http://www.example.org/ontologies/BuyStockWS#statesignature"
    importsOntology {"http://www.example.org/ontologies/DIP/Bank",
                    "http://www.example.org/ontologies/choreographyOnto" }
    in bank#buyStock withGrounding { _"https://www.theBank.com/wsBrokerService/Service1.asmx?WSDL#
                        wsdl:interfaceMessageReference(BrokerServiceSoap/performBuySell/in0"
                        } ...
    out bank#buyStockResponse
    ... controlled ControlState
    transitionRules _"http://www.example.org/ontologies/buyStock#transitionRules"
    forall (?controlstate, ?request) with {
        ?controlstate[oasm#value hasValue oasm#InitialState] memberOf oasm#ControlState and
        ?request memberOf bank#buyStockRequest
    } do
        add(?controlstate[oasm#value hasValue oasm#BuyingStock])
        delete(?controlstate[oasm#value hasValue oasm#InitialState])
        add(# memberOf bank#buyStockResponse)
    endForall
...
```

Listing 5.2 shows the fragment of the provider’s choreography and the selected rule described above. The choreography is described from the service point of view thus the rule says that the service expects to receive the `buyStock` message and send the reply `buyStockResponse` message. In the `StateSignature` section (lines 3-11), concepts for the `input`, `output` and `controlled` vocabulary are defined. Input concepts correspond to messages sent to the service, output concept corresponds to messages sent out of the service, and controlled concepts are used for controlling the states, and transition between states during processing of the choreography. Each concept used is prefixed with the namespace definition (e.g. bank, oasm) corresponding to the imported ontologies (lines 4, 5). The choreography is part of the service definition which in addition also contains definition of non-functional properties and capability. For brevity, these elements are not included in the listing.

In the adapter, lowering the WSML message to XML is performed using transformation rules from the BankInter ontology to the corresponding XML Schema used by the Web front end.

After that, the actual service for buying the stock is invoked, passing the parameters of the `stockID` and `volume`. The service executes and sends its results in XML to the adapter. As before, the XML data is lifted to the Bank ontology, passed to the WSMX, parsed by the Parser and after the evaluation of the Process Mediator, the data is added to the provider’s and requester’s choreography memory respectively. Once the choreographies have been updated, the reasoning engine is used to check if any transition rules can fire as a consequence of the new information. In this case, there is only one further message to be sent. This is a confirmation message for the service requester that the stock purchase has been completed.
After the message is sent, no additional rules can be evaluated from the requester’s choreography, thus the choreography gets to the end of conversation state. Since both requester’s and provider’s choreography are in the state of end of conversation, the Choreography Engine notifies the execution semantics and the conversation is closed.

5.2 IRS-III and the DIP Architecture

The Internet Reasoning Service version 3, (IRS-III) is a framework and implemented platform which acts as a broker mediating between the goals of a user or client and available deployed Web services. It allows publication, composition, execution of different and heterogeneous Web services building upon the previous version, IRS-II. IRS-III uses WSMO to define the information model for Semantic Web services and has evolved into a second reference implementation for WSMO by implementing some of the functional components defined in the architecture and using the DIP API as the programmatic interface for accessing those components. The eMerges DIP case study [21] is implemented on both the IRS and WSMX platforms.

As described in [21], the IRS-III implementation of the WSMO conceptual model has been extended in a number of ways. These are:

- **Explicit input and output role declaration.** The capability descriptions of Web services and Goals are defined to include input and output roles, each having a name and a semantic type (defined ontologically).

- **Web Services are linked to Goals by mediators.** Goals are always associated with a mediator. If a wgMediator, associated with a Web service, has a Goal as its source, then the Web service is considered to solve the Goal. ggMediators provide data flow between sub-Goals. They are used to link sub Goals within an orchestration.

- **Web services can inherit from Goals.** A Web service linked to a Goal through a wgMediator is considered to inherit the input and output roles of the Goal. In such a case adding input and output roles to the Web service overrides the inherited values.

- **Client Choreography.** The provider of a Web service must describe the choreography from the perspective of the client.

- **Mediation services are Goals.** A mediator can declare a Goal as the mediation service. The Goal is then resolved to a Web service at run-time.

5.2.1 Integration Points for IRS-III with the DIP Architecture Prototype (WSMX)

Both IRS and WSMX are implemented frameworks for Semantic Web services based on the conceptual model defined by WSMO. This provides a strong basis for common ground between the systems. In terms of IRS, this meant defining system interfaces that could accept messages represented using the Web Services Modeling Language (WSML) [1]. This provided design-time integration in the sense that IRS and WSMX both are capable of exchanging WSML. Additionally, both systems used the same
open-source software to parse the messages and represent them internally as Java objects (WSMO4j[3]). This common internal information representation ensures common semantics in the mapping of WSML to the Java programming language used by both systems.

With both systems supporting message exchange using WSML, the next step was the adoption by IRS of Entry Point interface of the DIP API as a public interface to their functionality. This is run-time integration following the pattern exemplified by the odbc and jdbc specifications with respect to relational database technology. It means that a client of a Semantic Web services execution environment can be transparently configured to use either IRS or WSMX without needing to change the client code. This holds equally true for instances of WSMX and IRS themselves as clients of instances of each other (or another compliant system).

The following subsections provide a short overview of the functionality and signatures of the common API. We look at the interfaces providing access to the functionality of discovery, goal-based service invocation, direct service invocation and registry-invocation. For convenience, the interface method descriptions are given in the syntax of Java Standard Edition 5.0 (J2SE 5.0)[4].

Many of the methods return an instance of the Context class as defined in the API. The notion of Context is used to represent a unique handle for identifying the context of communication between the Semantic Web Services execution environment and another entity. For example, a client wishing to use the environment to perform a goal-based service invocation, would initiate a conversation with the environment by sending a message containing the description of the goal that they wish to achieve. On receipt of this message the environment would create a unique Context for that specific conversation and return it to the client. All subsequent interactions as part of the conversation must then include this Context. In this sense the Context is used as a shared token between the client and the environment to identify the state of the conversation on both sides.

**Goal-based Service Invocation**

The environment will carry out all activities required for service invocation (discovery, mediation etc.) based on the input of user goal defined in WSMO.

```java
public Context achieveGoal (WSMLDocument wsmlMessage)
```

**Discovery**

Discovery allows the client to find Web services that match a specific goal description. The method accepts as input a WSMLDocument object which should contain the WSML description of the goal. The method signature is:

```java
public Context getWebService (WSMLDocument wsmlMessage)
```

---


Direct Service Invocation

The client of the environment can invoke a Web service directly by sending a WSML document identifying the Web service and the data that is required for input to the Web service.

    public Context invokeWebService (WSMLDocument wsmlMessage)

Additionally the client can send a message to a Web service where a conversation has already been initiated (Context has been returned to client). in this case the client provides a WSML document containing the message to be sent to the Web service as well as the Context that identifies which conversation this action should be part of.

    public Context invokeWebService (WSMLDocument wsmlMessage, Context context)
6 Technology Viewpoint

The DIP architecture represents the design for a layer of middleware for the discovery, mediation, orchestration, and invocation of Semantic Web services based on ontological descriptions provided by elements of the Web Services Modeling Ontology (WSMO). In this chapter, we provide some details on the design for the Core component of the architecture (management of the platform services and the event-based messaging) and the technologies chosen to implement those designs in the WSMX prototype.

The stakeholders for this viewpoint are the owner and acquirer of the system (e.g. the technical lead of a project considering its adoption). The concern addressed is how can the core requirements for component management, inter-component messaging and configurable execution semantics be designed and implemented, taking advantage of the considerable existing middleware and application server technology.

6.1 Core Design

The Core component takes the role of component co-ordinator for the DIP architecture. It manages the interaction between the other components through the exchange of messages containing instances of WSMO concepts expressed in WSML and provides the microkernel and messaging infrastructure for the architecture. The component is responsible for handling the three main functional requirements:

- A framework for the management and monitoring to start and stop the system and allow its health to be monitored.
- The Core facilitates the communication between the components, it accepts messages and routes them to a suitable target component, enabling the communication between as well as the coordination of components.
- Support for the life cycle of Execution Semantics. Execution semantics specify the desired operational behaviour of the DIP architecture. Multiple definitions of execution semantics are supported and multiple instances of each execution semantics may run concurrently. This is described in section.

The architecture-prototype Core is realized as a kernel utilizing Java Management Extensions (JMX) as described by Haselwanter in [12].

6.1.1 Management

In common with middleware and distributed computing systems, management of components involved in the system becomes a critical issue. In the architecture we make a clear separation between operational logic and management logic, treating them as orthogonal concepts. If we would not separate these two elements, it would be increasingly difficult to maintain the system and keep it flexible. From a certain perspective it could be argued that the very process of making management explicit, captures an invariant that helps to leverage the support for dynamic change of the rest of the system. Figure 6.1 presents an overview of the infrastructure provides by the Core to the components, which allows to manage and monitor the system.
The Core is a management agent that offers several dedicated services, the most essential of which is perhaps the bootstrap service, responsible for loading and configuring the application components. The necessary information is obtained by a combination of reflection and supplied information in the form of a distributed configuration. The agent plays the role of a driver, in other words, it is built into the application, as opposed to a daemon approach.

The Core also employs self-management techniques such as scheduled operations, and allows administration through a representation independent management and monitoring interface that allows for a number of different management consoles, serving different purposes. Text-Terminal-, Web-Browser- and Eclipse-Framework-based consoles have been implemented.

As is common with middleware and application servers, the Core hosts a number of subsystems that provide services to components and enable inter-component communication. Besides systems that are responsible for communication matters, pool management takes care of handling several component instances, along with a number of the above subsystems. The Core is in the unique position of offering services to the individual components, such as: logging services, transport services and life cycle services. Presenting a coherent view of all management aspects of components and, at the same time, not getting lost in complexity are two conflicting goals and subject to compromise. Ideally a component’s manageability would be presented in a unified view that covers the core subsystem as well as the component’s instrumentation. For part of this view the Core exploits the underlying (virtual) machine’s instrumentation to monitor performance and system health metrics. Even though some manageability like the aforementioned metrics may be captured generically for all components, additional aspects of components may be specific to them and require custom instrumentation. If this necessary, it can be achieved by extending the configuration for the instrumentation of that specific component, independently of the implementation of the components itself.
One of the principles underlying the DIP architecture design is support for distribution. Although part of the Core infrastructure may act as facade to distributed components, the preferred way to distribution is to organize the system as a *federations of agents*. Each agent has its own Core component and a subset of the architecture’s functional components. It is possible to hide the complexity of the federation from the management application by providing a single agent *view*. In other words, provide a single point of access to the management and administration interfaces. This can be achieved by propagating requests within the federation via proxies, broadcasts or directories. From a management point of view an instance will consist of a set of cores, organized in federations, one kernel per machine, each of which may host a set of components. This strategy allows taking advantage of locality while keeping remoteness transparent.

### 6.1.2 Communication and Coordination

The architecture avoids hard-wired bindings between components; communication is based on events. That is, if some functionality is required then an event that represents that request is created and published. Any components subscribed to this event type can fetch and process the event. The event-based approach naturally allows asynchronous communication. As illustrated in Figure 6.2 events exchange is conducted via a Tuple Space (originally demonstrated in Linda [7]), which provides persistent shared space enabling interaction between parties without direct event exchange between them.

![Figure 6.2: Communication and Coordination in the DIP Architecture Core](image)

The Tuple Space enables communication between distributed units running on remote machines or on the local machine. It should be emphasised that the functional components themselves (discovery, mediation etc.) are completely unaware of this distribution. That is, an additional layer provides them with a mechanism of communication with other components that shields them from the actual mechanism of transport which can be local or remote.

Interaction is carried out by exploiting a publish-subscribe mechanism. Subscription is specified by utilizing templates and matching them against tuples available in
A Tuple Space may be possibly composed of many distributed Tuple Space repositories that need to be synchronized. In order to maximize usage of components available within a local machine, instances of distributed Tuple Space are running on each machine and newly produced entries are published locally. Before synchronization with other distributed Tuple Spaces takes place, a set of local template rules needs to be executed in order to check if there are any local components subscribed to the newly published event type. That is, if not otherwise specified, local components should have priority in receiving locally published entries.

Through the infrastructure provided by the Core, component implementations are separated from communication issues. This infrastructure is made available to each component implementation during instantiation of the component carried out by the Core during the bootstrap process. This is the process that occurs when a component is identified and loaded by the system. Through the use of JMX and reflection technology in the prototype, this can occur both at start-up as well as after the system is up and running. In particular the communication infrastructure carries the responsibility to interact with the transport layer (a Tuple Space instance). Through the transport layer, component subscribe to an event-type template. Similarly, they publish result events in the Tuple Space.

To enable a component to request functionality from another component a proxy mechanism is utilized. The component currently being executed might need to invoke other components functionality and does so via a proxy, which creates the proper event for this situation and publishes it on the Tuple Space. The proxy subscribes to the response event and takes care of the correlation. From the perspective of the invoking component, the proxy appears as the component being invoked. This is the same pattern as used by remote method invocations (RMI) in object-oriented distributed systems.

6.2 Execution Semantics

Execution Semantics enable the combined execution of loosely-coupled components as illustrate in Figure 6.3 and provide the necessary execution logic to functionally
realize the system. The Core infrastructure provides the implementation that enforces the general computation strategy by enforcing the Execution Semantics, operating on the transport as well as the component interfaces. It takes events from the Tuple Space and invokes the appropriate components. Local component functionality is invoked by the Execution Semantic, while keeping track of the current state of execution. Additional data obtained during execution can be preserved in the particular instance of an Execution Semantic.

Without tying itself to a particular Execution Semantic, the Core provides the framework that allows Execution Semantics to operate on a set of components without tying itself to a particular set of implementations. The Core provides the means for Execution Semantics by taking care of their life cycle, allowing to manage them, and monitor them during their execution.

Each execution semantics can be considered to represent a process. Figure 6.4 depicts a simplified process for goal-based discovery and subsequent invocation of a Web service. The ovals in the diagram represent the steps in the execution semantics for this definition of the discovery process.

![Figure 6.4: Sample Abstract Execution Semantics Definition](image)

Each middleware service has associated configuration metadata which allows the Core to pick up that service and generate wrappers for it as its deployed. Figure 6.5 shows the introduction of wrappers and how it is these wrappers that are invoked by the implementation of the execution semantics rather than the services themselves. The wrappers are generated and managed by the Core to separate middleware services from the transport layer for events. One wrapper raises an event with some message content and another wrapper can at some point in time consume this event and react to it. Developers of individual middleware services do not need to be concerned about how this happens. This is significant in that the design and development of these services needs only a minimal knowledge of the messaging mechanism for DIP. From their perspective, they appear to directly invoke other components even though
the architecture takes over all responsibility for messaging. The services are never explicitly aware of the wrappers and never invoke them directly.

Finally, figure 6.6 depicts how services are decoupled from the process (described in the execution semantics). Based on an execution semantics definition, these wrappers will only be able to consume and produce particular types of events. In a running system dynamic execution semantics are achieved by mapping abstract system behavior into real event infrastructure of the system.
Figure 6.6: Event SOA for WSMX
7 SUMMARY

In this document we described the architecture of the current version of the DIP architecture. The document follows the recommendations on using architectural views of IEEE Recommended Practice for Architectural Descriptions of Software-intensive systems. The deliverable identifies stakeholders in the architecture and concerns that these stakeholders are likely to have. Four viewpoints, or perspectives on the architecture are presented. In each case, the stakeholders and concerns the description of the viewpoint addresses, is made clear.

We focused on how the DIP architecture has evolved as a Service Oriented Architecture (SOA) from earlier versions. The adoption of the principles of component decoupling and standardized component interface definitions provide DIP with flexibility as the implementations of functional components are decoupled from the definition of the possible behaviour (execution semantics) of the system.

In particular, in the services viewpoint, we have provided an overview of each the various services involved with the architecture including a UML component diagram showing each components’ interfaces and dependencies. In the section describing the process viewpoint we gave a brief description of the control flow of the three process defined for the architecture, using UML activity diagrams, and pointed to the D6.16 deliverable where these are fleshed out as execution semantics.

The API for the DIP architecture is documented based on the work in the open-source WSMX project which is a prototype implementation for DIP. The API is documented using Javadoc based on the Java definitions of the component and system interfaces. A zip file containing the API has been submitted to the BSCW document management system used by DIP and to the WSMX prototype homepage and is available through the link included in the relevant section of this deliverable.

The chapter on the usage viewpoint took one of the DIP case studies and walked through how the WSMX prototype implementation was used to realise the required functionality. Finally the chapter on technology provided more in-depth information on technologies used in the prototype to make the service-oriented DIP architecture a reality that can be picked up and used as the platform architecture for ongoing EU research projects, as well as the basis for the work of the OASIS Semantic Execution Environment Technical Committee.
BIBLIOGRAPHY


