DIP
Data, Information and Process Integration with Semantic Web Services

FP6 – 507483

Deliverable

WP6: Interoperability and Architecture

D6.2

DIP Architecture

Manfred Hauswirth (EPFL), Roman Schmidt (EPFL), Michael Altenhofen (SAP),
Christian Drumm (SAP), Chris Bussler(NUIG), Matthew Moran (NUIG),
Michal Zaremba (NUIG), Laurentiu Vasiliu (NUIG), Joachim Quantz (inubit),
Laurent Henocque (ILOG), Armin Haller (NUIG), Brahmananda Sakpota (NUIG),
Edward Kilgariff (NUIG), Simeon Petkov (NUIG), David Aiken (NUIG),
Eyal Oren (NUIG), Manuel Ohlendorf (NUIG), Adrian Mocan (NUIG)

December 15, 2004
SUMMARY

This deliverable describes the first version of the DIP architecture. The goal of the architecture is to provide a high-level overview of the necessary system components and their interactions. Each component is described to the level of detail that is needed to understand its function in the architecture. For more detailed descriptions of the components the reader is referred to the respective component deliverables. Then the important interactions, i.e., workflows, to support typical scenarios in the processing of semantic web services are described. The description uses the external interfaces of the components which are standardized in the DIP API (deliverable 6.3). The goal of this deliverable is to provide understanding of the internal processes of DIP and support developers in providing components that can interoperate with other components in DIP. Additionally, users can get a high-level overview of the internal processing and thus get a better understanding of the system.

The contribution of this deliverable is to provide the basic description and the basic framework of the open web services architecture which is one of the three main goals of the DIP project. This is a key issue to enable interoperability. Thus this deliverable is relevant to all technical work packages in DIP (WP1–WP5). In conjunction with deliverable 6.3, this deliverable provides the interoperability guidelines for all components developed in DIP. The target audience are component providers, users, and any person inside or outside of DIP interested in learning about the internal processing of the DIP infrastructure.

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.
This deliverable describes the first version of the DIP architecture and provides a high-level overview of the necessary system components and their interactions. The components are described only to the level of detail necessary to understand the architecture. For more detailed descriptions of the components the reader is referred to the respective component deliverables. Then the important interactions, i.e., workflows, to support typical scenarios in the processing of semantic web services are described. The description uses the external interfaces of the components which are standardized in the DIP API (deliverable 6.3). The goal of this deliverable is to provide understanding of the internal processes of DIP and support developers in providing components that can interoperate with other components in DIP. Additionally, users can get a high-level overview of the internal processing and thus get a better understanding of the system.

Keywords: Open architecture for semantic web services, interoperability.
## Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>National University of Galway</td>
<td>NUIG</td>
<td>Prof. Dr. Christoph Bussler Digital Enterprise Research Institute (DERI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>National University of Ireland, Galway Galway Ireland E-mail: <a href="mailto:chris.bussler@deri.ie">chris.bussler@deri.ie</a> Tel: +353 91 512460</td>
</tr>
<tr>
<td>Fundacion De La Innovacion.Bankinter</td>
<td>Bankinter</td>
<td>Monica Martinez Montes Fundacion de la Innovation. BankInter, Paseo Castellana, 29 28046 Madrid, Spain Email: <a href="mailto:mmtnez@bankinter.es">mmtnez@bankinter.es</a> Tel: 916234238</td>
</tr>
<tr>
<td>Berlecon Research GmbH</td>
<td>Berelcon</td>
<td>Dr. Thorsten Wichmann Berlecon Research GmbH, Oranienburger Str. 32, 10117 Berlin, Germany E-mail: <a href="mailto:tw@berlecon.de">tw@berlecon.de</a> Tel: +49 30 2852960</td>
</tr>
<tr>
<td>British Telecommunications Plc.</td>
<td>BT</td>
<td>Dr. John Davies BT Exact (Orion Floor 5 pp12) Ipswich IP5 3RE, United Kingdom Email: <a href="mailto:john.nj.davies@bt.com">john.nj.davies@bt.com</a> Tel: +44 1473 609583</td>
</tr>
<tr>
<td>Swiss Federal Institute of Technology, Lausanne</td>
<td>EPFL</td>
<td>Prof. Karl Aberer Distributed Information Systems Laboratory École Polytechnique Fédérale de Lausanne Bât. PSE-A 1015 Lausanne, Switzerland Email: <a href="mailto:Karl.Aberer@epfl.ch">Karl.Aberer@epfl.ch</a> Tel: +41 21 693 4679</td>
</tr>
<tr>
<td>Essex County Council</td>
<td>Essex</td>
<td>Mary Rowlatt. Essex County Council, PO Box 11, County Hall, Duke Street, Chelmsford, Essex, CM1 1LX, United Kingdom. Email: <a href="mailto:maryr@essexcc.gov.uk">maryr@essexcc.gov.uk</a> Tel: +44 (0)1245 436524</td>
</tr>
<tr>
<td>Forschungszentrum Informatik</td>
<td>FZI</td>
<td>Andreas Abecker Forschungszentrum Informatik Haid-und-Neu Strasse 10-14 76131 Karlsruhe Germany E-mail: <a href="mailto:abecker@fzi.de">abecker@fzi.de</a> Tel: +49 721 96540</td>
</tr>
<tr>
<td>Institution</td>
<td>Address</td>
<td>Contact</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Institut für Informatik, Leopold-Franzens Universität Innsbruck</td>
<td>UIBK Technikerstr. 25 A-6020 Innsbruck, Austria</td>
<td>Prof. Dieter Fensel Institute of computer science University of Innsbruck Email: <a href="mailto:dieter.fensel@deri.org">dieter.fensel@deri.org</a> Tel: +43 512 5076485</td>
</tr>
<tr>
<td>ILOG SA</td>
<td>ILOG 9 Rue de Verdun, 94253, Gentilly, France</td>
<td>Christian de Sainte Marie E-mail: <a href="mailto:csma@ilog.fr">csma@ilog.fr</a> Tel: +33 1 49082981</td>
</tr>
<tr>
<td>inubit AG</td>
<td>inubit Torsten Schmale, inubit AG, Lützowstraße 105-106 D-10785 Berlin, Germany E-mail: <a href="mailto:ts@inubit.com">ts@inubit.com</a> Tel: +49 30726112 0</td>
<td></td>
</tr>
<tr>
<td>Intelligent Software Components, S.A.</td>
<td>iSOCO Pedro de Valdivia 10 28006 Madrid, Spain E-mail: <a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a> Tel. +34 913 349 797</td>
<td></td>
</tr>
<tr>
<td>The Open University</td>
<td>OU</td>
<td>Dr. John Domingue Knowledge Media Institute, The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK E-mail: <a href="mailto:j.b.domingue@open.ac.uk">j.b.domingue@open.ac.uk</a> Tel.: +44 1908 655014</td>
</tr>
<tr>
<td>SAP AG</td>
<td>SAP Vincenz-Priessnitz-Str. 1 76131 Karlsruhe, Germany E-mail: <a href="mailto:elmar.dorner@sap.com">elmar.dorner@sap.com</a> Tel: +49 721 6902 31</td>
<td></td>
</tr>
<tr>
<td>Sirma AI Ltd.</td>
<td>Sirma Atanas Kiryakov, Ontotext Lab, - Sirma AI EAD, Office Express IT Centre, 3rd Floor 135 Tzarigradsko Chausse, Sofia 1784, Bulgaria E-mail: <a href="mailto:atanas.kiryakov@sirma.bg">atanas.kiryakov@sirma.bg</a> Tel.: +359 2 9768 303</td>
<td></td>
</tr>
<tr>
<td>Tiscali Österreich GmbH</td>
<td>Tiscali</td>
<td>Dr. Dieter Haacker Tiscali Österreich GmbH. Diefenbachgasse 35, A-1150 Vienna, Austria E-mail: <a href="mailto:Dieter.Haacker@at.tiscali.com">Dieter.Haacker@at.tiscali.com</a> Tel: +43 1 899 33 160</td>
</tr>
<tr>
<td>Unicorn Solution Ltd.</td>
<td>Unicorn</td>
<td>Vrije Universiteit Brussel</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>---------------------------</td>
</tr>
<tr>
<td></td>
<td>Jeff Eisenberg, Unicorn Solutions Ltd, Malcha Technology Park 1, Jerusalem 96951, Israel, E-mail: <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a>, Tel.: +972 2 6491111</td>
<td>Carlo Wouters, Starlab- VUB, Vrije Universiteit Brussel, Pleinlaan 2, G-10, 1050 Brussel, Belgium, E-mail: <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a>, Tel.: +32 (0) 2 629 3719</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Introduction .............................................. 1

2 Architecture Overview .................................... 2
   2.1 Web Services vs. Semantic Web Services ............... 2
   2.2 Discovery ........................................... 4
   2.3 Registry ............................................ 5
      2.3.1 Publishing ...................................... 6
      2.3.1.1 Requirements for Publishing .................. 6
      2.3.1.2 Information Relevant for Publishing .......... 7
      2.3.1.3 Publishing Service GUI ....................... 8
   2.4 Mediation .......................................... 8

3 Rationale of the architecture ............................. 10

4 Components ............................................. 11
   4.1 Overview .......................................... 11
   4.2 Configurations ..................................... 12
   4.3 WSMX/DIP Architecture Entry Points .................. 13
   4.4 Communication Manager .............................. 14
      4.4.1 Sending Messages from DIP/WSMX ............... 14
      4.4.2 Receiving Messages into DIP/WSMX .............. 14
      4.4.3 Grounding ...................................... 14
   4.5 QoS Manager ....................................... 15
      4.5.1 Security ....................................... 15
   4.6 Message Adapter ................................... 16
   4.7 Execution Manager .................................. 16
   4.8 Resource Manager .................................. 17
   4.9 Validator .......................................... 17
   4.10 Event Manager ..................................... 17
   4.11 Matchmaker ....................................... 18
   4.12 Mediation ........................................ 19
      4.12.1 Data Mediation ................................ 19
      4.12.2 Process Mediation .............................. 20
   4.13 Orchestration and Choreography ...................... 20
   4.14 Composition ....................................... 21
   4.15 Discovery ......................................... 22
   4.16 Supporting components ............................. 23

5 Sequence diagrams based on use cases .................... 24
   5.1 Discovery .......................................... 24
   5.2 Mediation .......................................... 24
   5.3 Service Execution .................................. 25
   5.4 Security .......................................... 31

6 Summary and conclusions .................................. 32

---

Deliverable 6.2  vi  December 15, 2004
1 INTRODUCTION

This deliverable describes the first version of the DIP architecture. The goal of the architecture is to provide a high-level overview of the necessary system components and their interactions, to provide understanding of the internal processes of DIP, and to support developers in providing components that can interoperate with DIP and its components. Additionally, users can get a high-level overview of the internal processing and thus get a better understanding of the system. The contribution of this deliverable is to provide the basic description and the basic framework of the open web services architecture which is one of the three main goals of the DIP project. This is a key issue to enable interoperability. Thus this deliverable is relevant to all technical work packages in DIP (WP1–WP5). In conjunction with deliverable 6.3, this deliverable provides the interoperability guidelines for all components developed in DIP. The target audience are component providers, users, and any person inside or outside of DIP interested in learning about the internal processing of the DIP infrastructure.

The deliverable is structured as follows: We start with an overview of the architecture by presenting a bird’s eye view on the system. We discuss the basic roles, their functions, and high-level relationships. Additionally, we relate the DIP architecture to the standard “triangle model” of traditional web service architectures to exemplify the commonalities and differences. This is followed by a presentation of the basic ideas underlying the architecture (rationale) to make our design decisions understandable to the reader.

Having set the initial frame, we then give more in-depth presentations of the components in the architecture. Each component is described to the level of detail that is needed to understand its function in the architecture. For more detailed descriptions of the components the reader is referred to the respective component deliverables. Based on these descriptions we then describe the important interactions, i.e., workflows, to support typical scenarios in the processing of semantic web services. The description uses the external interfaces of the components which are standardized in the DIP API (deliverable 6.3). In the final section we then present a condensed discussion of our findings and present our conclusions.
2 Architecture Overview

2.1 Web Services vs. Semantic Web Services

The “triangle” architecture as shown in Figure 2.1 has become the widely accepted architectural standard for conventional web services.

![Figure 2.1: The triangle architecture for web services](image)

Providers publish their available web services in a registry (typically UDDI). Requesters can connect to a registry and query it for web service descriptions. If a desired service is found, the requester can implement code which matches the published interfaces of the service, connect to the provider and invoke the service. All these interactions are based purely on syntactic properties and do not involve semantics at all.

Though this architecture serves many application scenarios, it has shown a number of shortcomings. In this “old” architecture a requester has to know registries, contact them individually and query them until it would find a service that would match its request. Queries are constrained to “syntactic” queries and exact matches. The registries do not offer semantic descriptions and semantic query functionalities. Thus discovery of appropriate services is merely based on the “semantics” inferred from proper naming conventions and is a cumbersome task which requires human intervention and knowledge.

DIP intends to extend this standard “triangle” architecture of web services (requester, provider, registry) with semantics (ontologies, tools for description, semantic discovery, composition, and mediation, etc.) and extend it where appropriate or required. By basing the DIP architecture on the old architecture we also try to insure the largest possible degree of backward compatibility, so that, web services and semantic web services can co-exist on top of the same infrastructure and we also support a gradual evolution from web services into semantic web services.

Figure 2.2 provides a high-level view of the DIP architecture in terms of the involved parties and their communication relationships.

Naturally requester, provider, and registry are still in the picture as DIP is intended to be a superset of the existing web services architecture. To denote them as the “old” components they are shown in light-blue (light-gray) along with the “old” communication relationships in the same color. Also only the “new” interactions are labeled as the other ones are the same as in Figure 2.1. To denote that “semantics” is the common underlying enhancement a light-gray semantics box is shown in the background of the figure.
It is important to note, that, though these components are “old”, their internal architecture and functionality may be quite different from the current (again, a superset). This means that in the “new” architecture the components may be enhanced with semantic capabilities based on WSMO/WSML. The new components are discovery and mediation along with semantic descriptions of web services given in WSMO/WSML. The external communication is still compatible with the existing protocols as long as requester and provider use traditional web services. However, if the requester invokes a semantic web service then additional processing has to happen which is indicated by the additional dark-blue (dark-gray) arrow between requester and provider and the additional dark-blue (dark-gray) components attached to requester and the provider.

We added the discovery component for several reasons. The simplest one is compatibility with the old architecture and re-use of existing UDDI registries. The more strategic one is that we wanted to have a clean separation of data (registry) and processing (discovery) to keep the discovery process flexible. In a world-wide network with a very high number of players it is essential that the discovery process scales and meets the specific requirements of different user groups. Thus tailored discovery strategies are likely to surface as well as we did not want to exclude new technologies. This separation of concerns enables us, for example, to use, standard discovery technologies, agent-based technologies, or peer-to-peer- based technologies at the same time, depending which one will meet user requirements best. The same holds true, if new technologies and advances in the domains of semantics and reasoning should be included. The discovery service again may be offered as a semantic web service.

In contrast to the old architecture the discovery of DIP will improve the situation in several ways. It will support discovery which is (1) semantic, (2) goal-oriented, and (3) distributed. Phrased differently, the requester can define (semantic) capabilities and attributes of the services it is interested in (the goal) and the discovery component will then start a distributed discovery process involving a possibly very large set of registries to find matching services (where match is to be seen rather broad since it may involve reasoning, etc.).

If no “exact” matches (in the semantic sense) can be found, mediation can be exploited to address this, which was the motivation for the second new component.
Basically the task of the mediation component is to mediate between the needs of a component (e.g., the requester) and the offered functionalities of the provider on different levels (data, process, protocol). This functionality can again involve discovery and may again be offered as a (semantic) web service.

Since discovery and mediation are new components we will provide more detailed descriptions in the following sections.

In the course of introducing semantic web services we envision three levels of functionality:

**Non-semantic level:** The standard functionality with the “old” components, “old” communication relationships and “old” protocols.

**Basic semantic level:** The same components but already enhanced with semantics, i.e., the registry already stores semantic descriptions of the web services, a requester can already discover web services based on their semantic description, etc. WSMO/WSML are the underlying language standards.

**Full semantic level:** The fully-fledged architecture with mediation and sophisticated goal-oriented discovery. WSMO/WSML are the underlying language standards.

### 2.2 Discovery

Discovery in DIP is developed as part of WSMO: *WSMO Web Service Discovery* [3] defines the methodological framework and *WSMO Discovery Engine* [2] describes discovery in a more implementation-oriented view. As we are early in the project, not all parameters could be fixed yet. Thus the following descriptions give an overview of discovery but are subject to change to adjust to the final version of discovery in DIP.

In DIP semantic Web service discovery is based on the concepts introduced in [7]: An abstract service is a set of concrete services and is specified by a machine-processable abstract service description. As the name suggests, one can think of an abstract service as a suitable abstraction of a set of concrete services instances that share a well-defined set of characteristics expressed in the abstract service description. For example, there might be an abstract service that describes an exchange rate service. One of its concrete services may compute the exchange rate between Swiss Francs and Euros. Note that there might be more than one abstract service that has the same or least similar characteristics, but the way how service requesters need to interact with them may differ. This interaction pattern is part of a contract between the service requester and the service provider and constitutes the mutual (implicit or explicit) agreement of what the service does and may include, among other things, legal terms and conditions, and compensation methods in case of failures during service execution.

These observations lead to the idea to abandon concrete service discovery and limit the core discovery process to abstract service discovery. Choosing a particular concrete service requires a separate selection step that may also involve a sophisticated negotiation process. The abstract service discovery process is based on the concepts of goals and capabilities [8]: Service requesters use a goal to specify their service requirements whereas service providers describe their services through capabilities. These goals and capabilities will be expressed using a formal language, presumably WSML [5].
To find a set of matching abstract services, a service requester sends a query to the discovery component that contains his/her goal in WSML including a reference to the ontology that contains the concepts used in the goal. Goals may contain concrete instance values, e.g., the requester may ask for the exchange rate from Swiss Francs to Euros but the ontology may help the discovery component to abstract this to European currencies or may even drop this information and infers that the abstract goal is to find services that can return exchange rates for currencies.\footnote{We deliberately leave this fairly vague as we can envision different approaches, algorithms and reasoning techniques for matchmaking and expect service discovery to be one of the active research areas during the course of the project. We do assume, though, that any match found will fulfill those criteria that the requester considers essential.} The result of the core discovery step will be a set of WSMO service descriptions representing acceptable services each tagged with a “matching score” similar to those defined in [6].

A selection step will then need to be performed to choose the “best” abstract service from the result set. Once the “best” has been found, the service requester may then perform a negotiation with the service provider to agree on a contract, and thus, on a concrete service it will eventually execute.\footnote{In the most simple case, invoking a concrete service will implicitly determine the contract.} To find this “best” service one can apply different techniques ranging from simple hard-wired selection criteria (e.g., “always the first”), to quantifying functional, non-functional preferences or even human interaction. Thus, we do not consider the selection step as part of the discovery component but rather see it handled by a sub-component that resides on the service requester side.

2.3 Registry

In principle, there are two approaches towards implementing a registry for semantic web services. On the one hand, one could take UDDI, the registry standard used by traditional web services, as a basis and extend it with semantic descriptions. On the other hand, one could base the registry on an ontology repository, as WSMO provides a homogeneous framework for the (semantic) description of semantic web services. DIP will start by following the latter approach, as it is closer to the philosophy underlying WSMO. Using an ontology repository as the basis of the registry allows to fully leverage the potential of semantic descriptions. For one thing, service descriptions can be easily combined with the domain model, since both use the same representation language. Moreover, inference engines can be integrated into the repository to check consistency and derive additional information from the available data. Finally, a generic query language for WSML and/or RDF can be used for querying the registry. The architecture will be designed flexible enough, however, to also support a UDDI-style registry. Whether such a registry will be actually implemented in DIP will be decided during the second project year. On the one hand, experiences with the registry based on an ontology repository will be evaluated. In addition, development in UDDI will be closely monitored. There is some discussion on a “Semantic UDDI”, e.g. by adding an “rdfBag” to UDDI, in the OASIS Technical Committee responsible
for UDDI. Finally, requirements arising in the use cases may call for a UDDI-based registry.

2.3.1 Publishing

2.3.1.1 Requirements for Publishing

In the context of (semantic) web services, the general idea of publishing a service is to collect all information needed or useful for its usage and management. As the fundamental idea of service-oriented architectures is that services are entirely defined by their interfaces and the contracts they offer, these are the minimally required pieces of information which have to be provided when publishing a service. For traditional web services, the interface is published via WSDL and additional information regarding service contracts is provided via free text or by using proprietary sets of attributes. In WSMO, the concept web service has several properties enabling the formal representation of this information. For one thing, these are non-functional core properties based on the Dublin Core Metadata Standard, which can be used to describe all WSMO elements. Moreover, the concept web service has specific non-functional properties, such as “accuracy”, “financial”, “networkRelatedQoS”, “performance”, “reliability”, “robustness”, “scalability”, “security”, “transactional” and trust. See [8] for details. It is expected that in addition to these non-functional properties there will be also application-specific properties and therefore WSMO envisages extensions of the basic model of non-functional properties. Finally, a web service in WSMO has the property “capability” which enables the specification of preconditions, assumptions, post conditions, and effects. These properties are particularly relevant for discovery and composition of services. In general, the decision on which information to include when publishing a service should be based on the various use case scenarios in which this information is to be used. The most important use cases are the following:

- Discovery of services
- (Semi-)automatic composition of services
- Monitoring of services
- Tracking, auditing, reporting

The most obvious use of published information is discovery of services. This comprises two slightly distinct though related aspects. On the one hand, it involves the search for available services; on the other hand, it involves the detailed verification whether a found service exactly matches the requirements. Whereas the initial search might be based on rather generic criteria, the second step usually takes into account specific aspects of a contract offered by a service.

A related but more complex use case is the (semi-)automatic composition of services. Here the focus is on the automatic evaluation of information available on services. Whereas “manual” discovery is feasible on the basis of free text information, standardized and structured formats are needed to make automatic procession of published information possible.
Whereas the first two use cases focus on the phase before service invocation, information on services is also useful during service execution or after service execution. When monitoring service execution, information about services can be used to filter and select data. Thus monitoring can be restricted to services having certain properties.

Similarly, tracking, auditing, and reporting, which in general occurs retrospectively, i.e., after services have been executed, can use information about services. An example would be the comparison of actual execution times with quality of service guarantees. The latter would be part of the published contract and accessible via the registry, the former would be data stored in a database logging information about service executions.

It should be obvious that the choice of information to be included when publishing a service ultimately depends on the specific application scenario. The publishing process thus has to provide a generic framework that can be adapted according to the specific needs of individual applications.

2.3.1.2 Information Relevant for Publishing

When publishing a (semantic) web service, three types of information have to be specified:

- **Service Interface**: in the case of traditional web services this will be a WSDL file, in the case of semantic web services it could also be a WSMO definition.

- **Metadata**: in the case of traditional web services this will be free text, in the case of semantic web services it will be a formal description based on WSMO or Semantic Web formalisms such as RDF or OWL.

- **Registry**: in the case of traditional web services this will be the address of a UDDI registry, in the case of semantic web services it could also be a WSML or RDF store.

As specified in detail in D4.2, the DIP publishing service will publish WSMO-type semantic web services to a UDDI registry. However, in addition to this functionality the publishing service will also support different modes of publishing. The main rationale for this is to enable the usage of parts of the functionality provided by semantic web services - even in a setting where no full usage of WSMO is envisaged. As an example, consider the scenario in which metadata are to be used purely in monitoring of traditional WSDL-style web services.

The following will briefly describe the information provided in publishing in more detail. In doing so, different alternatives are presented, ranging from full WSMO support to scenarios limited to semantically annotated WSDL files.

**Service Interface**

For traditional web services, service interfaces are specified by using WSDL (Web Services Description Language). In WSMO, a web service is a complex concept which has properties such as “nonFunctionalProperties”, “capability” and “interfaces”. The concept “interface”, in turn, has properties such as “choreography” and “orchestration”. In both cases, the publishing service will require a URL pointing to the location of either the WSDL file describing the service interface or of the WSMO file containing the service definition.
Metadata

The metadata describes non-functional properties of the service. In the full-fledged WSMO approach, metadata will be part of the WSMO file describing the service. However, if the service interface is described via plain WSDL, metadata has to be specified separately. Even when using WSMO it might make sense to add additional metadata in publishing, which is not part of the WSMO-definition of the service.

In order to make the alternative to the WSMO-based publishing as lightweight as possible, a simple RDF-based approach will be supported. Thus, metadata associated with a service in publishing can be specified as a list of RDF statements.

Registry

For traditional web services, UDDI is used as a registry. The DIP publishing process will also support an UDDI registry as specified in detail in D4.2. In addition, it will also be possible to store metadata specified in the publishing process in an RDF store. The basic idea here is to use existing RDF functionality in implementing prototypes for discovery and monitoring.

2.3.1.3 Publishing Service GUI

On top of the Publishing Service which is accessible through a web service API specified in WSDL, it will also be possible to manually publish services through a publishing GUI. The GUI will support the manual specification of metadata by listing properties for which values can be specified.

In order to know which property values are to be collected, the GUI needs to know the ontologies to be used for semantic annotation. In addition to WSMO, these can be application specific extensions. The GUI will enable users to specify the ontologies to be used during publishing and provides fields to fill in property values accordingly.

2.4 Mediation

Mediation is needed whenever two components cannot communicate directly but need to interact. The reasons for components not being able to communicate can either be the usage of syntactically different data formats as well as semantically different ontologies to describe the data or the usage of different choreographies. Therefore, mediation is necessary on two levels in the DIP architecture: the data- and the process/protocol level. Data mediation is concerned with mediation on the data level, involving both syntactical and semantic differences, whereas process/protocol mediation is concerned with mediation of different interaction protocols. Note that these two levels of mediation can be performed independent of each other.

From an architectural viewpoint, data mediation can be seen as an isolated problem that can be performed by a component external to the requester and the provider component. In contrast to that, process/protocol mediation needs to deal with interaction patterns that are tightly coupled to the requester and the provider and thus, can only be handled inside the infrastructure deployed at the requester and provider side. Besides the functional separation into data and process/protocol mediation, the overall mediation process can also be separated temporally into a design-time and a runtime step. During the design-time phase necessary transformations between different ontologies or choreographies are created with user support. The created transformations
will be stored into a repository for execution during runtime. During runtime the transformations stored in the repository will be executed in order to solve data or process/protocol mismatches.
3 RATIONALE OF THE ARCHITECTURE

In the design of the architecture we followed a set of basic design principles which we overview below. Understanding the rationale behind the architecture will clarify design decisions and support the reader in understanding the architecture and its goals.

**Components are self-contained.** Making the components self-contained supports a clear *separation of concerns*. Each component has a well-defined functionality that other components can use and it does not depend on other components in the first place to fulfill this function. For example, the mediation component can mediate between incompatible services and can do so without having to use any other component given it has enough information to perform its task. Of course, this does not mean that components do not cooperate. For example, if the mediation component would need further services to perform its task, it certainly would cooperate with the discovery component to find them. Yet, it does not depend on discovery to fulfill its central task which is to mediate.

**Standardization of external behavior.** As for any distributed architecture, the goal of the DIP architecture is to standardize the *external interface and behavior* of the components, so that any component conforming to these specifications can be used. This enables third-party software vendors to supply components, it is prerequisite for standardization, and it facilitates conformance testing. The internal architecture and implementation of a component is usually not considered in standardization.

**Expressive data formats.** Having a standardized API for the components supports interoperability. To push the flexibility of cooperations even further, we have decided to add expressive data formats, i.e., languages that are processed at runtime, for example, WSMO and WSML. This allows us to keep the APIs rather simple and compact while the expressive power and flexibility of WSMO and other data formats provide a declarative way of expressing all relevant processing steps, parameters, and application requirements. This means that the DIP architecture is *data centric*.

**Functionality is provided via (semantic) web services.** A key design principle in DIP is that each available functionality should be offered as a (semantic) web service as well thus unifying the overall design and apply identical paradigms at different levels. This applies to both static and dynamic cooperations. For example, the interface of the mediation component will be a web service. If the mediation component now creates a certain mediator that others may want to reuse, it could provide this functionality as a stand-alone web service.
4 COMPONENTS

4.1 Overview

To work with semantic web services a node (requester, provider, etc.) needs to install a component infrastructure. Figure 4.1 shows the internal architecture of a single DIP node.

The components will be described in detail below and possible interactions will be presented in terms of UML interaction diagrams. However, a brief introductory tour will simplify understanding the architecture. All (incoming/outgoing) network communication is done through the communication manager. We assume that the line format is SOAP as it is the current standard. DIP SOAP messages in fact typically should carry WSML content. If a different format has to be used for backward compatibility, e.g., ebXML, a message adapter converts back and forth to/from WSML. In the following descriptions we assume that the standard message exchange format is WSML and we do not consider anything below this abstraction layer.

Before an incoming message can hit any component after having been handed to the communication manager, it is provided to the QoS component. The QoS component provides a simple framework into which all kinds of specialized QoS subcomponent can be plugged in. Those can ensure functional and non-functional types of QoS. For example, we see security as a typical subcomponent. It can check message integrity, authenticity, access rights and return its results to the communication manager via the QoS component. If the communication manager agrees, then a message not meeting certain QoS requirements can be abandoned (this depends on the local configuration and user preferences). If the message is OK, then it will be registered with the event manager for further processing (WSML format).

The event manager is the internal communication backbone of a node. The communication paradigm is event-based, i.e., components subscribe to the event manager
and can issue events. For example, if discovery would be needed a component can raise an event equipped with the necessary data, the discovery component would be notified, process the event and again return the result in an event which the original requester would be notified upon. The advantages of this event-based communication are: (1) Receivers and event generators are highly decoupled which allows to dynamically reconfigure the system (start and stop components at runtime) (2) multiple components can react to events (parallelization, multicast) and (3) events are a natural paradigm for workflows. To guide and coordinate the whole process we have included a special component, the execution manager.

The functionalities of the other components are straight-forward: the validator checks syntactic correctness of WSML documents, discovery is the interface to DIP’s discovery service, mediation is the interface to DIP’s mediation, in case it is required, and the matchmaker is in charge to decide which candidate services match the requirements and user preferences in order to select a concrete service to be called.

As a result of processing incoming requests, outgoing messages can be generated which follow the same processing paths as incoming ones.

Figure 4.1 defines the basic processing components. Yet, a number of “supporting” components are needed for maintenance, management, configuration, etc., for example, an editor for producing semantic web services definitions as given in Figure 4.1. Such supporting components communicate with the architecture as given in Figure 4.1 in the same way as other nodes, i.e., via WSML messages. Thus we have a uniform interface and a flexible way to add new supporting components.

The other parties of the overall DIP architecture as shown in Figure 2.2 may have a completely different architecture, for example, the registry or the discovery components, which however, is irrelevant as long as they obey the required APIs.

4.2 Configurations

Figure 4.1 shows a complete setup. However, not all components need to be present if their functionality is not needed in certain application scenarios. For example, if the set of semantic web services a node is allowed to communicate with is fixed, e.g., due to contracts, no discovery will be needed. Since we used event-based communication this means that no discovery component needs to be running. Thus, depending on the requirements, “light-weight” configurations can be used and deployed. The minimum configuration would include communication manager (CM), event manager, execution manager (EM), and matchmaker (MM). This configuration can be extended as needed, for example:

1. Minimum configuration: communication manager (CM), event manager, execution manager (EM), and matchmaker (MM)
2. (1) + QoS
3. (1) + message adapter (MA)
4. (2) + MA
5. (4) + validator (V)
6. etc.
4.3 WSMX/DIP Architecture Entry Points

Four mandatory entry points must be available in each instance of a WSMX/DIP system. WSMX is an implementation of the here proposed architecture. They also define the required functionality of a WSMX/DIP compliant system. By selecting a given entry point the predefined execution semantics of a given set of components is triggered. Execution semantics is the formal definition of the operational behavior of the system and it describes in a formal language how the system behaves. These four obligatory entry points enabling the execution of any of the four available execution semantics are:

`realiseGoal(Goal, OntologyInstance):Confirmation`

Any external entity, which expects to get its goal realized without back and forth interactions (communication) with the WSMX/DIP system, might wish to provide a formal description of a goal (in WSMO terms) and instance of an ontology. This quite simplified scenario assumes that the service requester knows even before service discovery all the data, which might be required by the service provider. WSMX/DIP selects and executes a service on behalf of the service requester. The service requester might receive a final confirmation but this step is not obligatory (many entities which might wish their goals to be realized by WSMX/DIP system might not have permanent addressing, so there is no possibility to make an asynchronous call back to them returning the final result of the service invocation).

`receiveGoal(Goal):WebService[]`

The ReceiveGoal entry point addresses a more realistic scenario, if a service requester might wish to consult WSMX to learn about services which satisfy its goal. In this asynchronous call, the service requester provides a goal and expects to get back a set of services.

`receiveMessage(OntologyInstance, WebServiceID, ChoreographyID):Confirmation`

Once the service requester knows the service which he wants to use, he must carry a back and forth conversation with the WSMX/DIP system to provide all the necessary data and to make the execution of this service feasible. By giving fragments of ontology instances (e.g., business documents such as catalogue items or purchase orders in a given ontology) and a reference to a service and a choreography (only if a choreography has been instantiated already), which should be used, it provides all data required by the service of the service provider.

`storeEntity(WSMOEntity):Confirmation`

The store entity entry points provides an administration interface for the system enabling to store any WSMO related entities (such as services, goals, ontologies) and making them available for other parties using WSMX/DIP system.
Additionally to these four entry points, we assume that WSMX/DIP provides an engine to support dynamic execution semantics enabling execution of any formal description of system behavior. In this way we can also define additional and not required by any implementation, functionality of the system.

Four entry points described in this section are fundamental to describe sequence diagrams, which we will presented in Chapter 5 (at this stage we have focused on providing sequence diagrams for the second and third option, as well as we have started already to work on addressing the first option as well).

4.4 Communication Manager

The communication manager (CM) is responsible for sending and receiving WSML messages to/from entities external to DIP or adapters representing such external entities. In this context, the term entity means any external system acting as a service requester as well as services that can be invoked by DIP. Adapters are expected to be used where the external entity cannot directly support communication using WSML messages and translation to another data representation is required. Translation from WSML to another data representation is often referred to as “lowering” while translation from another data representation to WSML is often referred to as “lifting”. The two central functionalities of the communication manager - sending and receiving messages - are described in the following.

4.4.1 Sending Messages from DIP/WSMX

The communication manager provides an interface to receive the WSMO description of the service to be invoked along with data to be sent in the message to the service. The data must consist of instances of concepts described in the ontology used by the service. Both the service description and data will be represented in WSML. The CM interprets the interface part of the WSMO service description to determine which binding is necessary to invoke the service or an adapter representing the service where one is required.

4.4.2 Receiving Messages into DIP/WSMX

The communication manager provides an interface to external entities to accept WSML messages. The WSML messages may represent a goal to be achieved by another DIP/WSMX instance or be a message corresponding to a choreography or orchestra- tion instance that already exists. From the perspective of the CM, the distinction is irrelevant. The CM accepts the message, handling any transport and security protocols used by the sending adapter.

4.4.3 Grounding

DIP extends web service technology by using semantic service descriptions. However, it must be possible for DIP to interact with existing web services, described with WSDL. To make this possible a semantic web service description for the service is created in WSMO. The choreography section of the WSMO description describes the interface
to the service including the concepts the service expects as inputs and outputs. When the service is invoked these WSMO instances of concepts, represented in WSML, need to be translated to and from the corresponding data structures specified in the XML types of the WSDL document for the service. This grounding takes place in the adapter framework at the boundary of the DIP architecture and is a syntactic translation from WSML to the XML types defined in the WSDL document. Adapters enable systems that do not represent data using WSML to interact with DIP. The mappings required for grounding need to be in both directions. Data sent from a web service corresponding to the XML schema of a WSDL document must be “lifted” to the corresponding WSML expected by DIP. Equally, data sent from DIP represented in WSML must be “lowered” to the corresponding XML schema types described in the WSDL document.

4.5 QoS Manager

The QoS manager handles envelopes of SOAP or SOAP-compliant messages such as security or reliability extensions. Incoming messages are forwarded by the communication manager and have to be checked against the QoS guidelines specifying the required and acceptable levels of QoS. If all constraints are fulfilled the message excluding envelopes is returned to the communication manager, otherwise a notification is returned. The communication manager uses the QoS manager before sending a message to add all required QoS envelopes following the QoS requirements.

In terms of reliable messaging the QoS Manager relies on already available web service technology, such as the WS-ReliableMessaging protocol promoted by IBM and Microsoft and WS-Reliability promoted by Sun. Both specifications describe a protocol that enables messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures.

The WS-ReliableMessaging protocol is described in a transport-independent manner with the availability of supporting web services via a SOAP binding. WS-Reliability on the other hand is only SOAP-based and defined as SOAP header extensions.

Both specifications depend upon other web services specifications for the identification of service endpoint addresses and policies. It complements with WS-Security, described below, to enable for a broad range of reliable, secure messaging options.

4.5.1 Security

A specific type of QoS are security issues. For security the architecture will use existing infrastructures and components. Security components register with the QoS manager and the QoS manager will call them as needed.

The registering will be done by enhancements of the SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The WS-Security specification provides the means to support multiple security token formats, multiple trust domains, multiple signature formats, and multiple encryption technologies.
4.6 Message Adapter

Adapters enable external systems to use their own message format to communicate with the system they are integrated with. The adapter integration (within the system they are designed to work with to provide a message in an understandable format for it) could be partially or totally. Depending on the system and business application, adapters can be used individually or grouped in clusters for more complex usage. The clusters are designed as frameworks where adapters can be plugged in or out according to the communication needs. On the conceptual level, an adapter transforms the format of a received message into the format understood by the targeted system. The transformation is concerned with the syntactical mapping of the messages formats, maintaining the semantics of the message unchanged. The semantic part is enforced by personalized ontologies that are used to build the outgoing message format.

4.7 Execution Manager

The purpose of the execution manager component is to take care of the coordination of activities of components available at run time of the system. The execution manager initializes the system startup and shutdown, schedules tasks to particular components, controls execution semantics, and reacts to any discrepancies from the expected behavior of the system or components (errors, exceptions, abnormal execution).

The execution manager initializes the system startup. It recognizes if the last shutdown of the system finished successfully and if not, it undertakes a recovery procedure. In case of normal startup, the execution manager initializes component threads, verifies available system resources, schedules maintenance procedures, etc. During the system shutdown, the execution manager returns resources, closes all open sessions with components (or passivates them), and takes care to preserve any information or data which might be required during the next system startup.

The execution manager takes care to manage internal workflow (process definition for components executions) of the system. It is responsible for coordinating the execution order of components either by executing hard-coded static execution semantics, or by executing a dynamic interpretable definition of the execution semantics. Regardless which approach is used, the definition of events can be derived directly from the definition of execution semantics, so another responsibility of the execution manager is to distribute events to particular listeners of components known to the system.

Finally, the execution manager maintains conversations with components. We recommend to maintain this conversation through WSMX specific connectors (called listeners, which are based on JMX technology) layering between the core system and each particular component, which offers functionality to the system. Listeners enable connections of components which might be running in different virtual machines on the same server or even on completely different remote systems. Components fulfill services requested by the execution manager, while it also takes care to monitor if components work properly (deadlocks, exceptions, etc).

The execution manager is the “heart beat” of the system.
4.8 Resource Manager

The resource manager (RM) provides an abstract interface which is responsible for managing a resource (a document) or a repository (UDDI). The RM manages objects received in requests or created during the execution of a service. The RM is expected to provide a “put” and a “get” function. This will enable a process or component on the “other side” of the RM to request a document or ask the RM to place a document into the repository. The resource or repository is accessed only by the RM within the architecture.

It is because of this clearly defined functionality that the RM is able to provide abstract interfaces enabling each component to be independent and modular within a dynamic system. The requester of any resource is not concerned with the repository but only with the RM; consequently the individual resources and/or repositories only interact directly with the RM providing the ability for components to be pluggable, independent and modular in a dynamic system. At the architecture level the resource manager accesses a resource or repository. It is not until the lower level details of implementation that the RM decides which repository shall be preferred over another.

4.9 Validator

The validator performs syntactic validity checks of WSML documents provided by the execution manager. It determines whether an WSML document can be processed and may convert it in its internal representation, for example, by compiling it.

The validator could make use of the WSMO API, which defines methods for validating a document against a certain WSML variant and returning a memory-model of the elements of the document. The WSMO API is complemented by a reference implementation, which includes a parser to check the validity of documents against a certain WSML variant.

The validator can either return an in-memory model of the elements of the validated document, or compile it into its internal representation, for instance into a relational database.

4.10 Event Manager

In distributed workflow systems built of web services the communication can be highly asynchronous and many things can happen in parallel. Thus it makes sense to base the internal communication on an event-based paradigm. Event-based communication is enabled by the event manager. Basically it means that components do not communicate directly via calls but can create events which then are communicated to all other components subscribed to this class of event. The execution manager, as the central component in control of the processing, is subscribed to all events by default. This communication paradigm makes the usual polling unnecessary and is thus more efficient. Additionally, it decouples communication from processing (loose coupling, separation of concerns) and enables flexible communication patterns (1:1, 1:n) and
facilitates flexible run-time architectures (components do not have to be present, can join/leave dynamically without having to stop the system, etc.)

An event consists of the following data: An unique identifier for the event, the type of the event, and the event state. The type of the event describes which component can handle the event and how it should react on it while the state-field contains what happens during the component is working, e.g., if an error occurs. Publish/subscribe systems can be classified in four main groups according to their subscription mechanisms. These are channel-based, subject-based, object-based (or type-based) and content-based. The channel-based approach is the most common one: events are routed from publishers to subscribers using logical channels. Channels are uniquely identified and transmit information from publishers to all subscribers of that channel. A publisher can send an event to many channels. This model is used by the CORBA event service, Enterprise JavaBeans with the Java Message Service (JMS) API, and the Publisher/Listener pattern in Java AWT. Every component, e.g., the discovery, mediator, matchmaker, WSML-parser and invoker implements an event listener. The execution manager starts every listener and an event scanner which always searches for new events in the event storage which were not touched and are not finished, i.e., event state should be something else then “finished”, “error” or “locked”, which means that the event is in use. Those new events are now handled over to the listeners which are responsible for them. The listeners change the status of the event and create a new one if there is something else to do. To generate new events, the components just have to create a new entry in the event storage so that the event scanner can read it. During this process, the history of every event should be stored in order to be able to reconstruct and track the events through the system.

4.11 Matchmaker

The matchmaker component performs service matching between one requested service and one available service.

A specific service will be selected based on a goal description ontology received from the potential customer and the service capability ontology stored in the local repository. In this case we assume that the capability ontology has already been received and stored locally.

The goal ontology describes the ontology of the required product/service and the rules (axioms) needed for accepting and purchasing the product/service. The capability ontology describes the ontology of the offered product/service and the rules (axioms) for which a product/service is available to be purchased.

The matchmaker component performs the matching between the goal ontology and the capability ontology, by matching the product/service ontologies and also by ensuring that the rules of the customer and the provider are fulfilled.

Before any matching is performed, mediation between the goal ontology and capability ontology might be needed. Also, in order to ensure that the rules are fulfilled for both ontologies, an ontology reasoner is needed.

In the case that there is no local match, the discovery component is invoked and also provided with the customer goal ontology.
The discovery component is responsible for searching and retrieving capability ontologies from potentially compatible web services, to be matched with the customer goal ontology. For searching and retrieving, there are several technologies available, e.g., P2P technology or triple space computing. The technology usage is to be determined on the implementation level.

4.12 Mediation

The mediation could be required at different levels to cope with the potential heterogeneity problems that may occur, in order to enable the overall functionality of the architecture. As a consequence, different mediation components will handle the problems for each level: data mediation, process mediation and protocol mediation. Protocol mediation is not in the scope of DIP.

4.12.1 Data Mediation

The data mediation module consists of two separate components: the design-time component and the run-time component. The design-time component offers support for specifying mediation operations at design-time of a system, whereas the run-time component responds to run-time mediation requests, for example, by other components. By this, only the run-time component is explicitly part of the DIP/WSMX architecture but its performance depends on the design-time component output.

The specifications for such a mediation module are provided in [4], covering both the design-time and the run-time phase. The design-time component includes a graphical user interface that enables the user to define a set of mappings in a semiautomatic manner. These mappings overcome the heterogeneity problems existing between the given schemas and they have to be made available in a persistent way in order to be used by the runtime component later.

The design-time component takes as inputs the ids (URIs) of two schemas and the actual data (source instance(s)) to be mediated. Its role is to identify the ontologies and to determine if it is aware of any identified similarities between the given schemas. If these tasks are successfully accomplished and all the information are available (by its own means, offline, or by discovering other parties that can provide them, online) the mediator returns the mediated data, in fact, the target instances.

As described in [4], the data mediator may be extended with auxiliary modules that are able to perform lifting/lowering of both incoming/outgoing XML schemas and instances to/from the ontology and ontology instance levels respectively. These extensions are required to overcome the different levels of representation used for the inputs of the mediator and for transforming the mediated data in the XML format required for particular computations.
4.12.2 Process Mediation

Semantic web services have choreographies that define the message exchange sequence that has to be obeyed when interacting with them. The choreography of a web service is its interface and the invoker has to satisfy it completely. Fundamentally, a web service defines precisely in what sequence it expects messages and in what sequence it returns messages. Receiving and sending messages can be arbitrarily intertwined. A web service requester, in turn, has to make sure that it sends and receives the messages as defined by a web service it requests. Otherwise there would be a mismatch and the web service communication would fail completely or cause inconsistent web service states.

In general, a web service requester can invoke several web services. The order of these different invocations are defined in an orchestration. The orchestration defines therefore the overall interaction between a web service requester and all requested web services.

If a web service is dynamically discovered it might be very well the case that its choreography does not fit the expected behavior of a web service requester as defined in the orchestration. In this case the web service can either not be invoked at all or the requester has to adjust the message exchanges in such a way that the choreography of the web service can be satisfied. The latter case is enabled by process mediation that mediates between the web service requester’s defined message exchange sequence and the invoked web service choreography. Any message mismatch (see [1] for details) must be overcome for the requester and a requested web service to match. Due to the nature of interfaces, it is not guaranteed that process mediation is possible in all cases.

For example, superfluous messages have to be discarded, missing messages introduced, or maybe the order of the message exchange has to be changed. In analogy to data mediation a process mediator is “in between” a provider and a requester in order to intercept the messages. The interception enables to change messages if necessary.

Process mediation is a concept that has been identified, however, the research into this concept and how to provide a working solution is still not clear and requires future work. In general, though, it is clear that a component has to be added to the architecture that implements this concept eventually.

4.13 Orchestration and Choreography

The component “Orchestration and Choreography” (ORCA) is responsible for the execution of orchestration as well as choreography as defined by WSMO.

Every web service has a web service interface. A choreography defines how to interact with the web service in terms of exchanging messages. A web service that requires only a single message can be seen as a special case. However, for ease of discussion in this document, this would translate into a very simple choreography, namely one, that only defines a single message exchange. This approach avoids to make a single-message web service a special case and all web services can be treated exactly the same way from an architecture viewpoint. The execution semantics would not have to make a distinction either.
As web services are invoked by messages and responses are returned to the invoker through messages, every interaction with a web service involves ORCA. Several types of messages within one choreography can be distinguished.

First, a web service is initiated by an initiation message. This initiation message creates an instance of the web service’s choreography. Since one web service can be initiated several times, several instances of its choreography might execute concurrently. An instance of a choreography represents a context in which subsequent messages are exchanged. Subsequent messages are the second type of message. They are characterized by the fact that they are sent in context of an existing choreography instance. There are two subtypes of subsequent messages. One is sent from the web service back to the requester, e.g., response messages. The other subtype are subsequent messages from the requester to the web service to continue the message exchange with it.

ORCA is consequently always involved when messages are exchanged in order to ensure the correct execution of message exchanges. This means that ORCA is in the main execution path of the architecture. Every incoming as well as outgoing message will pass through ORCA to either cause the creation of a new choreography instance, the continuation of an existing choreography instance or in fact the termination of a choreograph instance in case the message was the last one that can happen.

Web services also have orchestrations. An orchestration defines how a web service uses other web service in order to fulfill its functionality. An orchestration can invoke several other web services in sequence, concurrently, conditionally, etc. Once a choreography is started, and the web service to be initiated is determined, it sends the first message as an invoker to the web service. In the general case a message exchange follows (see above).

In principle, the conceptual model of choreography and orchestration is the same with regard to the concepts required to formulate a process. From an architecture viewpoint ORCA can execute both, instances of choreographies as well as instances of orchestrations.

The architecture and its execution model have to assume that message exchanges are not always correct. For various reasons it might be that messages are lost, replayed, or superfluous messages are introduced. This means that either choreographies might be started without ever being terminated or choreographies will get “stuck” since no correct message is received any more. In addition, messages might be received that cannot be dealt with at all since they neither continue a choreography nor initiate a choreography. The latter case is an error situation that can be clearly detected. The former case is difficult to detect since it might always be that a correct message shows up after a long time.

4.14 Composition

Composition is the process of designing a workflow of web services based on their choreography specifications. Such a workflow is a DIP orchestration. The composition process can be external (be it manual or automatic), in which case the orchestration pre-exists in the SWS definition, hence is not under the responsibility of the orchestration component. In other cases, the composition must be dynamically computed.
As nothing in the WSMO/WSMX definitions precludes this possibility, the orchestration component can achieve such a composition, either as a preliminary activity (the orchestration is computed entirely, then executed), or interleaved with execution. To summarize, when not external, composition is a sub-activity of the orchestration component activity.

4.15 Discovery

Discovery in DIP is developed as part of WSMO: WSMO Web Service Discovery [3] defines the methodological framework and WSMO Discovery Engine [2] describes discovery in a more implementation-oriented view. As we are early in the project, not all parameters could be fixed already. Thus the following descriptions give an overview of discovery but are subject to change to adjust to the final version of discovery in DIP. Also we will include an approach for the discovery of adapters in the second version of the architecture document.

As described earlier in Section 2.2, the core discovery process in DIP is limited to finding abstract web service descriptions that can fulfill a service request. Determining a specific concrete service that implements the abstract service is most likely outside the scope of the discovery component. Discovery is carried out by matching WSMO goals to web service capabilities. Goals enable a service requester to describe formally what they want to achieve independently of any service, or composition of services, that may be able to satisfy the request. Goals have logical expressions describing post-conditions and effects. The post-conditions describe results expected by the service requester after the goal has been achieved. Effects are the real-world side effects expected if the goal is achieved.

Web service capabilities describe what the service offers and have preconditions, postconditions, assumptions, and effects. The semantics of the postconditions and effects of the capability are the same as those for the goal. The preconditions describe constraints on the input data required by the service and the assumptions describe the state of the world expected for the service to execute. The job of the discovery component is to match the logical descriptions of requester goals with the capabilities of web services whose descriptions are known to it. Matching logical descriptions in this way will most likely require the support of a WSML reasoner.

It is the responsibility of the discovery component which service registries it can consult to find semantic descriptions of web services to match requester goals. An implementation of discovery may consult with one or many local, remote, or distributed registries or another WSMX to carry out the discovery task. Additionally, the discovery component may require a mediation service in case different ontologies are used by service requesters providing the goal, and service providers providing the web service description.
4.16 Supporting components

Besides the major processing components described so far a number of supporting components are needed to be able to administer, configure, and manage DIP. Supporting components communicate with the architecture as given in Figure 4.1 in the same way as other nodes, i.e., via WSML messages. Thus we have a uniform interface and a flexible way to add new supporting components.

A preliminary list of supporting components include:

- WSMO/WSML editor
- Configuration manager (user preferences, general configuration, etc.)
- Security policy editor
- Trading partner management
- Semantic web services deployment
- Etc.
5 Sequence diagrams based on use cases

This section provides a closer look on key interactions in the DIP architecture by providing the related UML sequence diagrams.

5.1 Discovery

Service requesters are forced to discover web services before they can invoke them if a desired service is not known already. Figure 5.1 shows a user request to discover a service using a goal description.

![Interaction diagram of discovery](image)

The requester fills out a goal template first to specify the desired service she/he is looking for. The goal is then sent to the WSMX system, through an adapter if the requester needs to adapt its message to WSML, as a “discover” request. The WSML message includes the goal, the user’s ontology, and the input parameters given in the user’s ontology. The receiver hands over the “discover” request to an instance of the discovery service using an instance of the corresponding choreography. The discovery component checks if it can fulfill the request locally or not. If yes, it returns a list of matching services to the requester, otherwise it uses an external discovery component to find matching web services by invoking a service through the invoker. This external discovery components are independent of WSMX and can also be used without WSMX. Matching services are returned at the end to the requester for selection or further usage.

5.2 Mediation

As soon as a requester has discovered a desired service the service description including the service ontology is known to the requester. In case its ontology does not match
the user’s ontology, the requester asks the WSMX system to mediate between the two ontologies. Figure 5.2 shows this situation to mediate data adhering to two different ontologies.

![Interaction diagram of mediation](image)

The requester provides the data to mediate, the used ontology, and the service identifier in a WSML message which is generated by an adapter if necessary. The WSMX system checks if the internal mediator can fulfill the request otherwise an external mediator is invoked through the invoker. The mediated data is returned to the user and he/she can invoke the service.

### 5.3 Service Execution

Figure 5.3 shows the interaction between components during the execution of a simple scenario assuming that the requester has already discovered the desired web service and the data was mediated according to the service ontology. We assume that the service to be called is composed of one or more services but does not require any user interaction after invocation. Therefore the requester invokes the service at its local WSMX system by providing the service identifier and the input parameters in a WSML message. We assume that the requester can invoke services using WSML or it knows an adapter which translates requester requests into WSML invocations. WSMX only understands WSML but provides a framework for adapters and their definition.

The receiver of the local WSMX system transforms the request into an internal representation, before the choreography component is taking over. A new instance of a service choreography and implementation is created and initialized with the provided input parameters. The service implementation is either performing “real-world” actions, e.g., booking a ticket or printing an invoice, at the service provider, e.g., OEBB or VISA, or using the orchestration component to execute a sequence of other services. Choreographies are representations of service implementations for service invokers and
provide the communication API to interact with a service. On the other hand, service implementations extend a WSMX system at service providers to fulfill “real-world” actions. Composed services are also represented by implementations defining the order of executions, message inputs and outputs, etc. and are used by service requesters even if a composed service only consists of one service. The execution itself is handled by the orchestration component which executes the service (sequentially or concurrently, synchronously or asynchronously). The involved services can be local or remote. A choreography instance is created and initialized for each invoked service of a composition regardless if the service is local or remote. The choreography component uses the invoker to call the service using a WSML message. If a service provider does not support WSML an adapter is used to create the required format. Otherwise the message is tunneled by the adapter (the adapter is transparent).

The interactions between the WSMX components at the service provider side are symmetric to the service requester side. Therefore the WSMX components of the service provider side are omitted in Figure 5.3. The service results are returned to the service requester (through the adapter if necessary). After all services of a composed service have been executed or a service implementation of a local service was executed the results are returned to the requester.
To illustrate the interactions given in Figure 5.3 and simplify understanding, Figure 5.4 shows a use case example for the above described interactions between WSMX components and WSMX systems.

A user is booking a ticket for a train trip at OEBB, the Austrian railway company, using her/his VISA credit card and the ticket should be shipped to her/his address via mail. Post is the Austrian mail company. The user, the OEBB, and VISA are using the WSMX system to invoke and provide web services. The Post provides a shipping web service by running an application server, i.e., the web services are defined in WSDL.

First, the user enters her/his trip parameters (departure, arrival, VISA credit card information, user address, etc.) using a graphical user interface (GUI) representation of the web service (1). The GUI application knows an adapter to transform this input into a WSML message (2) which is sent to the local WSMX system of the user (3). The WSMX system creates and initializes a new choreography instance for this request and detects that an OEBB service has to be invoked. As only one remote service has to be invoked a simple orchestration is created containing only the OEBB booking service `book`. The orchestration component executes this composed service and creates a choreography instance for `OEBB.book` which will invoke and interact with the service provider. The choreography knows from the service description that OEBB is using WSMX and sends an WSML message via the invoker (4).

The WSMX system at OEBB creates a choreography instance for `OEBB.book` to handle the received request. To simplify understanding we use an informal representation. Still, the reader should have Figure 5.3 in mind while walking through this use case.

The service is composed of 3 services: `OEBB.reserve` to reserve a seat for a given train, `VISA.bill` to bill a user, and `Post.ship` to ship the ticket. First, the OEBB WSMX orchestration component processes the first service of the composed service to reserve the desired seat. Therefore it creates a new choreography instance of the `OEBB.reserve` service and invokes itself by sending an WSML message (5). The received message creates a second choreography instance of `OEBB.reserve` which will execute the service, i.e., adding the booked seat in a database, printing the ticket, calculating the price, etc., and returning a confirmation (6). Failure handling is omitted in this example.
for simplicity. The reservation confirmation is returned to the invoking choreography which will inform the orchestration component to continue with the next service of the composed service $OEBB.book$. The next services are executed in parallel (indicated by `<par>` in Figure 5.4).

A WSML message is sent to the VISA company (7) to bill the user. As VISA is also using the WSMX system and VISA’s billing service is not composed by other services, OEBB sends a WSML message not requiring an adapter and VISA executes the service implementation to withdraw the given amount of money from the users account (8).

The shipping service of Post is defined in WSDL and an adapter is required to transform the WSML message from the OEBB WSMX system (7) into an WSDL message for Post (8). This adapter is known to the OEBB system and used by the choreography instance at OEBB. The application server at the Post executes the request and the ticket will be shipped to the users address (9).

Figure 5.5 shows the interaction of an WSMX system with an application server providing services defined in WSDL. The WSMX components and interactions are equivalent to Figure 5.3. The difference in this example is that the adapter at the right-hand side adapts WSML messages to WSDL messages and vice versa to communicate with the application server.
WSDL

Figure 2.5: Simple example interaction diagram with an application server using...
5.4 Security

As security is a key concern in DIP, Figure 5.6 shows an example, where an incoming message is first decrypted, then authenticated and if these tests verify OK, it is forwarded for further processing. This is the basic processing for all requests that require enhanced security.

Figure 5.6: Security processing
6 SUMMARY AND CONCLUSIONS

In this document we have defined the first version of the DIP architecture. The contribution of this deliverable is to provide the basic description and the basic framework of the open web services architecture which is one of the three main goals of the DIP project. This is a key issue to enable interoperability. Thus this deliverable is relevant to all technical work packages in DIP (WP1–WP5). In conjunction with deliverable 6.3, this deliverable provides the interoperability guidelines for all components developed in DIP.

We have described the current standard “triangle” model of web services and how we evolve this architecture into the DIP architecture supporting semantic web services. The key extension is that the three party model is extended into a five party model by adding a discovery and a mediation party. Based on this model we have described DIP’s internal architecture for nodes (WSMX). WSMX is an event-based, dynamically configurable framework for supporting the definition and processing of semantic web services. We have described the individual components and provided detailed discussions of the processing of semantic web services in this architecture. The major interactions have been discussed and have been semi-formally defined in UML sequence diagrams which enable the reader to understand the architecture and its application in concrete scenarios. The scenarios we used in the sequence diagrams were derived from the DIP use cases and generalized.

Architectures defined in the course of projects as the one provided in this document are to be seen as “living” documents. The architecture defined in this document is not final yet as we have to envision possible changes and adjustments that will surface during the further stages of the project. These changes will be applied to the architecture and will be worked into the second version (deliverable 6.5) of the architecture.
REFERENCES


