Abstract

This deliverable comprises the specification of the testbed orchestration system and is part of the specification of Teagle. The orchestration system is used to allow users to express requests on the federated testing environment. These requests are mapped over existing services exposed by the testbed. In the general case, more than one testbed is used, which requires means for combining and synchronizing various unconnected components. The testbed orchestration system provides the facility for achieving such collaborative process, starting from the user request definition to the actual execution of an orchestration script.

The document firstly provides a list of requirements. The following section provides a state of the art on pre-existing languages and tools for the specification and the implementation of services orchestrations. Then, the document explains the essentials of the model driven engineering approach, as it is expected to play an important role. Finally, it describes the specification of the orchestration system in the form of a list of entities and reference points. The orchestration system roughly consists of a front end - with graphical tools and an intelligent request interpreter facility - and a back-end: the orchestration engine with code generation and execution facilities. It is connected to the rest of the Teagle architecture by repositories and interfaces.
Disclaimer

This document contains material, which is the copyright of certain PII consortium parties, and may not be reproduced or copied without permission.

All PII consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the PII consortium as a whole, nor a certain party of the PII consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Impressum

[Full project title] Pan European Laboratory Infrastructure Implementation
[Short project title] PII
[Number and title of work-package] WP3 - Service description, discovery and orchestration
[Document title] Testbed orchestration system specification
[Editor: Name, company] Mariano Belaunde, FT
[Work-package leader: Name, company] Sebastian Wahle, Fraunhofer
[Estimation of PM spent on the Deliverable] 20

Copyright notice

© 2009 Participants in project PII
Executive summary

This deliverable comprises the specification of the testbed orchestration system and is part of the specification of Teagle. The orchestration system is used to allow users to express requests on the federated testing environment. These requests are mapped over existing services exposed by the testbeds. In the general case, more than one testbed is used, which requires means for combining and synchronizing various unconnected components. The testbed orchestration system provides the facility for achieving such collaborative process, starting from the user request definition to the actual execution of an orchestration script.

The document firstly provides a list of requirements. The following section provides a state of the art on pre-existing languages and tools for the specification and the implementation of services orchestrations. Then, the document explains the essentials of the model driven engineering approach, as it is expected to play an important role. Finally, it describes the specification of the orchestration system in the form of a list of entities and reference points. The mapping between the requirements and the components identified in the architecture is provided. The orchestration system roughly consists of a front end - with graphical tools and an intelligent request processor facility - and a back-end: the orchestration engine with code generation and execution facilities. It is connected to the rest of the Teagle architecture by repositories and interfaces.
## List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>Mariano Belaunde</td>
</tr>
<tr>
<td>Fraunhofer FOKUS</td>
<td>Thomas Magedanz, Florian Schreiner, Sebastian Wahle</td>
</tr>
<tr>
<td>Univ. of Palermo</td>
<td>Luigi Alcuni, Silvana Greco Polito, Vincenzo Mancuso</td>
</tr>
<tr>
<td>Univ of Patras</td>
<td>Christos Tranoris</td>
</tr>
</tbody>
</table>
### Table of Contents

1. **Introduction** ................................................................................................................................. 12
   1.1 Objective of this Document........................................................................................................ 12
   1.2 Overview ......................................................................................................................................... 12
2. **Requirements** .................................................................................................................................... 13
3. **Alignment with the General Architecture** .................................................................................. 15
4. **State of the Art** .............................................................................................................................. 17
   4.1 Languages ......................................................................................................................................... 17
      4.1.1 BPEL ......................................................................................................................................... 17
         4.1.1.1 BPEL Business Processes ............................................................................................... 17
         4.1.1.2 BPEL and Java .................................................................................................................. 19
      4.1.2 UML Behaviour Paradigms ..................................................................................................... 20
      4.1.3 SPATEL ...................................................................................................................................... 20
      4.1.4 BPMN ........................................................................................................................................ 22
      4.1.5 SCXML ...................................................................................................................................... 23
   4.2 Automatic Composition Techniques .......................................................................................... 24
      4.2.1 Semantic Representation of Services ..................................................................................... 24
   4.3 Orchestration Tools ....................................................................................................................... 26
      4.3.1 BPEL Engines .......................................................................................................................... 26
         4.3.1.1 BPEL Engine: Orchestra v.4 ......................................................................................... 27
         4.3.1.2 BPEL Engine: ActiveBPEL ............................................................................................. 29
         4.3.1.2.1 ActiveBPEL for People .............................................................................................. 31
      4.3.2 SCXML Implementation on Apache ..................................................................................... 33
      4.3.3 SPATEL Engine (from IST SPICE) ....................................................................................... 33
      4.3.4 Semantic-based Frameworks for Service Composition ....................................................... 34
   4.4 Model Engineering ......................................................................................................................... 35
      4.4.1 Meta-modeling ......................................................................................................................... 36
      4.4.2 Operating with Models ........................................................................................................... 36
      4.4.3 Model Exchange ..................................................................................................................... 36
      4.4.4 The Power of Meta-modeling ............................................................................................... 36
      4.4.5 Model Transformations ......................................................................................................... 36
5. **Specification** .................................................................................................................................... 38
   5.1 Introduction ....................................................................................................................................... 38
   5.2 Entities and Reference Points ........................................................................................................ 39
      5.2.1 Orchestration Frontend ............................................................................................................ 39
         5.2.1.1 The Policy Evaluation Function ...................................................................................... 39
         5.2.1.2 The Creation Environment ............................................................................................... 40
         5.2.1.3 The Request Processor ..................................................................................................... 40
         5.2.1.4 The Reference Point REP .................................................................................................. 40
         5.2.1.5 The Reference Point POL ................................................................................................. 40
         5.2.1.6 The Reference Point SPEC .............................................................................................. 41
      5.2.2 The Orchestration Engine (Orchestration Backend) ............................................................ 41
         5.2.2.1 Rationale of Orchestration Engine architecture ................................................................ 42
List of figures

Figure 1 - Overall PII Architecture ................................................................. 15
Figure 2 - Example of BPEL Process.............................................................. 18
Figure 3 - BPMN Process Sample................................................................. 23
Figure 4 - States Machines in SCXML .......................................................... 24
Figure 5 - OWL-S Top Level Service Ontology ........................................... 25
Figure 6 - Architecture of Orchestra v.4 .................................................... 28
Figure 7 - Process Virtual Machine (PVM) .................................................. 28
Figure 8 - PVM Interfaces for Multi-language and Multi-service Support .......... 28
Figure 9 - Example of ActiveBPEL BPEL Process Created with the ActiveBPEL Designer .... 31
Figure 10 - BPEL Code Generated by the ActiveBPEL Designer in XML format .... 31
Figure 11 – Description of a People Activity in a BPEL Process ......................... 32
Figure 12 - Development Process with Apache SCXML .................................. 33
Figure 13 - WSMX Architecture ................................................................. 35
Figure 14 - Teagle Overall Architecture ...................................................... 38
Figure 15: The Teagle Request Interpreter .................................................... 39
Figure 16 - The Orchestration Engine Architecture ....................................... 42
Figure 17 - IMS Overall Components ......................................................... 48
Figure 18 - HSS Configuration Entities ....................................................... 50
Figure 19 - Configuration Data for HSS ....................................................... 51
Figure 20 - The Generated HSS Orchestration Interface ............................... 53
Figure 21 - The HSS Configuration Service ................................................. 53
Figure 22 - Orchestration Script logic ......................................................... 54
<table>
<thead>
<tr>
<th>Abbreviations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABE</td>
<td>Aggregate Business Entity</td>
</tr>
<tr>
<td>CCM</td>
<td>CORBA Component Model</td>
</tr>
<tr>
<td>CIM</td>
<td>Common Information Model</td>
</tr>
<tr>
<td>DEN-ng</td>
<td>Directory Enabled Networks-new generation</td>
</tr>
<tr>
<td>EAST-ADL</td>
<td>EAST Architecture Description Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Foundation</td>
</tr>
<tr>
<td>eTOM</td>
<td>enhanced Telecom Operations Map</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>IGW</td>
<td>Interconnection Gateway</td>
</tr>
<tr>
<td>JET</td>
<td>Java Emitter Templates</td>
</tr>
<tr>
<td>JUL</td>
<td>Java Sun’s java.util.logging package</td>
</tr>
<tr>
<td>LwCCM</td>
<td>Lightweight CCM</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modeling and Analysis of Real Time and Embedded systems</td>
</tr>
<tr>
<td>NGOSS</td>
<td>New Generation Operations Software and Systems</td>
</tr>
<tr>
<td>NXD</td>
<td>Native XML Databases</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PTM</td>
<td>Panlab Testbed Manager</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RDFS</td>
<td>Resource Description Framework Schema</td>
</tr>
<tr>
<td>SAWSDL</td>
<td>Semantic Annotations for WSDL and XML Schema</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SoaML</td>
<td>Service oriented architecture Modelling Language</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modeling Language</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TMForum</td>
<td>TeleManagement Forum</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UML-DI</td>
<td>UML Diagram Interchange</td>
</tr>
<tr>
<td>VCT</td>
<td>Virtual Customer Testbed</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Meta Interchange</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language for Transforms</td>
</tr>
</tbody>
</table>
# Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>The term user is referring to a person that is actively using or testing services provided by a customer’s resource inside the testbed federation. Differentiation between End-User and Test-User may only be useful when directly referring to the special task of active service testing or the usual task of common service usage. End-User is also a person acting as a part of User Driven Innovation (UDI) process, thus collecting information and giving feedback to Customers for instance in Living Labs environments’ field testing projects.</td>
</tr>
<tr>
<td>User Driven Innovation</td>
<td>Inclusion of the user as part of the testing process in order to take into account early feedback. In this project two levels of User Driven Innovation (UDI) will be applied.</td>
</tr>
<tr>
<td>-</td>
<td>The first level refers to the customer (the organisations and companies that use the federation to test their products and services).</td>
</tr>
<tr>
<td>-</td>
<td>The second level refers to the potential end-users of the services.</td>
</tr>
<tr>
<td>To generally differentiate between these two groups in this document, we will refer to the first as federation-customers, service-testers or service-providers and the second we will refer to as end-user.</td>
<td></td>
</tr>
<tr>
<td>Provider</td>
<td>This term refers to a party owning testing infrastructure and other testing resources and has entered an agreement allowing a customer to use its testing infrastructure and resources to develop and test services.</td>
</tr>
<tr>
<td>Testbed</td>
<td>A testbed is an environment which allows experimentation and verification for research and development products. A testbed provides rigorous, transparent and replicable testing and herein it is always used in the context of new information and (tele-)communications technologies for networks and services.</td>
</tr>
<tr>
<td>Testbed federation</td>
<td>A testbed federation or federated testbeds is the interconnection of two or more independent testbeds for the creation of a richer environment for testing and experimentation, and for the increased multilateral benefit of the users of the individual independent testbeds.</td>
</tr>
<tr>
<td>Customer</td>
<td>A customer of Panlab is someone that used TEAGLE to set up testbed resources for the purpose of testing or developing services. The customer is able to directly connect to the rented resources (his VCT) using a VPN client (U3 interface).</td>
</tr>
<tr>
<td></td>
<td>The customer can offer his services under the terms of Panlab user domain access (U2 interface) to external Test-/End-Users.</td>
</tr>
<tr>
<td><strong>Testing session</strong></td>
<td>Herein the term testing session refers to a well-defined temporal and spatial relation of testing infrastructures and testing resources by the customer or user(s).</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>VCT</strong></td>
<td>A Virtual customer Testbed is the sum of all resources, including interconnections, rented by the customer. It basically is an isolated network in that the customer is able to “dial in” and directly access the items assembled in TEAGLE. Each customer operates inside its own VCT and has no access to other VCTs. The purpose of direct VCT access is to configure and develop but not to widely test new services; such testing is intended to be done from outside using external Test- and/or End-Users.</td>
</tr>
<tr>
<td><strong>Information Model</strong></td>
<td>An information model is a platform independent model of entities and high-level relationships linking them.</td>
</tr>
<tr>
<td><strong>Data Model</strong></td>
<td>A data model is a platform specific model of entities and lower level relationships linking them. From RFC 3444: “They are intended for implementers and include protocol-specific constructs”.</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Objective of this Document

The objective of this document is to provide a specification of the testbed orchestration system that will guide the implementers of the software components that will provide the orchestration functionality. At this stage of the project the specification consist mainly in an architectural view providing the list of major sub-components (entities) and the list of reference interface points between the sub-components or between other components of the Teagle system.

1.2 Overview

The orchestration system is used to allow users to express requests on the federated testing environment. These requests are mapped over existing services exposed by the testbed. In the general case, more than one testbed is used, which requires means for combining and synchronizing various unconnected components. The testbed orchestration system provides the facility for achieving such collaborative process, starting from the user request definition to the actual execution of an orchestration script.

The document firstly provides a list of requirements. The following section provides a state of the art on pre-existing languages and tools for the specification and the implementation of services orchestrations. Then the document explains the essentials of the model driven engineering approach, as it is expected to play an important role. Finally it describes the specification of the orchestration system in the form of a list of entities and reference points. The orchestration system roughly consists of a front end - with graphical tools and an intelligent request interpreter facility - and a back-end: the orchestration engine with code generation and execution facilities. It is connected to the rest of the Teagle architecture by repositories and interfaces.
2 Requirements

This section gathers all requirements relevant for the Testbed Orchestration System specification. Different requirements categories have been categorized as either Basic or Advanced depending on the perceived complexity.

The requirements gathered are labelled in accordance with the following naming scheme:

##D3.3- Category##M. ReqNumber, where

- **Category**: name of the Category used to classify the requirement.
- **ReqNumber**: A number identifying the requirement within each category.

**NOTE**: Two consecutive numbers differ in 50. This allows including new requirements between two already existing ones without changing the numbering.

**Requirement ##D3.3-Basic##M.100**

**Statement**: The orchestration system should be able to orchestrate resources and configuration components distributed in various testbeds that are part of the federation.

**Justification**: A major motivation for the orchestration system is to simplify the task of aggregating resources from various complementary testbeds. A request may include various resources belonging to different Panlab Partners of the federation.

**Requirement ##D3.3-Basic##M.150**

**Statement**: The orchestration system should provide facilities to help users (partners and customers) to formulate testbed requests that are complete - that is to say, containing sufficient information for automated execution.

**Justification**: A testbed request may potentially involve a large amount of configuration data to be assigned. Without proper assistance this task will be too difficult to handle for non specialists.

**Requirement ##D3.3-Advanced##M.200**

**Statement**: The orchestration system should provide facilities to help users (partners and consumers) to formulate testbed requests that are valid from the point of view of policy rules (like authorization to access some restricted components).

**Justification**: It would not be acceptable to users of Teagle to receive a great number of notification errors that arrive at execution time and that could, in reality, be checked at request design time.

**Requirement ##D3.3-Advanced ##M.250**

**Statement**: The orchestration system should provide facilities to easily simulate the behaviour of a specific invoked component that would not be available (at least temporarily) during the testbed request design.

**Justification**: It is a common situation that some components become temporarily unavailable. In order to test anyway the overall orchestration script, it is necessary to replace the invocation to the real component to another "fake" one.

**Requirement ##D3.3-Advanced ##M.300**
Statement: The orchestration system should provide adaptation facilities to allow replacing a call to a service component to a call to another component providing similar functionality, even when the exposed interfaces of both components differ.
Justification: It is important to support situations like version upgrade or change of testbed provider. The "real" interface may be different, but an adaptation step can suffice to allow reusing the same orchestration script despite the change of the invoked component.

**Requirement ##D3.2-Basic##M.350**
Statement: The orchestration system may offer the possibility to orchestrate freely available service components exposed as web services coming from the internet.
Justification: Some utilities could not be present in the registered testbeds but instead exist and be freely available in the internet.

**Requirement ##D3.3-Basic##M.400**
Statement: the Testbed Orchestration System should allow PII Partners to create testbed requests.
Justification: PII Partners will use the orchestration system to check if their testbed is well integrated.

**Requirement ##D3.3-Basic##M.450**
Statement: the Testbed Orchestration System should allow PII Customers to create testbed requests.
Justification: PII Customers will use the orchestration system to achieve their tests.

**Requirement ##D3.3-Advanced##M.500**
Statement: The orchestration system should support the checking of policy conformance when a request is created.
Justification: Users should be able to test requests at design time (opposed to execution time).
PII Partners and PII Customers can use the PII testbed resources according to personalized profiles.

**Requirement ##D3.3-Advanced##M.550**
Statement: The orchestration system should support delayed execution of a testbed request - implying resource reservation - in addition of supporting immediate execution.
Justification: Reservation is a major concept when dealing with testbed usage.

**Requirement ##D3.3-Basic##M.600**
Statement: testbed requests and orchestration specifications should be stored for future testing environment reproduction.
Justification: after a testbed has been used for experiment, additional experiments could be required by the same customer in order to check additional aspects of the research or to verify the previously obtained results.
3 Alignment with the General Architecture

This section presents a summary of the underlying architecture as described in D2.1 [52]. The aim is to provide the reader with an understanding of the work this deliverable is building upon and how this work will be further advanced.

The architecture detailed in D2.1 (Figure 1) defines four main conceptual components:

- TEAGLE and support functions
- The Panlab Testbed Manager (PTM)
- The Interconnection Gateway (IGW)
- The Testbed Components

The first component listed, TEAGLE and support functions is not described in detail in D2.1 as this work has been allocated to its own work package (WP3), which this document is a part of. In D2.1 the TEAGLE was described for completeness of the infrastructure specification and also to outline the interfaces between TEAGLE and the other architectural components (primarily the PTM). In short, TEAGLE provides a user interface for the PII testbed federation to the customer. It allows the customer to view all available testbed resources and construct a Virtual Customer Testbed (VCT) request that satisfies their unique testing requirements.

The Panlab Testbed Manager (PTM) is a major component of each PII testbed. It essentially provides the link between the customer’s request coming from TEAGLE and the individual testbed components that realize this request. The PTM exposes a Web Service interface towards TEAGLE. This interface allows TEAGLE to issue a provision or configuration request for the customer after interpreting their initial request based on resources available, SLA constraints, etc. The PTM provides the backend support to implement this request. It essentially translates the provision/configuration messages sent from TEAGLE into resource specific commands that the individual testbed components can understand and execute.
The PTM also provides other functionality besides its primary message translation role. This functionality relates to resource management. It can implement a resource discovery service that detects changes that occur to the testbed resources. For example when a new testbed resource is added to a testbed or a resource becomes unavailable for any reason (i.e. maintenance upgrade, resource goes offline due to a local network failure, etc.) the PTM communicates these updates to TEAGLE, which in turn reflects the change towards the customer. The PTM provides a monitoring service for all testbed resources connected to it. This facilitates the customer’s need for validation by ensuring the testbed behaves correctly and that it has been setup properly. Finally, the PTM facilitates the configuration of a testbed federation involving multiple testbeds. It achieves this by instructing the Interconnect Gateway (IGW) to communicate provision and configuration commands sent from TEAGLE to the other involved testbeds through their own IGWs.

The IGW is another mandatory component that all testbeds must contain one or more IGWs. Its purpose is to connect testbeds and components inside these testbeds to establish the required testbed federation setup requested by the customer. The IGWs will be instructed by the PTMs through specific protocols to create or delete connections between the involved testbeds and their components.

VCTs will be used in PII to provide the support for fulfilling the customer’s requested testbed federation setup. The VCT solution will be delivered by establishing one VPN per customer to connect the required resources from the various testbeds. By using independent VPNs for each VCT request, security will be inherently enforced as the VPNs are completely independent from each other.

The final conceptual component described in the overall PII architecture is the testbed components that form the Resource Plane of each testbed. A testbed component is a single logical entity within the infrastructure offered by a testbed for the use of PII customers. No specific technology limits are placed on these components by PII; however each testbed component must be individually controllable by the PTM. In addition to the primary functionality the testbed component provides in the Resource Plane, all testbed components must provide:

- Individual identification
- Addressability and be accessible from the PTM
- Controllability by accepting provisioning and configuration commands
- Optional monitoring capabilities to inform the PTM about its current state and convey test results back to the PTM.

For more information and an in-depth discussion of the general architecture entities as well as the associated interfaces please refer to D2.1.
4 State of the Art

4.1 Languages

4.1.1 BPEL

The Business Process Execution Language (BPEL) [3], also known as WS-BPEL and BPEL4WS, is a standard issued by the Organization for the Advancement of Structured Information Standards (OASIS) [9]. The latest release of the standard is known as WS-BPEL 2.0. Note that BPEL specifications were firstly released by IBM in a public draft release, which is known as BPEL 1.1 [10].

BPEL standard defines a language for specifying business process behaviour based on Web Services. It extends the Web Services interaction model and enables it to support business transactions. BPEL allows composition of web services to build and execute business processes. Composition is made following the orchestration paradigm in which a central process takes the role of coordinator and controls the execution of the web operations of the different web services involved in the composition process.

BPEL is an XML programming language, and it has three basic components: Programming logic, Data types, and Input/Output. These three components are split up in the following way:

- Programming logic - BPEL
- Data types - XSD (XML Schema Definition)
- Input/Output (I/O) - WSDL (Web Services Description Language)

XSD is used to define the types to be used in a program. WSDL is used to define the web service that will actually execute a service. Finally, BPEL will put all these things together.

More in general, BPEL can be used to support integration of applications inside an enterprise and to support cross enterprise business interactions in which the business processes of each enterprise interact through Web Service interfaces [11].

BPEL allows to describe both abstract business processes and executable ones. Abstract business processes are partially specified and are not intended to be executed. They serve a descriptive role and may hide concrete operational details. Executable processes are fully specified and are intended to be executed. In BPEL, all the constructs of the executable processes are available to the abstract ones. In addition, abstract processes are provided with mechanisms to hide operational details. Abstract processes have more than one use case. Examples of use cases of an abstract process are describing the observable behaviour of services offered by an executable process and definition of a process template that embodies domain-specific best practices. In this case the process template should describe the process logic excluding execution details which are provided by an executable process.

BPEL is an XML-based language and supports the protocols of the web services technology stack such as SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), UDDI (Universal Description Discovery and Integration), and other protocol for the handling of Web Services (WSs), such as WS-Reliable Messaging, WS-Addressing, WS-Coordination and WS-Transaction.

4.1.1.1 BPEL Business Processes

A BPEL business process is activated after receiving a request from a caller. To satisfy such a request, the BPEL process invokes the involved web services and finally provides the caller with a response. BPEL processes can be synchronous or asynchronous. Synchronous processes block the entity that...
using the process until the process finishes and returns a result to it. Asynchronous processes use a callback to return the result to the client and do not block it.

In BPEL the process logic is described through a set of activities. Two kinds of activities are defined: basic activities and structured activities. Basic activities describe elemental steps of the process behaviour. Structured activities describe the control-flow logic and can contain other basic and structures activities recursively. Examples of BPEL basic activities are:

- `<invoke>`, it is used to invoke other web services;
- `<receive>`, it is used while waiting for a message to invoke the process;
- `<reply>`, it is used to generate a response;
- `<wait>`, it is used to introduce a delay;
- `<assign>`, it is used to manipulate data variables;
- `<terminate>`, it is used to terminate the process.

Structured primitives define the order in which a collection of activities is executed. They describe how a business process is created. The composition of basic primitives comes to structures that express the control patterns, handling of faults and eternal events, coordination of message exchanges between the different process instances involved in a business protocol.

Examples of BPEL structured activities are:

- `<sequence>`, `<if>`, `<while>`, `<repeatUntil>`; they allows sequential control between activities and loops;
- `<flow>`, it defines a set of activities that will be invoked in parallel and is used to guarantee concurrency and synchronization between activities;
- `<switch>`, it is a case-switch construct to implement branches;
- `<pick>`, it allows deferred choice.

When we define a BPEL process, we actually define a new web service that is a composition of existing services. The interface of the new BPEL composite web service uses a set of port types, through which it provides operations like any other web service.

Figure 2 shows a schematic view of a BPEL process, in which the interaction between BPEL processes and web services is also enlightened.

![Figure 2 - Example of BPEL Process](image)

Note that BPEL processes can interact with external web services in two ways:
- the BPEL process receives invocations from clients. A client can be the user of the BPEL process that makes the initial invocation, or another web service for example one that has been invoked by the BPEL process and has to make a callback to return a result;
- the BPEL process itself invokes operations on other web services.

Also from Figure 2, it is possible to see that BPEL uses the notion of Partner Links to model partner relationships and interactions with other processes. The relationship of a business process to a partner is typically peer-to-peer and requires a two-way dependency at the service level. Typically a partner represents both a consumer of a service provided by a business process and a provider of a service to the business process. BPEL uses port types to shape the relationship with the partner in the two directions. In BPEL, each process has at least one partner link, i.e., the client that has invoked the process, and can have one or more invoked partner links related to the invoked web services.

The reason why BPEL treats clients as partner links is twofold: i) to support asynchronous interaction of processes, and ii) because BPEL process can offer services through port types, and that can be used by more than one client. Note that the process may wish to distinguish between different clients and offer them only the functionality they are authorized to use.

To summarize, the partner links describe links to partners, where partners might be:
- services invoked by the process;
- services that invoke the process;
- services that are invoked by the process and that can also invoke the process.

### 4.1.1.2 BPEL and Java

For simple processes, the composition of web services could be operated by means of Java. However, for more complex processes BPEL provides at least two important advantages over Java.

First, BPEL processes are portable even outside the Java platform. In fact, BPEL processes can be executed on orchestration servers based on Java platform or on any other software platform such as Microsoft .NET. This is particularly important in business-to-business interactions where different partners use different platforms.

Second, BPEL support specifically business processes, which are usually long-running, particularly if they involve interactions with partners over Internet. Business processes might invoke a web service and need to wait for the callback a relatively long time. Java does not offer support for such a kind of interaction between processes, e.g., Java does not natively support the tracking of Java applications/processes that can be closed or kept alive in order to receive callbacks.

Additionally, BPEL supports compensation in a relatively easy way. Compensation, or undoing steps in the business process that have already completed successfully, is one of the most important concepts in business processes. The goal of compensation is to reverse the effects of previous activities that have been carried out as part of a business process that is being abandoned.

Compensation is related to the nature of most business processes, which are long running and use asynchronous communication with loosely coupled partner web services. Business processes are often sensitive in terms of successful completion because the data they manipulate is sensitive. Because they usually span multiple partners (often multiple enterprises) special care has to be taken that business processes either fully complete their work or that the partial (not fully completed) results are undone – compensated.

BPEL also allows for concurrent activities, which are modeled using the `<flow>` activity. Gathering nested activities within `<flow>` is straightforward and very useful for expressing concurrency scenarios that are not too complicated. To express more complex concurrency scenarios, `<flow>` provides the ability to express synchronization dependencies between activities. In other words, it is possible to specify which activities can start and when (depending on other activities) and define complex dependencies.
Finally, note that legacy web services are stateless, while model business processes require a stateful model. New instances of a business process are created and linked to other business processes for the duration of the business process. BPEL provides a mechanism to use specific business data to maintain references to specific business process instances and calls this feature correlation.

It is clear that Java cannot offer enough support for business processes with that kind of requirements, therefore BPEL cannot be replaced by Java. Of course, it is also obvious that BPEL cannot replace Java neither as a programming language nor as a platform. Actually Java platform is very suitable for running BPEL processes. Therefore BPEL and Java fit together, where Java takes the role of the programming language for web services and the platform on which web services and BPEL processes are executed.

### 4.1.2 UML Behaviour Paradigms

UML (Unified Modelling Language) is a general-purpose graphical notation that is widely used for software modelling, typically at design time and as a mean to generate code automatically. There are various popular usages - like its capacity for defining information models - see Deliverable D3.2. In respect to the orchestration topic, we are interested here in the paradigms that UML offer for describing behaviour.

Three main paradigms can be mentioned here:

- **Activity Diagrams**: Activity Diagrams consists of a graph of nodes and edges used to formalize a sequence of actions. A node represents typically an activity and an edge the move from an activity from another. Specific nodes are used for represent specific construct like conditions, split and synchronization of the flow control. Activity diagrams are appropriate for represent processes, that’s why they can be used to represent business process or software engineering processes. The execution semantics behind UML activity diagrams is based on Petri nets theory [48].

- **State Machines**: State Machines Diagrams represent finite state machines composed of a list of states, transitions between these actions, possibly triggered by events, and actions, executed internally within a state or when firing a transition. State Machines are well appropriate to describe relatively complex behaviour, such as to represent the interaction between a human and a machine.

- **Sequence Diagrams**: Sequence diagrams are mainly used to represent the sequence of messages that are interchanged between one or more parties when collaborating. It is particularly useful to define nominal scenarios (the flow of messages when everything goes well) of services. Different kind of communication can be represented like asynchronous and synchronous messages, procedure calls. The time progression is represent by a vertical line for each of the involved parties.

### 4.1.3 SPATEL

SPATEL [49] is a high-level and executable language for describing composite telecommunication services. SPATEL stands for Spice Advanced language for Telecommunication services, where SPICE is the name of the IST collaborative project that created. SPATEL can essentially be seen as a customization of the UML language for expressing the definition of service interfaces and service composition logic that is well-suited to the telecom domain.
In contrast with most IT web services, telecom services are generally transactional, asynchronous, state-full and sometimes long-running processes. In addition a telecom service can be designed to be multi-modal – the ability to achieve a conversation using parallel interaction means like voice, text and image. Also the behavior of a telecom service needs to be often split in two parts: one running in the mobile terminal of the user – dealing with GUI aspects and local activation of telecom resources (like SMS sending) - and the other part running at server side, usually hosted by a telecom operator.

In the service developer formalism, a service is primarily described through an external view which provides information that is useful for service clients. The external view is basically an interface declaring a list of operations, input and output events, multimedia streams and relevant side-effects. The constraints on the service interface such as the ordering of operation invocations can be precisely defined through a contract. An important feature of SPATEL is the ability to annotate the elements of the interface (like the operations and the parameters) with semantics tags and non functional features to enable rich scenarios for service discovery and dynamic composition. Non-functional features are partitioned on the basis of categories like quality of service (QoS), charging, internationalization or resource usage. The annotation mechanism, which is similar to the approach taken in SA-WSDL approach – as it relies on pointers to pre-existing ontologies – is not detailed in this paper – which is focused on static composition.

The service developer formalism also allows representing the internal view of a service (white box representation) by means of a set of inter-connected service components. Two distinct views are available: an architectural view showing the list of involved components and their connections and a behavioral view consisting of state machines that define precisely the logic of an operation – an orchestration of components being a particular case. We will see some examples of the usage of this formalism in Section 4.2. The choice of state machines – rather than activity diagrams – is motivated by the idea of integrating "voice-based" dialogs in a service specification, since state machines are the most used paradigm for expressing the complexity that can be found in human-machine voice conversations. We should note however that the scope of SPATEL is much broader than the scope of traditional voice services since we have to deal with remote synchronous and asynchronous invocations, parallel threads of execution and dedicated GUIs definition at terminal level.

Concerning the definition of user interaction at client side, SPATEL provides the ability to represent potentially the usage of different GUI frameworks found in mobile world like, the very constrained J2ME [7] GUI environment or the richer GUI framework available in S60 Nokia [8] smartphones. This heterogeneity is enabled in the SPATEL meta-model by the fact that the coding of GUIs elements is generic: a Container contains recursively GUI Elements which in turns define GUI properties – which are name/value pairs. In addition to that GUI events can be connected to service events used within the logic of the service. Hence, thanks to this very pragmatic approach – not trying to model the whole IT world! – in our context, supporting a GUI framework means having the corresponding library of the GUI widget model components instantiated in the SPATEL design tool and having the corresponding code generator targeting the specific GUI framework.

In addition to this GUI aspect, SPATEL provides means to represent typical voice-based interactions: recognition of voice as utterance events, buffered construction of voice messages – which are delivered when reaching a stable state and support of specific events – like inactivity or failure recognition. The SPATEL language and execution environment inherits from previous research work done in the field of voice service modeling [9]. Hence, this aspect will not be detailed here. However it is important to point out that the combination of voice interaction modeling with GUI modeling brings the ability to model multi-modal services.
As we can notice, despite the specificity in telecom services, from a design point of view, the concepts needed by the service description language are not significantly different from those exposed by the well-know SOA standards like WSDL [10] and BPEL [11] and formalized in a more abstract way by UML [4]. The SPATEL formalism aggregates in fact well-know constructs coming from different sources (ITU-SDL [12], SA-WSDL [13], VoiceXML[14]) in order to provide the needed subset – not less, not more – that is needed for a high-level and executable formalism usable in telecom context. Among the potentially infinite design choices that UML can offer, SPATEL makes a very precise and exclusive selection like: using simple UML 1.4 state-machines instead of the full UML2 capability, not using collaborations, representing an orchestration as the behavior specification of an operation. Selectivity in the usage of the constructs offered by UML for behavior definition is necessary to have at the end a formalism that is effectively implantable and executable at reasonable cost.

Technically speaking the SPATEL formalism has been defined by an EMOF meta-model and is accompanied with a UML2 profile defining the conventions for using the UML graphical notation – like adding an specific icon to represent the invocation of remote service operations. This approach, which makes the distinction between abstract syntax and concrete notation, allows using a rather compact and understandable XMI serialization format as exchange and pivot format, significantly less complex than the one associated with the complete UML2 metamodel. Also it allows attaching alternative notations to the SPATEL language, such as a dedicated textual syntax, and still relying on a common abstract representation in memory and in persistent storage.

4.1.4 BPMN

The Business Process Management Initiative (BPMI) has developed a standard Business Process Modeling Notation (BPMN). The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. BPMN will also be supported with an internal model that will enable the generation of executable BPEL4WS. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A Business Process Model, then, is a network of graphical objects, which are activities (i.e., work) and the flow controls that define their order of performance.

A BPD is made up of a set of graphical elements. These elements enable the easy development of simple diagrams that will look familiar to most business analysts (e.g., a flowchart diagram). Within the basic categories of elements, additional variation and information can be added to support the requirements for complexity without dramatically changing the basic look-and-feel of the diagram. The four basic categories of elements are:

- Flow Objects
- Connecting Objects
- Swimlanes
- Artifacts

For modelers who require or desire a low level of precision to create process models for documentation and communication purposes, the core elements plus the connectors will provide the ability to easily create understandable diagrams like the following example diagram.
BPMN was designed to allow modelers and modeling tools some flexibility in extending the basic notation and in providing the ability to additional context appropriate to a specific modeling situation, such as for a vertical market (e.g., insurance or banking). Any number of Artifacts can be added to a diagram as appropriate for the context of the business processes being modeled. The current version of the BPMN specification pre-defines only three types of BPD Artifacts, which are: Data Object, Group and Annotation.

Business process modeling is used to communicate a wide variety of information to different audiences. BPMN is designed to cover many types of modeling and allows the creation of process segments as well as end-to-end business processes, at different levels of fidelity. Within the variety of process modeling objectives, there are two basic types of models that can be created with a BPD:

- Collaborative (Public) B2B Processes
- Internal (Private) Business Processes

4.1.5 SCXML

State Chart XML (SCXML) is a specification of the World Wide Web Consortium (W3C). It provides a generic state-machine based execution environment based on Harel State Tables. Here's an example - taken from the SCXML W3C specification [50] - which shows how a simple state machine - depicted in UML - is encoded in SCXML.
SCXML can represent arbitrary complex state machines, with sub-states and parallelism. It can be seen as a simplified XML representation of UML State Machines (version 1.5, not UML 2 which has introduced some more concepts).

An open source implementation is provided by Apache (Commons SCXML, see Section 4.3.2).

### 4.2 Automatic Composition Techniques

#### 4.2.1 Semantic Representation of Services

Semantic service descriptions are not only required for an automation of service discovery, orchestration and composition mechanisms, but also for users to be able to judge whether a given service’s functional as well as non-functional aspects serve a user’s needs. The requirements for the capabilities of the service discovery logic can vary. If the main purpose of the discovery process is to find single applicable services, focusing on functionality only, limited service description mechanisms might be sufficient. If the discovery mechanism should also be able to distinguish between non-functional aspects of a single service, service description mechanisms get more and more sophisticated. If service discovery systems should provide information about several services, sufficient for automating the service composition, the task of describing and discovering services gets complex.
Generally speaking, there are several processes that have to be conducted in order to allow for the delivery of semantically enabled services compositions. After a given service has been designed, implemented and deployed, service descriptions have to be either annotated or created with sufficient information about functional (input, output), non-functional (performance, reliability, security, price, etc.) aspects of a service as well as about information for the mediation (for conducting the composition) between service components. These service descriptions have to be published at a service registry for subsequent discovery. After service descriptions have been discovered, service components can be invoked ready for generating service compositions that finally can be executed.

In order to reduce efforts for the annotation process and in order to provide automation for the generation of service descriptions, the relevant information should already be integrated at service creation/implementation time. The “Meta-Tag” feature of Java ([13] and [14]), as an example, allows for source-code integration of service descriptions, which later on can automatically be published at service deployment time.

Technically speaking, in order to describe a service in a commonly understandable manner, a profound, extensive and extensible ontology that describes e.g. a Telco / NGN service (like [12]) is required. Furthermore, a semantically enriched service registry (such as Universal Description, Discovery and Integration (UDDI) or Electronic Business Extensible Markup Language (ebXML)) for storing and retrieving service information is needed. Finally, the description of each service has to be dynamically fed into and discovered from the service registry. Currently there exist several concurrent approaches for Semantic Web Services (SWS). Rather light weighted approaches are the Web Services Semantics (WSDL-S) [15] approach and its successor, the Semantic Annotations for WSDL and XML Schema (SAWSDL) [16]. Here only a limited amount of semantic information can directly be annotated as part of the WSDL files, so that semantically enriched UDDIs are able to enhance the service discovery process with this additional information. With WSDL-S only a limited set of tags, such as service concept, location and domain, service pre-/post-conditions for annotating WSDLs can be specified.

More sophisticated architectures like the Web Service Modeling Ontology (WSMO) [18] and, based on the Web Ontology Language (OWL), the Semantic Markup for Web Services (OWL-S) [17] make use of additional systems that enhance the service description and discovery process. Whereas WSDL-S and SAWSDL exclusively use the semantic information stored in the WSDL for describing Web Services, OWL-S defines a “Service Profile”, “Service Model” and a “Service Grounding” with which additional aspects about a given service can be described.

Figure 5 - OWL-S Top Level Service Ontology [18]

Furthermore, WSDL-S and SAWSL primarily can be used for BPEL-based interactions between services; OWL-S defines its own “Process Model” which can be used for different service interaction
mechanisms. WSMO’s architecture consists of the Semantic Web Service Language (WSML) based on F-logic and the Semantic Web Service Architecture (WSMX). WSMO supports functional aspects, non functional aspects, and mediators for Web Services and petri nets for execution semantics. By defining its own ontology WSMO enables discovery of Web Services based on goals and service capabilities. Supporting and utilizing a broad range of description logic reasoners such as Pellet [19], WSMX allows for enhanced matchmaking, even between different ontologies.

Summing it up, the selection of an appropriate approach heavily depends on the requirements for the service discovery and service composition mechanism. The more versatile, flexible and dynamic the requirements get, the more complex are the applicable service description, service discovery and service composition architectures. For service environments with rather static numbers of services, with limited requirements for agile and flexible service composition, light weighted approaches like WSDL-S and SAWSDL might be sufficient. In order to support rich sets of services, with dynamic service lifecycles and flexible service composition mechanisms approaches like OWL-S and WSMO are required.

4.3 Orchestration Tools

4.3.1 BPEL Engines

Several BPEL engines have been developed in the last few years by many companies, using proprietary and open source solutions. Here we firstly give a few examples of BPEL engines:

- BPEL4J was proposed in the early 2000’s by IBM as a platform for creating and executing BPEL4WS processes;
- WebSphere Business Integration Server Foundation (WBISF) has been introduced by IBM as an integration platform for building and deploying composite applications; it runs on top of the WebSphere Application Server platform, and it is compatible with Windows and Linux operating Systems;
- BPEL Process Manager, by Oracle, is meant to facilitate the development of SOA applications through composing synchronous and asynchronous services into end-to-end, standard BPEL process flows; it runs on top of the Oracle Application Server;
- BPEL Orchestration Server, running on top of a J2EE application server, was introduced by Collaxa to integrate BPEL4WS and WS-Transaction standards; since Collaxa has been acquired by Oracle, the BPEL Orchestration Server has been integrated in the Oracle Application Server;
- OpenStorm ChoreoServer was proposed for the Microsoft .NET platform;
- Fivesight PXE is a modular BPEL engine with a focus on reliability and flexibility;
- Active Endpoints ActiveWebflow Server is a complete BPEL engine running on top of a J2EE application server or standalone with a Web Servlet container, e.g. Tomcat;
- ActiveWebflow Designer is an Eclipse-based visual BPEL designer;
- Bexee BPEL Execution Engine is an open source, J2EE-based BPEL engine; it is also an environment for working on Business Process Modeling (BPM);
- Orchestra, developed by Bull and OW2, is an another open source middleware which provides a SOA Orchestration solution for handling long running, service oriented processes.
- ActiveBPEL Engine is a comprehensive BPEL runtime environment; ActiveBPEL is open source, released under the GPL, and it is written in Java;

In what follows, we will give more details on two particular open source BPEL engines: Orchestra and ActiveBPEL.
4.3.1.1 BPEL Engine: Orchestra v.4

Orchestra is a SOA Orchestration solution. In particular, Orchestra v.4 open source project is the newer generation technology for Workflow, Business Process Management and SOA-based applications. Orchestra v.4 has been released in May 2008 under the LGPL license, and it offers a unique open source package consisting in a BPEL engine, an administration console, and a designer and monitoring tool. Orchestra v.4 supports XPDL and BPEL standards, and it is the first solution to implement the BPM Process Virtual Machine (PVM) technology jointly developed by Bull, OW2 and JBoss/Red Hat communities.

Bull is the key actor in the development of Orchestra solutions and is providing a unified and professional set of services including worldwide support, training and expertise under its NovaBPM offering. Other contributors include INRIA, eXo Platform, EBM Websourcing and an active open source community named OW2. In particular, the OW2 consortium is an international open source community which includes R&D teams, industrials partnerships and research organizations, aiming at bringing permanent innovations and quick time to market.

BPM PVM applications adapt precisely to real life business needs, easily combining human workflow, application process and web applications. The Orchestra v.4 BPEL engine is provided on top of the PVM engine, as a standard language to be used in the case of web applications. The main features provided by the BPEL engine are as follows:

- support for BPEL 2.0, standardized by OASIS;
- Web 2.0 based process console;
- Eclipse-based editors;
- graphical BPEL designer;
- Java JEE deployment

Additionally, the Orchestra v.4 offers:

- a graphical BPM console based on Web 2.0 and Ajax technologies;
- a pluggable infrastructure allowing easy services configuration;
- a flexible architecture in which persistence, transaction, security, identity, asynchronous messages and timers notifications can easily be added with no intrusive integration in both development and production environment;

The resulting architecture [38], which is shown in Figure 6, is meant to develop, control and improve business processes and to design the interaction between services. Overall, Orchestra v.4 is a tool for the improvement of the “productivity and agility of the company”. It is worth noting that, thanks to the open source nature of the project, Orchestra’s architecture can be deployed and run on top of many different environments, such as Tomcat, Spring, Java application, JEE application server (JOnAS, Jboss, Weblogic, WebSphere, Oracle, and so on). In particular, there are three different versions which better suits to different customer needs:

- Lightweight version: deployable on any existing application as Tomcat, Swing applications…
- Entreprise version: deployable on any Java EE server (JOnAS, Jboss, Weblogic…) 
- “Sur mesure” version : adapted to any particular customer’s environment
Figure 7 better characterizes the Process Virtual Machine (PVM). PVM is a revolutionary technology for processes based applications. It has been delivered by Bull and Jboss as a generic process engine which allows to build on top embeddable, pluggable and extensible workflows and BPM solutions. PVM offers a conceptual model for processes, therefore it is not limited to the BPEL language, as it can be seen in Figure 8. On the contrary, PVM is meant for multi-language support, including BPEL and XPDL, and it is generic enough to support new additional languages, e.g., specific languages based on graphs. Furthermore, the architecture of PVM allows for the support of multiple pluggable services that can be defined by the company. Note that PVM is flexible in which it support both descriptive process models and executable processes like the ones described in BPEL, and it is highly customizable, by allowing the embedding of standard solutions as well as “sur mesure” company-defined solutions.
4.3.1.2 BPEL Engine: ActiveBPEL

The ActiveBPEL engine [39] is a BPM runtime environment for executing process definitions created according to the BPEL 1.1 and WS-BPEL 2.0 specifications. ActiveBPEL is a powerful open source instrument which contains a large set of tools for the process design, analysis, simulation and execution.

To be more precise, ActiveBPEL is a product family aiming at supporting the creation, testing, deployment and execution of composite software applications. In contrast with monolithic systems architectures of the past, composite applications are based on a services-oriented computing model. In ActiveBPEL, the interfaces to all application components are firstly described using standards like the Web Services Description Language (WSDL), and then components are woven into coherent process compositions. The objective of composite applications is to deliver a highly adaptive information architecture, and then to meet the evolving needs of the users.

The main features of ActiveBPEL are listed below.

- **Process editor canvas with diagrammatic and hierarchical view of process.**
  - Processes can be diagrammatically created on the canvas, and process element can be simultaneously displayed in hierarchical and diagrammatical forms. Additionally, source view is available for the BPEL code generated by ActiveBPEL.
  - ActiveBPEL allows for drag and drop icons onto a canvas to create a process and the related BPEL code. The engine also generates a task list for missing and invalid activity properties.

- **Web References.**
  - ActiveBPEL supports WSDL and schema files to be included as Web References view for automatic discovery and organization of all pertinent information stored in existing WSDL. Comprehensive searching is available to locate namespaces, messages, and other elements.

- **Management of Sample Data.**
  - For testing and simulation purposes, the engine automatically generates and manages sample data files for all WSDL messages

- **Automatic Task Generation (static analysis)**
  - ActiveBPEL generates a problem list for the debugging of incomplete or invalid BPEL constructs, both in the case of imported code and locally generated code.

- **Create Partner Link Types (WSDL extensions required for BPEL processes).**
  - The engine adds partner link types to an existing or new WSDL file by using the partner link type wizard.

- **Swimlanes**
  - Visual display of each partner’s role in the process to show a service is being invoked, received from or replied to.

- **Expression and query builders**
  - ActiveBPEL gives the user visual expression/query editing controls for building a wide range of scripts. ActiveBPEL’s expressions can be extended to include user-defined expression languages and custom functions.

- **Activity Properties**
  - ActiveBPEL groups required and optional activity attributes for easy selection, and allows the user to insert comments as well as correlation properties, compensation, and fault handling.

- **BPELets, for previously defined activity re-use in other processes**
  - Activities created with the process editor canvas can be saved to a custom palette for later use.

- **Simulation and Debugging**
The engine simulates process execution using sample data. It can be used to set breakpoints, step through or run the process. Remote debugging is also possible.

- **Process Deployment**
  - ActiveBPEL offers *deployment wizards* to guide the users deploy the user-defined process and for service binding to the process.

- **Migrate Processes to the latest BPEL version**
  - ActiveBPEL gives the possibility to update the syntax of the old code, so that old and non-conformant processes can be reshaped into the newest version of code supported by the current BPEL specification.

- **ActiveBPEL Extensions to WS-BPEL 2.0**
  - In addition to fully supporting the WS-BPEL 2.0 specification, ActiveBPEL supports some extra Process Level Extensions: a) the Compensation handler and the termination handler; b) the Query handling including Create XPath and Disable Selection Failure; c) Suspend, Break, and Continue extension activities.

- **ActiveBPEL for People.**
  - This is a plugin that installs an extension activity, called the People activity. This activity introduces human workflow into a BPEL process, when a process activity requires human decision-making.

There are actually different versions and releases of the ActiveBPEL engine. The basic version is the ActiveBPEL Community Edition Engine, while the richest version is the ActiveBPEL Enterprise. In particular, the ActiveBPEL Community Edition Engine is an open source implementation of a BPEL engine, written in Java and running on top of any standard servlet container such as Apache Tomcat. It reads BPEL process definitions, WDSL files and other inputs, and then it creates representations of BPEL processes. When an incoming message triggers a start activity, the engine creates a new process instance and starts it running. The engine takes care of persistence, queues, alarms, and many other execution details. The ActiveBPEL Community Edition Engine does not contain a visual designer that allows for easy and quick generation of BPEL orchestrations. It only supports the execution of previously-coded BPEL processes. The server supports the full complement of BPEL activities including event handling, exception handling and scope/compensation management.

Since hand-coding BPEL process definitions can be tedious and error-prone, several *Designers* have been proposed, including the ActiveBPEL Designer. ActiveBPEL Designer is a plugin to the Eclipse integrated development environment, and it provides a visual environment for creating, testing and deploying BPEL-based process compositions. It makes it possible to build processes by choosing partners, services and operations, and by defining how data flows among those entities. As one organizes icons on the Process Editor canvas, ActiveBPEL constructs valid BPEL code which is stored in XML format. An example of ActiveBPEL BPEL process is reported in Figure 2, and an example of BPEL code generated by the ActiveBPEL Designer in XML format is given in the snapshot reported in Figure 10.

Another designer has been developed by Active Endpoints, i.e., the very same ActiveBPLE team. This is the ActiveVOS Designer, that differs from the ActiveBPEL Designer in what the ActiveVOS is targeted to commercial solutions [39]. ActiveVOS Designer is an Eclipse-based development environment that allows the user to learn about the OASIS WS-BPEL 2.0 standard and the BPFL4People and WS-Human Task specifications. ActiveVOS Designer also allows developers to create BPMN (Business Process Modeling Notation) models that can be converted directly to BPEL.
ActiveBPEL for People

ActiveBPEL for People is an extension to WS-BPEL 2.0 that allows human workflow activities to be included within a BPEL process. ActiveBPEL for People is also based on the WSDL 1.1, WS-HumanTask 1.0 and WS-BPEL Extension for People (BPEL4People), Version 1.0 specifications.

In addition to receiving, replying, and invoking Web services, ActiveBPEL for People invokes a person to handle and complete a task. Similar to the behavior of an invoke activity, the person has to return output data to the BPEL process. ActiveBPEL for People contains an extension activity, called the People activity, which contains properties to do the following:

- Create a task definition
- Select potential owners for working on the task
- Select administrators to manage tasks
- Create workflow requirements, such as timeouts/deadlines and an expiration
A People activity is built by adding values for its properties. The People activity can be linked to any BPEL activity or contained within a scope or other container. One can add as many People activities to a BPEL process as needed.

As shown in Figure 11, when a BPEL process executes a People activity, the task defined by the activity is routed to the inboxes of all potential owners and administrators defined by the task. Each user signs into his or her inbox, and so can claim a task to work on, making it unavailable for other users. When the task owner completes the task, by submitting the required information into a form, the output data is sent back to the process so that the next activity can execute.

![Figure 11 – Description of a People Activity in a BPEL Process](image)

Since the People activity effectively invokes a person, a BPEL process containing a People activity is an asynchronous, and typically long-running process. As a consequence, tasks associated to a People activity have to be dealt with care, and special state machines have to be associated to them. Therefore, the BPEL extension defines the following states of a task after a BPEL process begins execution of the People activity:

- **Unclaimed**: the task is available for anyone designated as a potential owner.
- **Claimed**: one person reserved the task.
- **Started**: the task is in progress.
- **Completed**: the output data has been submitted and the owner declared the task complete.
- **Failed**: the owner provided fault data and failed the task.
- **Exited**: an error condition occurred, such as expiration or People activity termination.
### 4.3.2 SCXML Implementation on Apache

*Commons SCXML* is an open source implementation of the SCXML W3C standard. It is mainly composed of a Java SCXML engine capable of executing a state machine defined using a SCXML document, while abstracting out the environment interfaces.

Figure 12 (taken from the official site: http://commons.apache.org/scxml/) shows the normal process for creating and executing SCXML applications with the Apache implementation:

First a converter from UML to SCXML has been developed to allow users take advantage of UML graphical notation. Then the developer can complete refine the parameters of the state machine using any Java editor - like the ECLIPSE Java IDE, finally the state machine can be executed in a server and possibly connected to different kinds of terminals to activate the behaviour (for instance through a vocal platform controlled using VoiceXML).

### 4.3.3 SPATEL Engine (from IST SPICE)

The SPATEL Engine can be seen as a framework that implements *natively* the SPATEL language (see Section 4.1.3):

To the concept of Service Interface in SPATEL corresponds a Python class, sharing exactly the same structure: same visible operations and same exposed attributes. This class acts either as a client proxy for the service - a glue component connected to a potentially remote implementation - or as the "real" implementation, to be defined locally.

To the concept of State Machine in SPATEL corresponds a State Machine implemented in Python. Similarly to some VoiceXML [4] systems, the state machine is loaded into memory once at the
activation of the service. Then each session object - representing the usage of the service by a user - has a pointer to store its position in the execution of the state machine.

The SPATEL engine relies on an HTTP server to offer multi-threaded and asynchronous support. A session mechanism is explicitly maintained by the framework to allow keeping alive the context when dealing with long-running services.

Two forms of execution are supported: one uses CGI protocol, the other uses servlets on top of TOMCAT. In the first case, the HTTP server invokes Python CGI which rebuilds the saved context at each invocation. In the second case a Jython interpreter is used to connect Java and Python.

4.3.4 Semantic-based Frameworks for Service Composition

There are a huge number of approaches to service composition, particularly Web Service composition. An exhaustive explanation of the current state of the art is out of the current document’s scope. In general, service composition can be differentiated into orchestration and synthesis. Whereas service orchestration has to deal with coordinating services and service components, synthesis is the actual process of generating composite services. State of the art telecommunication Service Delivery Platforms (SDPs) make use of service enablers, which in this regard extend the notion of application enablers provided by exposure APIs such as Parlay X [2] and comprises all services attached to the Enterprise Service Bus (ESB). This provides the possibility of mapping a complete service life-cycle as an orchestration script.

The most well known orchestration language is WSBPEL 2.0. But there are other promising orchestration specifications as the Web Services Choreography Description Language (WS-CDL) [20] standardized by the W3C that is an XML-based interface description language describing the flow of messages exchanged by a Web Services interacting with other Web Services. Furthermore Business Process Modeling Language (BPML) has been proposed by Business Process Management Initiative Organization (BPMI), but BPMI [21] has dropped support for this in favor of BPEL and joined the Object Management Group (OMG) in 2005. A good comparison between BPEL for Web Services and BPML is given in [22], a good requirements analysis can be found in [23].

In general two approaches for Web Service compositions must be differentiated; the syntactic (XML based) and the semantics-based (ontology based) approach. Whereas BPEL and WS-CDL account for the syntactic composition of Web Services, OWL-S and WSMO account for the semantic-based Web Service composition. Conceptual differences between OWL-S and WSMO are explained in [24]. Apart from these already standardized mechanisms, theoretical models like automata, Petri nets and process algebras provide complementary mechanisms for automated service compositions.

Focusing on synthesis, mechanisms range from static service compositions to semi-automatic to automatic service compositions; static and automatic composition approaches are compared in [25]. Whereas static service composition requires manual generation of choreographies (a.k.a. workflows in BPEL) automatic service compositions are capable of discovering components and synthesizing composite services at run-time.

Static service compositions are already commonly in use. Synthesizing Web Services with the help of BPEL workflows already provides a huge range of service mash-ups. However these solutions require manual generation of workflows, increasing services time to market.

The current spectrum of automatic service composition approaches is broad. As Küster et al. in [26] summarizes, there exist several strategies to generate a service composition: Fulfillment of preconditions (graph search, forward chaining, backward chaining, estimated regression planning), generation of multiple effects (behavior based service composition, component based service composition) and overcoming of a lack of knowledge (creating conditional plans, searching while planning).
Although in the long term, automatic service composition are regarded as the ultimate goal of SOA, at the current stage there are many problems to be overcome for industrial usage. Amongst other shortcomings, established solutions like BPEL, WS-CDL, OWL-S and WSMO as explained in [27] provide no mechanisms to verify correctness of service compositions. In the mid-term, utilization of formal methods (automata, petri-nets, process algebras) might help to overcome these issues. Apart from that, increased automation usually comes with an increased architectural complexity of required components for automated synthesis, yet reducing service time to market. As an example, the Figure 13 shows the architecture of Web Service Modelling eXecution environment (WSMX), the reference implementation of WSMO.

As can be realized, the WSMX architecture is comprised of several components. The Web Service Modeling Toolkit (WSMT) [32] is a development environment enabling the development of Ontologies, Web Services, Mediators and Goals through WSMO. The WSMX Core allows for matching a goal that has previously be specified by a user to a service element that has previously been described in Web Service Modeling Language (WSML) [33]. If the matching was conducted successfully, the data mediator provides support for the transformation between ontologies of the requestor and the Web Service provider [34]. Reasoners such as Pellet [19] are capable of searching through ontologies and checking their consistencies. Altogether the WSMX framework allows for service discovery, service selection, mediation, service composition and invocation at runtime.

In conclusion, as stated in [35] and [30] compared to OWL-S, Semantic Web Services Framework SWSF [36], IRS-II and WSDL-S, the WSMO framework most comprehensively provides mechanisms that satisfy most of the requirements for a Semantic Web Service (SWS) framework. However this versatility comes with high complexity and a specific modeling ontology.

For the initial stages of PII, a semi-automatic service composition approach, at this stage, might be sufficient and feasible. Since neither can PII services tolerate long execution times, nor can currently automated mechanisms guarantee quality and correctness of automatically composed services.

### 4.4 Model Engineering

Model engineering is an important discipline in our context since resource definition and orchestration specifications relies on the manipulation and exploitation of static and behavioural models.

In this section we provide a basic introductory material.
4.4.1.1 Meta-modeling

Meta-modelling is the activity that constructs concepts within a certain domain. Classically to model a system, we will need first to have a modelling language to define our model. A meta-model plays this role: a model to model our concepts.

The simplest language is to have boxes to represent concepts and arrows between boxes to connect our concepts. The OMG (Object Management Group) standardization body has standardized the MOF (Meta Object Facility) as a generic meta-model to serve as language for defining specific models. This language uses object-oriented concepts: the boxes are classes that can contain operations and attributes and the arrows are class associations. Inheritance relations may exist between classes. In terms of UML notation the MOF corresponds to a subset of the UML class diagram.

The relationship that exists between a model and its meta-model is conformance: any element in the underlying level is defined in terms of the concepts defined in the upper level. This relationship can be generalized to the case of models and data: a data element can be treated as an instance of an element in the model level. This homogeneity in the mechanisms to represent models and data is exploited to build powerful code generation tools.

4.4.1.2 Operating with Models

In order for an object-oriented model to be treated by programs it is necessary to have interfaces (the list of operations that I can invoke when accessing an object). An important feature of object-oriented modeling techniques is the fact that the interfaces can be generated automatically from the model definition: for instance if we have a Car concept with an association to Gate concept, we will generate in Java or in C++ the classes Car and Gate, and the Car gate will have a 'getGates' operation. Interface generation allows implementing a model in different programming environments - like Java or C++.

4.4.1.3 Model Exchange

Object-oriented models are exchanged using an XML based standard named XMI (XML Metadata Interchange). The XML Schema that is used to exchange models conformant to a meta-model is automatically derived from the meta-model definition. In addition the parser specific to the XML Schema will be generated automatically.

4.4.1.4 The Power of Meta-modeling

In summary, the model definition, expressed in terms of its meta-model, will contain sufficient information for generating automatically not only the operational interfaces to navigate and update model instances but also the XML machinery to interchange models.

Similarly, due to the relationship between model instantiation and data instantiation (see Meta-modeling section), the data definition, expressed in terms of a object-oriented model will contain sufficient information to generate automatically not only the operational interfaces to navigate and update data instances but also the XML machinery to interchange the data.

4.4.1.5 Model Transformations

One of the major motivations of modelling is the capacity to apply automated processes to generate as complete as possible the code of a system or the code of an application. For this reason model transformation occupies an essential place. We can distinguish different kinds of transformations, like horizontal transformation (for instance, translating a formalism to another within the same level of abstraction) and vertical transformation (for instance, translating an abstract functional model into a more concrete implementation model). Also we can make the distinction between model to model transformations and model to code transformations.

Transformations can be developed in different ways. The most popular way is to use the operational interfaces (see previous section 'Operating with models') . Another alternative is to use dedicated
languages. The OMG has standardized a language specific for model to model transformation named QVT and another for code generation, named Mof2Text.
5 Specification

5.1 Introduction

The Testbed Orchestration System of Teagle has two macro components: The Request Interpreter and the Execution Engine. These two components interact with the rest of the Teagle system with a list of interfaces that are summarized in the diagram below.

The VCT Tool within the Request Interpreter macro component provides to the Execution Engine a specification script. This specification script is either edited manually using the Graphical Composition module or is generated using the Request Processor. In both cases policies are to be taken into account through the Policy Enforcement component.

The Execution Engine transforms the specification script as an executable script - through the Script Compiler. The Script Executor invokes intermediate proxies (Component Adaptor) which in turn are...
connected to the web services components residing in the accessed PTM. The details of this architecture and the role of each module of the request interpreter and composition engine are explained in the rest of this section. For the details of the Portal and the Repository macro components see Deliverable D3.2.

5.2 Entities and Reference Points

In this section we describe the functional specification of the two components of the Testbed Orchestration System.

5.2.1 Orchestration Frontend

The Orchestration Frontend plays a central role in satisfying the customer request and providing the desired functional environment. The Orchestration Frontend sub-modules are:

- Policy Evaluation (defined section 5.2.1.1)
- Creation Environment (defined section 5.2.1.2)
- Request Processor (defined section 5.2.1.3)
- The Creation Environment and the Request Processor are both part of the VCT tool

The following reference point need to be supported by the Orchestration Frontend:

- REP (defined section 5.2.1.4)
- POL (defined section 5.2.1.5)
- SPEC (defined section 5.2.1.6)

The following subsection will explain the mentioned sub-modules and reference points in more detail. Figure 15 gives an overview on how the Orchestration Frontend is connected to the other TEAGLE entities.

5.2.1.1 The Policy Evaluation Function

This functional entity allows the evaluation of policies stored in the Teagle repository. This service is offered via the reference point POL. The Open Mobile Alliance (OMA) has specified a Policy
Evaluation, Enforcement and Management Architecture [40] and our Policy Evaluation Function shall be aligned with the current OMA standardisation activities. Via the reference point REP, the Policy Evaluation Function shall be able to query the repository and obtain policy documents.

Policies shall be described in Common Policy Format [47]. As the RFC is very generic it might need to be extended to serve out needs.

5.2.1.2  The Creation Environment

This function is part of the VCT tool and shall allow for the graphical design and configuration of a VCT. This might be offered via a dedicated application (for example Eclipse [41] based) that needs to be downloaded and run locally or via a web based solution (creation client, e.g. [42]) offered by the Teagle portal. The environment shall allow for placing and interconnecting testbed components in a workbench in an intuitive drag-and-drop manner. By clicking on components, suitable properties menus shall allow for easy configuration of desired VCT entities. The environment shall offer templates for usual testbed setups and configurations that can be fine tuned by the customer if desired.

In order to enable the above functions, the Creation Environment needs a connection to the repository (reference point REP) in order query existing and available testbed components described by the testbed description system [51]. Furthermore policy evaluation (via reference point POL) allows the tool to check whether a certain customer is allowed to use certain resources and to what extent. For some functions such as policy evaluation and resolving of configuration dependencies, the Creation Environment shall rely on the Request Processor as described in the next subsection.

5.2.1.3  The Request Processor

This entity works in close cooperation with the Creation Environment. The graphical representation shall be separated from the logic of processing a certain request, resolving conflicts as well as checking potential prerequisites and dependencies of a given configuration/composition of resources. Therefore, the entire VCT tool entity is connected to the repository and the policy evaluation via the reference points POL and REP. The outcome of graphically designing (using the Creation Environment as part of the VCT tool) a desired VCT and resolving any constraints on the given configuration/composition (done by the Request Processor as part of the VCT tool) is an orchestration specification that is delivered to the Orchestration Backend via reference point SPEC as described in section 5.2.2.7.

5.2.1.4  The Reference Point REP

In addition to [51], the reference point REP shall be aligned with OMA specifications [40], [43], [44]. The REP interface shall be designed as a management interface for the policy repository. The operations that should be supported by the repository are: create, modify, view and delete. The OMA specifications provide a solution to this interface implementation using the XCAP protocol for management queries and a XDM [45] implementation. The XML Document Management Server (XDMS) [46], also specified by OMA, provides an extensible data model using XML schema which allows to easily access and manipulate XML documents and supports the XCAP protocol. The XCAP protocol supports HTTP as transport and uses XPATH-style URLs to target content using PUT, GET and DELETE. Data stored in a XDMS is grouped based on its Application Unique Id (AUID) as the first level and XCAP User Identifier (XUI) as the second level taken into consideration for grouping data. Policies reside under the AUID org.openmobilealliance.policy-commonpol.

5.2.1.5  The Reference Point POL

The reference point POL shall support exchanges between requestors for policy processing (the VCT tool) and the Policy Evaluation Engine. In correlation with the OMA specifications [40], [43][44], POL has a well-defined structure of requests and responses, where each request or response message always carries, from a protocol perspective, a binary large object, encompassing the collection of input parameters (in the case of a request message) or the collection of output parameters (in the case of a response message). The variability of potential input and output parameters is hidden from the
interface, by encoding all input or output parameters in the binary large object, according to a well-defined convention. Currently the specifications contain a few predefined templates about the way the binary large object should look like, but can be extended to any level.

The implementation of the interface has to fulfill the following basic requirements:

- Provide critical information based on which the policies will be fetched from the repository and enforced, for example customer identity or target identity.
- Provide further data on which policy defined constraints may apply.
- Understanding of the policy enforcement result codes (policy decision) by the requestor.

5.2.1.6 The Reference Point SPEC

The SPEC reference point is described in the Orchestration Engine section, sub-section 5.2.2.6.

5.2.2 The Orchestration Engine (Orchestration Backend)

The Orchestration Engine - also named Orchestration Backend - is responsible for implementing and executing customer requests originally edited or generated from the Orchestration Frontend module. The Orchestration Engine sub-modules are:

- Script Compiler (defined section 5.2.2.3)
- Script Executor (defined section 5.2.2.4)
- Script Launcher (defined section 5.2.2.2)
- Component Adaptor (defined section 5.2.2.5)

The following external reference points need to be supported by the Orchestration Engine:

- SPEC (defined section 5.2.2.6)
- TC (defined section 5.2.2.7)

The following internal reference points need to be supported by the Orchestration Engine:

- COMP (defined section 5.2.2.8)
- EXE (defined section 5.2.2.9)
- CALL (defined section 5.2.2.10)
- GEN (defined section 5.2.2.11)

Figure 16 gives an overview on how the Orchestration Engine is connected to the other TEAGLE entities. It is connected to the Orchestration Front End through the SPEC reference point (on the left) and connected to the Teagle Gateway through the TG reference point (on the right).
5.2.2.1 Rationale of Orchestration Engine architecture

The architecture of the Orchestration Engine has been defined taking into account the following two important considerations:

- To execute an orchestration definition that invokes one or more components enablers, it should be possible to switch easily from an implementation of an enabler to an alternative implementation without changing the original orchestration definition. Variability includes possible change in the communication protocol. This is useful for instance to support a change in the provider of a communication enabler like SMS sending, provided using SOAP by one provider and provided using simple HTTPS requests by another.

- It should be possible to change the technology used to execute an orchestration script without changing the original orchestration definition. This is useful to take into account variability and evolution in technology adoption in different companies. For instance a BPEL solution could be selected in some simple circumstances, whereas a state machine based workflow engine would be more appropriate when human intervention is predominant.

This yields to the idea of having a distinction between an "orchestration specification" script and an "executable execution" script, where the link between both is done through a compilation step. Also the notion of "component adapters" allows supporting component replacement in a flexible way. Meta-modeling and code generation techniques (see Section 4.4) are expected to be used in an implementation of the Orchestration Engine module.

Two main advantages of this architecture can be pointed out:

- The request processor component (see section 5.2.1.3) - in charge of transforming a user request into a script - will have its work simplified by the fact that he will not have to deal with platform specificities and communication protocols.

- It is relatively easy to change technical aspects of the interface of a service offered by a PTM (like the actual URLs and authorization method, or the communication protocol being used)
without impacting the validity of the abstract script. The abstract script only deals with a pure functional description of the interface of the web service.

The details of the role of each component and interface are explained in subsequent chapters.

5.2.2.2 The Script Launcher

The Script Launcher component receives the user request in the form of a specification orchestration script (using the SPEC reference point) and performs one or both of the two following actions:

- Compiling the specification script so that it becomes an executable orchestration script, through the Script Compiler (COMP interface reference point).
- Executing an executable orchestration script, through the Script Executor (EXE interface reference point).

The request of the user indicates whether an immediate execution is to be performed. If not, the script launcher simply compiles the script and stores internally the executable script for later usage. If the execution is requested, the script launcher looks whether there is a pre-existing compiled executable script that satisfies the request, and if found, invokes the executor component with the found executable script, possibly re-configured. Configurations and executable scripts are managed separately to enhance re-usability.

5.2.2.3 The Script Compiler

The Script Compiler is basically a code generator. It parses and transforms the specification script and generates code.

The following properties need to be fulfilled:

- The code generator produces complete code (the executable script) corresponding to the specified orchestration logic and generates, when not already available, component adapters (GEN interface reference point) for each invoked component found in the original specification script.
- A component adapter is an implementation class reflecting the operations and configuration attributes of a local or remote component. Component Adaptors are described in more details in Section 5.2.2.5.
- For homogeneity purposes, the executable script is generated as a composite component - structurally and conceptually not different that other component adaptors. The execution script is the behaviour of the 'main' operation of the generated composite component.

5.2.2.4 The Script Executor

The Script Executor is responsible for executing an executable script previously built using the script compiler. An executable script is visible in the form of a composite component that has a 'main' operation. Executing the script means hence instantiating the composite component and entering the 'main' operation.

The - generated - behaviour of the main operation follows one of the two paradigms:

- Either it is a simple list of actions to be executed with run-to-completion semantics. Typical actions are a conditional test or the invocation of the component adapters representing each of the involved remote components.
- Or it is the code reflecting the execution of a state-machine, with the possibility of interruptions (wait states) and the receptions of asynchronous events.

The Script Executor should hence be able to support the execution of state machines and be able to maintain session information during potential interruption - waiting for explicit user input or external events.
Component Adaptors

Any service that can be invoked within an orchestration script is represented in the orchestration engine by a component adaptor. The component adaptors are generated by the Script Compiler based on service interface descriptions. The code of the component adaptor may be revised and/or completed manually by a developer. Services are, in most cases, remote which means that in that case the component adaptors in the orchestration engine are proxies for accessing the remote services.

In our context of testbed federation, remote components are typically the configuration web services hosted by a PTM (exposing single or various resources). However, it may also represent any arbitrary utility taken from the internet - such as a currency converter facility.

The "glue code" to invoke a remote component is generated automatically as part of the output of the Script Compiler. The actual binding settings like the address URLs of the web services to access are taken from separated configuration files.

From a technical point of view, a component adaptor is an implementation class (for instance a Java/C++/Python class) that reflects the operations and configuration attributes of a service interface.

For flexibility and easy substitution of services, the code generation of component adaptors may implement the following variance mechanism: for each of the declared operations in the service interface there will be three implementation variants generated. By convention, variant v0 is a "fake" implementation that does nothing except returning a valid empty value. The following variant v1 is a 'locally' defined implementation - empty when generated, but possibly completed by the developer to test the service. The second and last generated variant is the "glue" code connecting to the "real" remote component (SOAP request construction in case of standard web services).

This variance mechanism allows executing scripts in a debug, test or simulation mode. It can be also used to move from an implementation to another, like replacing the implementation from a provider A to the same functionality implemented by a provider B.

Reference point SPEC

The SPEC reference point is used to connect the orchestration front-end to the orchestration engine.

It is materialized in the form of a XML file containing a header with a list of various informative fields. We provide here a high-level specification of what should be part of the content of this file.

The header should contain at least the following fields:

- Author of the request: A user id denoting the user originating the request
- Request description: A documentation text describing the user request.
- Action: The action to be performed: execute or compile (see Script Compiler in section 5.2.2.3).
- Script kind: The kind of script:
- The list of operations (in most cases a unique one) with the signature (types of arguments and result).
- For each operation the body kind: action sequence, state machine or none (see Script Executor in section 5.2.2.4). None means that the body is not provided.

The script content itself, should be provided in textual form (protected through a XML CDATA tag). It consists of the definition of the operations listed in the header, with the form:

```
operation <opname> (<typedarglist>) : <returntype> { <body> }
```

In the case of 'action sequence' script kind, the syntax is a very restricted subset of Java script where we have basically variable declarations, operation invocation, if then else and while instructions.

In the case of 'state machine' the body contains sections for each defined state.

Here's an example showing the style of notation:

```
behavior TranslateWeatherInLannion::go(
```
Each state node is characterized by a section of type

\[
\text{<statekind>} "\text{<stateid>}" \{
\text{<actionsequence>}* \\
(\text{transition} \rightarrow \text{<targetnode>})*
\}
\]

Where \text{<statekind>} represents various kinds of states and \text{<actionsequence> is a list of actions following the same syntax than for "action sequence" script.}

5.2.2.7 Reference point TC

The TC reference point is used to connect the orchestration engine to the PTC. More precisely it is materialized by the binding that exists between a component adapter (a proxy classes generated by the Script Compiler) and the corresponding remote component. The default communication protocol selected by the project is SOAP and the default assumption is that all PTC exposed components are web services.

The WSDL interfaces of the accessed remote components should follow a uniform convention to facilitate the generation of binding code.

5.2.2.8 Reference point COMP

The COMP reference point is used to connect internally the Script Launcher to the Script Compiler. It is materialized by an interface with the following signature:

\[
\text{ScriptCompiler::compile(scriptId:scriptSpec:XMLDoc) : status}
\]

The scriptId is an ID for unique identification of the script in the orchestration system. The interface will be exposed also in the form of a HTTP POST.

5.2.2.9 Reference point EXE

The EXE reference point is used to connect internally the Script Launcher to the Script Executor. It is materialized by an interface with the following signature:

\[
\text{ScriptExecutor::execute(scriptId:Id) : status}
\]

The scriptId is an ID for unique identification of the script in the orchestration system. The interface will be exposed also in the form of a HTTP POST.
5.2.2.10 Reference point CALL

The CALL reference point represents the connection between the Script Executor and a Component Adaptor. It is simply materialized by the presence of a operation call within the executable script, where the operation call refers to an operation of an existing component adapter. An instantiation of the given component adaptor is required previous to the call. This instantiation is normally done at the beginning of the script.

5.2.2.11 Reference point GEN

The GEN reference point represents the relationship between the script compiler component and the component adaptors. Component adaptors are generated from the service interface descriptors exposed by the PTM to Teagle. In some cases - when using non standard interfaces - the generated coded will need to be manually edited.

5.3 Requirement Mapping

In this section we provide the relationship between requirements identified in section 2 and the components in the architecture.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Statement</th>
<th>Primary Components Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>##D3.3-Basic##M.100</td>
<td>The orchestration system should be able to orchestrate resources and configure components distributed in various testbeds that are part of the federation.</td>
<td>Orchestration Engine, PTMs</td>
</tr>
<tr>
<td>## D3.3-Basic##M.150</td>
<td>The orchestration system should provide facilities to help users (partners and customers) to formulate testbed requests that are complete - that is to say, containing sufficient information for automated execution.</td>
<td>Creation Environment</td>
</tr>
<tr>
<td>## D3.3-Advanced##M.200</td>
<td>The orchestration system should provide facilities to help users (partners and consumers) to formulate testbed requests that are valid from the point of view of policy rules (like authorization to access some restricted components).</td>
<td>Request Processor, Policy Evaluator</td>
</tr>
<tr>
<td>## D3.3-Advanced ##M.250</td>
<td>The orchestration system should provide adaptation facilities to allow replacing a call to a service component to a call to another component providing similar functionality, even when the exposed interfaces of both components differ.</td>
<td>Component Adaptors</td>
</tr>
<tr>
<td>## D3.3-Advanced ##M.300</td>
<td>The orchestration system should provide adaptation facilities to allow replacing a call to a service component to a call to another component providing similar functionality, even when the exposed interfaces of both components differ.</td>
<td>Script Compiler</td>
</tr>
<tr>
<td>## D3.2-Basic##M.350</td>
<td>The orchestration system may offer the possibility to orchestrate freely available service components exposed as web services coming from the internet.</td>
<td>Component Adaptors Script Executor</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>## D3.3-Basic##M.400</td>
<td>The Orchestration System should allow PII Partners to create testbed requests.</td>
<td>Creation Environment</td>
</tr>
<tr>
<td>## D3.3-Basic##M.450</td>
<td>The Orchestration System should allow PII Customers to create testbed requests.</td>
<td>Creation Environment</td>
</tr>
<tr>
<td>## D3.3-Advanced##M.500</td>
<td>The orchestration system should support the checking of policy conformance when a request is created.</td>
<td>Policy Evaluator</td>
</tr>
<tr>
<td>## D3.3-Advanced##M.550</td>
<td>The orchestration system should support delayed execution of a testbed request - implying resource reservation - in addition of supporting immediate execution.</td>
<td>Script Launcher</td>
</tr>
<tr>
<td>## D3.3-Basic##M.600</td>
<td>Testbed requests and orchestration specifications should be stored for future testing environment reproduction.</td>
<td>Creation Environment, Repository</td>
</tr>
</tbody>
</table>
6 Example

6.1 HSS Example Introduction

6.1.1 HSS Overview

A Home Subscriber Server (HSS) is defined in an IMS system to manage user profiles and associated routing rules. It is a master user database that supports the IMS network entities that actually handle calls. It contains the subscription-related information (subscriber profiles), performs authentication and authorization of the user, and can provide information about the subscriber's location and IP information. It is similar to the GSM Home Location Register (HLR) and Authentication Centre (AUC).

An HSS component is implemented by Fraunhofer FOKUS as part of the Open Source IMS Core project. Figure 17 shows the positioning of HSS component in the IMS architecture.

![Figure 17 - IMS Overall Components](image)

6.1.2 Manual Configuration

We provide below the different manual steps that are needed to configure the HSS manually without Teagle machinery.

(1) Pre-requisites:

- MYSQL database
- 2 scripts exist to load default database (for HSS + default users)
- `mysql -u root -p < FHoSS/scripts/hss_db.sql`
- `mysql -u root -p < FHoSS/scripts/userdata.sql`

(2) Edit DiameterPeerHSS.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- HSS Server config -->
```
<DiameterPeer
FQDN="hss.open-ims.test" ← Per domain name for HSS
Realm="open-ims.test" ← Need to be set to home domain.
This is resolved to I-CSCF within the same domain
Vendor_Id="10415"
Product_Name="JavaDiameterPeer"
AcceptUnknownPeers="1"
DropUnknownOnDisconnect="1"
Tc="30"
Workers="4"
QueueLength="32"

> <Peer FQDN="icscf.open-ims.test" Realm="open-ims.test" port="3869" />
<Peer FQDN="scscf.open-ims.test" Realm="open-ims.test" port="3870" />
← Need to be set to the Diameter peers that should be contacted (-> CSCFs)

<Acceptor port="3868" bind="127.0.0.1" />
← The Diameter port to be used
<Auth id="16777216" vendor="10415"/> <!-- 3GPP Cx -->
<Auth id="16777216" vendor="4491"/> <!-- CableLabs Cx -->
<Auth id="16777216" vendor="13019"/> <!-- ETSI/TISPAN Cx -->
<Auth id="16777217" vendor="10415"/> <!-- 3GPP Sh -->
<Auth id="16777221" vendor="10415"/>

</DiameterPeer>

(3) Edit Hibernate properties

### MySQL

# hibernate configuration
hibernate.dialect=org.hibernate.dialect.MySQLDialect
#hibernate.connection.driver_class=org.gjt.mm.mysql.Driver
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://127.0.0.1:3306/hss_db
hibernate.connection.username=hss
hibernate.connection.password=hss ← Data base details
hibernate.connection.isolation=1

(4) Edit HSS properties

# FOKUS HSS Properties file
#--------------------------------------------------------
# host & port : specify the IP address and the port where Tomcat is
# listening, e.g. host=127.0.0.1; port=8080;
host=127.0.0.1
port=8080
← Set to HSS IP or domain name. The port is for the web configuration
interface that can be accessed with a browser

(5) Edit DNS

If DNS is used: configure DNS with home domain. Resolve HSS name and CSCF
names
6.2 Models for Configuration

Three main concepts relates to the definition of the HSS provisioning facility.

- The HSSService represents the HSS service from a functional point of view, independently of configuration concerns.
- The HSSConfigurationService is a dedicated service that exposes operations to allow clients to configure the HSSService.
- The HSSServiceConfiguration represents the data structure that need to be manipulated for realizing HSS provisioning.

The model below shows how three concepts relates to the general information model (DEN-NG):

![Figure 18 - HSS Configuration Entities](image)

Here's how this conceptual model relates to DEN-ng:

- The ServiceConfiguration is a subclass of the Configuration class in the DEN-ng.
- The HSSService, strictly speaking is not an immediate subclass of CustomerFacingService, but a subclass of CustomerFacingServiceAtomic, which in turn, according to DEN-ng is a subclass of CustomerFacingService.
- Also the ConfigurationService is a subclass of ResourceFacingServiceAtomic (instead of directly being a subclass of ResourceFacingService).

The Figure 19 depicts the detailed model of the configuration data. This reflects relevant parameters that exist in the various configuration files that are part of the implementation of the HSS.
Figure 19 - Configuration Data for HSS

From the point of view of DEN-ng model all classes suffixed by "Conf" are sub-classes of the Conflet class.

6.3 Preparing the HSS Testbed Request

The different modules within the orchestration front-end are responsible for assisting a user to build testbeds requests that satisfy their needs in terms of resources and that can be understood and hence executed by the orchestration engine.

Basically, using the Creation Environment function of the VCT tool (see Section 5.2.1.2), the user will select a default configuration template for HSS. He will adjust the template to its precise needs by possibly changing default values (like authentication parameters of the HSS server) or by including other resources in its request (through drag and drop facilities). He will then be able to check compliance of its request to policy rules - like ensuring he has access rights. This is done through specific buttons linked to facilities provided by the request processor (section 5.2.1.3). At then, when the first shot of request editing work is done, he will call the request processor to generate the HSS provisioning script (see next section). The process is iterative: after testing the execution of the script the designer may want to change some values within the configuration.

The output of the orchestration front-end that is delivered to the orchestration engine are of two kinds: the configuration data and the orchestration script. Both data is passed to the orchestration engine using XML (reference point SPEC).
The configuration script is an instance of the declared configuration model (see Figure 19). Below is a possible XML representation excerpt, in line with the "manual configuration" provided in section 6.1.2.

```xml
<serverConf host="127.0.0.1" port="8080"/>
<peerConf FQDM="hss.open-ims.test" realm="open-ims.test">
  <auth>
    <authConnection id="16777216" vendor="10415"/>
    <authConnection id="16777216" vendor="4491"/>
  </auth>
  <contactedPeers>
    <peerUsage FQDM="icscf.open-ims.test" realm="open-ims.test" port="3869"/>
    <peerUsage FQDM="scscf.open-ims.test" realm="open-ims.test" port="3870"/>
  </contactedPeers>
</peerConf>
<databaseConf>
  <connection url="jdbc:mysql://127.0.0.1:3306/hss_db" username="hss" password="hss"/>
</databaseConf>
```

This configuration file contains all the values that can be potentially changed by the testbed user when dealing with HSS provisioning. In respect to the manual configuration in section 6.1.2, the user does not need anymore to edit directly each of the component configuration files (Diameter config, hibernate property file and so on). In addition as explained before, the user is assisted all along by the VCT tool to perform valid configuration assignments.

The orchestration script, which is one of the two outputs of the request preparation, is generated using the compact textual form (see SPEC reference point, section 5.2.2.6), and illustrated at the end of the next chapter.

### 6.4 HSS Provisioning Script

In this section we provide a possible orchestration specification to be produced by the Orchestration Front End as a result of the provisioning request done by a Teagle user. This orchestration script reflects the provisioning steps presented in Section 6.1.2.

Four entities are involved: the HSSProvisioningOrchestrator service is the entity representing the orchestration, the HSSConfigurationService is the proxy representing the corresponding remote service hosted by the PTM, a HSSServiceConfiguration instance is the actual configuration - conformant with the HSSServiceConfiguration structure (see Figure 18) and finally the DataConfManager which is a locally defined helper entity to retrieve specific pieces of the configuration data.

Notice that in this initial example only a PTM is involved (the one known in the script as "PTM/focus"), but nothing prevents having in a script references to different services coming from various PTMs (using the instruction createProxy(<service-name>,<target-PTM>).

This orchestration specification (see SPEC reference point in Section 5.2.2.6) is a specification of a composite service with a fixed interface and a list of helper entities and data types. In our case, the user request is materialized in the form of a service named

The orchestration class, materialized by the HSSProvisioningOrchestrator interface has a "performProvisioning" operation that contains the logic of the orchestration. This operation accepts as input parameter the HSS configuration data. Figure 20 shows its interface as well as the definition of the helper entity DataConfManager:

![Figure 20 - The Generated HSS Orchestration Interface](image)

The logic of "performProvisioning" operation references the HSSConfigurationService remote service. Figure 21 gives the interface - which we assume was necessarily defined before the user made his request of the HSSConfigurationService. The position of this class in the information model (which is based of DEN-ng) is explained in Section 6.2.

![Figure 21 - The HSS Configuration Service](image)

We can notice that every major step of the HSS provisioning process is represented here by an operation. This fragmentation allows to create potentially other variants of the provisioning process from outside the PTM supporting the HSS resource. Another less flexible option for the HSSConfigurationService could be to expose a unique single-piece "configure" operation, with the complete logic hard-coded.

The following figure shows the orchestration logic (the behaviour of "performProvisioning" operation). It is also given in graphical UML-based notation, although there is an equivalent textual form for it (see SPEC reference point 5.2.2.6). It shows the sequencing of the four steps. Each service operation call can be preceded by a local computation (the white rectangle) to achieve any needed formatting or computation.
Below, we provide its representation in terms of the textual syntax (using the state machine notation):

```
behavior
HSSProvisioningOrchestrator::performProvisioning(configdata:Conflet) {
  initialization:
  var HSSCONF = createProxy("HSSConfigurationService","PTM/Fokus");
  var DATAM = new DataConfManager(configdata);
  bodysection:
  init 'Initial' {
    transition -> actionseq 'node1';
  }
  call 'node1' {
    HSSCONF.loadSQLDatabase();
    transition -> 'node2';
  }
  call 'node2' {
    var peerConf = DATAM.retrieveByPath("peerConf")
    var hybConf = DATAM.retrieveByPath("databaseConf")
    var serverConf = DATAM.retrieveByPath("serverConf")
    HSSCONF.editDiameterPeer(peerConf)
    HSSCONF.editHibernate(hybConf)
    HSSCONF.editHssProperties(serverConf)
    HSSConf.prepareDns()
  }
}
```

**Figure 22 - Orchestration Script logic**
According to the Orchestration Engine architecture this script is compiled into running code. In addition to this configuration files are produced containing the mapping definition to web services (like providing the URLs of the remote HSSConfigurationService service, represented in the script by the variable HSSCONF). These mappings can be changed manually when necessary to reflect possible changes in the environment.
7 Conclusion

This document presented a high-level specification for the Testbed Orchestration system. It identifies
the list of major components that together form the orchestration system: a front end with editing
capacities as well as powerful derivation facilities (interpreter processor) to treat end-user testbed
requests, and the back-end, named the orchestration engine, which provides a flexible infrastructure
for executing the script orchestration specifications produced by the front-end. The specification
provides a list of sub-components and the list of reference points internal to the orchestration system
or external in respect to the rest of Teagle tool.

Apart from this specification the document provides the list of requirements that were used as input as
well as a detailed state-of-the-art section on different formalisms and tools available for service
orchestration.

It is expected that during the duration of the PII project the architecture of the testbed orchestration
system will be refined and also that more technical definitions for the list of entities and reference
points will be provided as an entry point for the implementation activities.
References

U. Küster, M. Stern, B. König-Ries, “A Classification of Issues and Approaches in automatic Service Composition”, 1st Int. Workshop on Engineering Service Compositions (WESC05) at ICSOC,05, Amsterdam, Netherlands, 2005


Mick Kerrigan, Adrian Mocan, Martin Tanler and Dieter Fensel: The Web Service Modeling Toolkit - An Integrated Development Environment for Semantic Web Services (System Description), Proceedings of the 4th European Semantic Web Conference (ESWC), June 2007, Innsbruck, Austria


The Orchestra website: http://orchestra.objectweb.org

The Active Endpoints’ website: http://www.activevos.com

Open Mobile Alliance Specification, Policy Evaluation, Enforcement and Management Architecture, OMA-AD-Policy_Evaluation_Enforcement_Management-V1_0-20080805-C

Eclipse Website, http://www.eclipse.org/


Open Mobile Alliance Specification, Policy Evaluation, Enforcement and Management Callable Interface (PEM-1) Technical Specification, OMA-TS-PEEM_PEM1-V1_0-20081014-C


[52] PII D2.1 Technical infrastructure specification, Dec 2008