Project no. FP7- 216556  
**Instrument:** Integrated Project (IP)  
**Objective ICT-2007.1.2:** Service and Software Architectures, Infrastructures and Engineering

---

**Deliverable D.A3a**  
**SLA-aware Service Management**

**Keywords:**  
Service Level Agreement, Service-Oriented Infrastructure, Architecture, Models

**Due date of deliverable:** 31\textsuperscript{st} July 2010  
**Actual submission to EC date:** 10\textsuperscript{th} September 2010  
**Start date of project:** 1\textsuperscript{st} June 2008  
**Duration:** 38 months  
**Lead contractor for this deliverable:** SAP  
**Revision:** V.1.0 (September 10\textsuperscript{th}, 2010)

---

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Document Status

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable Lead</td>
<td>Tariq Ellahi (SAP)</td>
</tr>
<tr>
<td>Reviewer 1</td>
<td>Gabriele Zacco (FBK)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Miha Vuk (XLAB)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Joe Butler (Intel)</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>June 24th, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>July 2nd, 2010</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>July 22nd, 2010</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>August 13th, 2010</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>September 10th, 2010</td>
</tr>
</tbody>
</table>

### Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP</td>
<td>Tariq Ellahi, Ulrich Winkler, Wasif Gilani</td>
</tr>
<tr>
<td>ENG</td>
<td>Paolo Zampognaro</td>
</tr>
<tr>
<td>PMI</td>
<td>Sam Guinea</td>
</tr>
<tr>
<td>CITY</td>
<td>Howard Foster</td>
</tr>
<tr>
<td>FBK</td>
<td>Michele Trainotti</td>
</tr>
<tr>
<td>TID</td>
<td>Beatriz Fuentes, Alfonso Castro</td>
</tr>
<tr>
<td>Intel</td>
<td>Víctor Bayón</td>
</tr>
</tbody>
</table>

### Notices

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2009 by the SLA@SOI consortium.

* Other names and brands may be claimed as the property of others.

---

This work is licensed under a [Creative Commons Attribution 3.0 License](https://creativecommons.org/licenses/by/3.0/).
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>24.06.2010</td>
<td>All</td>
<td>First version of the deliverable finalized</td>
</tr>
<tr>
<td>0.1</td>
<td>24.06.2010</td>
<td>All</td>
<td>Document submitted for internal reviews</td>
</tr>
<tr>
<td>0.2</td>
<td>07.07.2010</td>
<td>All</td>
<td>Reviewer 1’s comments incorporated into the document</td>
</tr>
<tr>
<td>0.3</td>
<td>30.07.2010</td>
<td>All</td>
<td>Reviewer 2’s comments incorporated into the document</td>
</tr>
<tr>
<td>0.4</td>
<td>10.09.2010</td>
<td>All</td>
<td>PMT review comments incorporated into the document</td>
</tr>
<tr>
<td>1.0</td>
<td>10.09.2010</td>
<td>Tariq</td>
<td>Document finalized for official submission to the commission</td>
</tr>
</tbody>
</table>
Executive Summary

SLA@SOI aims at addressing the SLA management problem from a holistic perspective. The objective is to design a multi-layer SLA management framework for SOA based application landscapes. The focus of work package A3 is the software service layer and aims at designing SLA aware service management capabilities for the overall SLA management framework. In this regard, WP A3 has divided the service management into a number of distinct yet complementary and interconnected working tracks. This deliverable document provides description of the activities, progress and achievements during the course of Y2. Building on the work carried out during Y1, work documented in this deliverable is targeted for the various cases. A brief summary of these activities are given in the following paragraph.

Firstly, A3 engaged in modelling related activities. The modelling aspects addressed were SOA modelling and software landscape modelling. SOA modelling investigated modelling of service component along with the non-functional properties as well as the service component behaviour which is leveraged by the design time prediction process. Software landscape modelling, on the other hand focused designing a metamodel which can be used to capture information about the service and software related artefacts. Additionally, landscape meta-model incorporated packaging and deployed related aspects to be used by service provisioning process.

Secondly, A3 continued work on the dynamic service binding and composition related activities. During Y1, design and architecture of a Dynamic Orchestration Engine was focus of Y1 activities. Building on this an implementation of the DOE was the main focus on Y2 activities. This document presents the DOE implementation specific details.

Last but not least, A3’s work focused on runtime monitoring and management of SOA based application landscapes. This document presents details of Monitoring Manager which is responsible for translating SLA guarantee terms into monitoring rules which can be observed during runtime to detect potential SLA violations. Additionally, the document discusses the process monitoring capabilities of the SLA@SOI framework.
# Table of Contents

1 Introduction .................................................................................................................. 10

2 Contribution Overview ................................................................................................. 11
   2.1 Key Innovations........................................................................................................ 11
   2.2 Framework Contributions ...................................................................................... 12
   2.3 Task-level activities ............................................................................................... 13

3 Software Service Manager ............................................................................................ 15
   3.1 Architecture ............................................................................................................ 15
   3.2 Implementation ...................................................................................................... 18
   3.3 Domain Specific Customization ............................................................................ 18

4 Enhanced SOA Modelling ............................................................................................. 21
   4.1 Business Continuity Management ......................................................................... 23
   4.2 Enhanced SOA Modeling Requirements ................................................................. 26
   4.3 Conclusion and Future Work ................................................................................. 27

5 Landscape Modelling ...................................................................................................... 28
   5.1 Landscape Metamodel ............................................................................................. 29
   5.2 Software Landscape ................................................................................................ 34
   5.3 Conclusion and Future Work ................................................................................. 36

6 Dynamic Service Discovery and Composition ............................................................... 37
   6.1 Dynamic Orchestration Engine (DOE) ................................................................. 37
   6.2 Future Work ............................................................................................................ 45

7 Unified SOA Manageability Interface .......................................................................... 46
   7.1 The Unified Manageability Interface ...................................................................... 46
   7.2 Integration with the SLA@SOI Framework ............................................................ 49
   7.3 Managing the DOE ................................................................................................. 49
   7.4 Conclusions and Future Work ................................................................................. 51

8 Advanced SOA Management ......................................................................................... 51
   8.1 Monitoring System Overview ............................................................................... 51
   8.2 Monitoring Manager ............................................................................................... 53
   8.3 Process Monitoring ................................................................................................ 64
   8.4 Adjustment: Autonomic Adjustment ..................................................................... 73

9 Multiprovider/Multidomain Service Management ......................................................... 84
   9.1 Overview ................................................................................................................ 84
   9.2 Scenarios ................................................................................................................ 85
   9.3 Framework extensions and implementation .......................................................... 95
   9.4 Future Work ............................................................................................................ 96

10 Conclusion .................................................................................................................... 97

11 References .................................................................................................................... 98
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>SoftwareServiceManager Architecture</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The BCM lifecycle as depicted in [28]</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Landscape Metamodel [Compact View]</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4</td>
<td>ServiceType Element</td>
<td>30</td>
</tr>
<tr>
<td>Figure 5</td>
<td>ServiceImplementation</td>
<td>31</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Detailed View of ImplementationArtefact Element</td>
<td>32</td>
</tr>
<tr>
<td>Figure 7</td>
<td>ServiceBuilder Element</td>
<td>34</td>
</tr>
<tr>
<td>Figure 8</td>
<td>ServiceInstance Element</td>
<td>34</td>
</tr>
<tr>
<td>Figure 9</td>
<td>SoftwareLandscapeManager Interface</td>
<td>35</td>
</tr>
<tr>
<td>Figure 10</td>
<td>WS-BPEL process: partners interactions model</td>
<td>38</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Several binding rules for the same partner role</td>
<td>38</td>
</tr>
<tr>
<td>Figure 12</td>
<td>DOE Architecture</td>
<td>39</td>
</tr>
<tr>
<td>Figure 13</td>
<td>DOE admin interface</td>
<td>40</td>
</tr>
<tr>
<td>Figure 14</td>
<td>addPartnerRoleBindingRule – sequence (1/2)</td>
<td>40</td>
</tr>
<tr>
<td>Figure 15</td>
<td>addPartnerRoleBindingRule – sequence (2/2)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Class diagram: Binder internal relationships</td>
<td>41</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Run time: interactions (1/2)</td>
<td>43</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Run time: interactions (2/2)</td>
<td>44</td>
</tr>
<tr>
<td>Figure 19</td>
<td>BME conceptual model and serialized instance example</td>
<td>44</td>
</tr>
<tr>
<td>Figure 20</td>
<td>The Unified Manageability Interface</td>
<td>46</td>
</tr>
<tr>
<td>Figure 21</td>
<td>The Manageability Agent in the context of the SLA@SOI framework.</td>
<td>49</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Managing the DOE</td>
<td>50</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Overview of Monitoring System in SLA@SOI</td>
<td>52</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Overview of the MonitoringManager Activity</td>
<td>54</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Process Flow of Monitoring Manager</td>
<td>55</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Sample Grammar Implementation for Agreement Term Parsing</td>
<td>56</td>
</tr>
<tr>
<td>Figure 27</td>
<td>MonitorSelection Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>Figure 28</td>
<td>SelectMonitor Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Sensor Component with Monitoring Features</td>
<td>58</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Partial SLA Java Object Code for an AgreementTerm</td>
<td>58</td>
</tr>
<tr>
<td>Figure 31</td>
<td>Reasoner Component with Monitoring Features</td>
<td>59</td>
</tr>
<tr>
<td>Figure 32</td>
<td>The MonitoringManager Implementation Packages</td>
<td>60</td>
</tr>
<tr>
<td>Figure 33</td>
<td>The MonitoringManager Core Interface Packages</td>
<td>61</td>
</tr>
<tr>
<td>Figure 34</td>
<td>RTML v2 language definition</td>
<td>67</td>
</tr>
<tr>
<td>Figure 35</td>
<td>RTML v2 architecture</td>
<td>70</td>
</tr>
<tr>
<td>Figure 36</td>
<td>Architecture of the SLA@SOI wrapper</td>
<td>72</td>
</tr>
</tbody>
</table>
Figure 38: MAPE-K Architecture ................................................................. 74
Figure 39: High-level view of the proposed architecture ..................... 76
Figure 40: Agent Class Diagram ............................................................... 78
Figure 41: Task Class Diagram ................................................................. 79
Figure 42: Configuration Class Diagram ................................................. 80
1 Introduction

This document provides an overview of the SLA-aware Service Management architecture and details about the various building blocks which enable such service management. The work described in this document is part of the overall SLA Management framework being developed by the SLA@SOI project. More specifically, work described in this document addresses the SLA management related aspects at the software / service layer of the IT stack.

This document serves as a formal progress report (deliverable D.A3a) for presenting the work carried out within work package A3 at project month 26. This deliverable presents progress on particular architectural building blocks (Service Management) of the overall SLA@SOI’s SLA Management framework. The overall architecture is described in the deliverable D.A1a and other architectural building blocks are presented in deliverables D.A2a (Business Management), D.A4a (Infrastructure Management), D.A5a (SLA Foundations) and D.A6a (Predictable Systems Engineering). Work presented in this document is driven by the requirements described in deliverable D.B1a and derived from various use cases. These use cases employ the technology developed in this and other work packages. Elaborate discussion on these use cases can be found in deliverables D.B2a-D.B7a.

SLA-aware Service Management addresses a number of aspects. First, modelling related activities focus on specifying metamodels for representing software and service landscapes for effective and efficient service management for large and complex landscapes. Secondly, a unified manageability interface is required for facilitating streamlined interactions with heterogeneous elements of the SOA landscapes. Thirdly, a dynamic orchestration engine enables composite service execution with the capability to perform dynamic service bindings to ensure compliance with the service level agreements established with the customer. Fourth, a large part of service management focuses on monitoring and runtime adjustment. Monitoring is essential to comprehend the execution status of the services and software elements. In this regard, two aspects must be addressed; 1) monitoring granularity i.e. monitoring the atomic services as well as the composite service or process monitoring 2) the monitoring apparatus should be configured in accordance to the guarantee terms specified in the SLAs. These guarantee terms should be translated into monitoring rules which should be used by the monitoring apparatus to observe the state and compliance of software and service elements. Regarding the runtime service adjustment, based on the monitoring information, the services might be operating in violation of the guarantee terms as specified in the SLAs, and hence actions should be taken to adjust the service levels by reconfiguring the service parameters.

The remainder of this document organization is as following. Chapter 2 presents a brief overview of the contributions from various tasks of work package A3. More specifically the scientific innovations of the work done are highlighted. The architecture of the SoftwareServiceManager is presented in chapter 3 with descriptions of the architectural building blocks. Chapter 4 and 5 discuss modelling related aspects addressed in this work package. Chapter 6 describes the dynamic orchestration engine and unified SOA manageability interface is presented in chapter 7. Service monitoring and adjustment is the topic of chapter 8 which discuss monitoring of atomic as well as composite services and the service adjustment. Finally, multiprovider/multidomain service management is the topic of chapter 9 followed by conclusions and bibliography.
2 Contribution Overview

This chapter provides an overview of the contributions that have been made by the work package A3 in terms of key innovations and contributions to the overall framework. Furthermore it summarizes the actual activities and progress at task-level for the reporting period.

2.1 Key Innovations

Key innovations in work package A3 with respect to individual tasks are described in the following paragraphs:

Task A3.1 provides enhanced modelling methodology in the SLA@SOI project to address business continuity aspects. This task extends existing models (such as deployment models, service composition models, business process models) to enable design time risk and dependency analysis. These extended models are used for business continuity related design-time decision support based on simulations and analytics. Such a decision support is needed to derive non-functional continuity properties and continuity related service level agreements (e.g. Maximum Tolerable Outage Time), to investigate business impact of failed SOA systems, to mitigate risks, and to devise optimised recovery plans and SOAs.

Task A3.2 addressed the landscape modelling at the software service layer. The main innovation of the activity is the capability to capture model elements beyond the core entities i.e. software, service and their relationships. Landscape metamodel designed in the task enables the service providers to represent monitoring configurations, service packaging and service dependencies. This information can be used by the SLA managers to identify the best combination of service elements that should be used for ensuring SLA compliance during service runtime. Moreover, service monitoring features enable the planning components to understand and reason about the monitorability of the guarantee terms as well as the capability to configure monitoring elements on the fly during service provisioning process.

Task A3.3 extends a widely adopted open source BPEL engine to implement SLA aware dynamic template based composition and which is moreover exploitable in multi-tenant applications to reduce the waste of resources. Key aspects of such dynamic orchestration engine include: 1) Abstract processes are described using standard WS-BPEL language while SLATs (SLA Templates) are used to describe services that will be dynamically discovered/selected and bound. 2) Different binding configurations can be associated to different process instances of the same deployed process on the base of specific conditions (multi-tenancy). The association can be made even at run time (run time re-configuration).

Task A3.4 provides a generic and unified interface for managing services. With respect to the management approaches that exist in literature, we propose a generic approach that leverages the service abstraction to enable the management of the whole application stack (Web services, service compositions, infrastructure, etc.). It provides a unified way to access service management-specific information such as the configurations of the sensors and effectors that have been deployed to the system. It also provides a unified API for interacting with these sensors and effectors at runtime. The generic and unified APIs bridge the gap between the technology-agnostic SLA@SOI framework and the domain-specific details involved in managing a concrete service instance.
Task A3.5 addresses the advanced SOA Management and in this regard a number of topics were under investigation including, service monitoring, process monitoring and service adjustment. Details of the key innovations in the work performed in these areas are given in the paragraphs below.

Regarding service monitoring, whilst existing approaches to service monitoring provide powerful mechanisms for performing the basic checks of service compliance with SLAs, they fall short of providing adequate support when either services are replaced or terms of an SLA change dynamically. In Task 3.5 we specifically address this issue by providing monitorability checks of SLA terms and conditions, and the dynamic configuration of monitoring infrastructure that the SLA@SOI framework provides. In Year 2 we have developed a Monitoring Manager component for the SLA@SOI framework to provide mechanical support for these checks and configurations in planning and optimisation of overall SLA monitoring.

Process monitoring concerns the monitoring of these services that implement complex business processes, specified, e.g., in BPEL or other orchestration languages. The key innovation for this research line is the definition of specialized notations, mechanisms and tools for managing event correlation at the process level. Indeed, process-level monitoring has to be able to handle in a very flexible and efficient way different forms of event correlation, including cross-instance correlation (i.e., correlation of events generated in different executions of the same process), cross-process correlation (i.e., correlation of events generated in different processes or sub-processes), and cross-layer correlation (i.e., correlation of process-level events with events coming from different levels of the service application). Efficiency in particular requires to extend existing approaches with specialized mechanisms and associated tools that are specifically designed to handle event correlation. Specialized languages and notations are also needed to specify the correlations in a way that is directly usable by these mechanisms and tools.

Adjustment in Task A3.5 presents a conceptual architecture for autonomic SLA enforcement. A generic implementation is also provided, aiming to be used for adjustment purposes throughout the project. While current approaches to SLA enforcement are domain-specific and usually based on monolithic platforms, Task A3.5 proposes a framework to build a flexible SLA enforcement layer. Given the subject of this task, domain-specific extensions are unavoidable, but these aspects are kept well isolated and are easily modifiable, through a policy-based mechanism.

Task A3.6 explores how to the SLA@SOI framework can be used in a multiprovider environment. In this kind of scenario, customer and service provider (namely, Service Aggregator or Service Broker) sign a SLA for a given service, which in turn is composed by other atomic services belonging to different 3rd parties. The relationship between the Service Aggregator and the service providers is also governed by an SLA. This task provides an automatic way for the Service Aggregator to modify the composite services when one of the 3rd parties services present a continuous malfunctioning, being substituted by a similar service offered by another provider.

### 2.2 Framework Contributions

Work package A3 contributions to the SLA@SOI framework include the following:

First and foremost, A3 provides the design and implementation of SoftwareServiceManager which is responsible for managing software services. The design provides an extensible architecture which is capable of
accommodating domain specific extensions enabling the users to customize SoftwareServiceManager according to the use case requirements. Also, part of the SoftwareServiceManager is the SoftwareLandscape which provides a repository of landscape models containing information about the services being offered by the service provider.

Secondly, A3 delivers a specialized ManageabilityAgent for managing composite services. This is an extended orchestration engine which provides dynamic service binding functionality. Bringing this dynamic orchestration engine provides out of the box capabilities for a number of use cases from B-Line.

Thirdly, work package A3 delivers a monitoring manager which translates the guarantee terms to low level monitoring rules which can be observed by the sensors to identify potential SLA violations. Complementary to this is the service adjustment apparatus which takes appropriate actions to adjust service configurations and parameters to ensure SLA compliance at the software service level.

2.3 Task-level activities

The main activities and progress at task level is summarized below.

**Task A3.1** Enhanced SOA Modelling aims at designing Business Continuity Management solutions to: (1) identify potential threats to an organisations’ business support system, services and operations, (2) assess the business impact of a threat, estimate probabilities and compute risk exposures, (3) determine strategies and responses to these threats, and model an IT business continuity plan to overcome or mitigate a possible business disruption. Business Continuity Managers struggle to conduct a comprehensive, thorough and valid analysis for Service Oriented Architectures. The authors of the Essent Netwerk [27] study, encountered several sever problems, which hinder their conduction of a comprehensive and thorough Business Continuity Analysis for enterprise grade Service Oriented Architectures.

**Task A3.2** Landscape Modelling designed a landscape metamodel which is capable to capturing detailed information about various aspects of the services being offered by the service providers. In addition to this, a repository to store and query these landscape models was developed which is available as part of the SoftwareServiceManager.

**Task A3.3** Dynamic Service Discovery and Binding focused the orchestration engine realized to implement the general architecture defined at the end of the first year (see [1]). Such architecture aims to offer an efficient mechanism to make dynamic binding (i.e. to bind steps of a process at run time) and to provide an execution environment for abstract processes described with a new methodology focused on a better separation between process description and (abstract) component services description.

**Task A3.4** Unified SOA Manageability Interfaces investigated the design of a generic unified manageability interface for managing services through appropriately deployed sensors and effectors. The generic manageability interface extends the work achieved in the project’s first year, and aligns it with the design of the overall SLA@SOI framework. The unified manageability interface enables the dynamic management of the lifecycle of the service: from initial deployment and startup, to advanced runtime activities such as monitoring and adjustment.

**Task A3.5** Advanced SOA Management addressed a number of aspects around service and process monitoring and service adjustment. The work done in this task included:
CITY’s work focused on the Monitoring Manager. The Monitoring Manager (MM) coordinates the generation of a monitoring configuration of the system. It decides, for an SLA model instance it receives, which is the most appropriate monitoring configuration according to configurable selection criteria. A monitoring configuration describes which components to configure and how their configurations can be used to obtain results of Guarantee Terms.

FBK’s work focused on the extensions and novel developments that have been undertaken during the 2nd year of the project on the process level monitoring framework. The extensions have been driven, on the one side, by the progress in the development of the whole SOA monitoring and management framework in the project; on the other side, by the outcomes of the evaluation of the Y1 process level monitoring framework, which has been undertaken exploiting the industrial case B6, “E-Government”. The most important extension that has been implemented has solved one of the most serious limitations of the previous version of the framework, namely the limited capability to do event correlation.

TID’s focus was on The Autonomic Management of SOA components to build systems with self-CHOP capabilities (self-CHOP stands for self-Configuration, self-Healing, self-Optimization and self-Protection). That is, systems able to configure themselves, able to make an auto diagnosis of themselves and recover from failures, able to find the optimized way to execute themselves and able to protect themselves of possible external threats.

**Task A3.6 Multiprovider / Multi-Domain Service Management** focused on enabling the SLA Management framework to be deployed and facilitate scenarios involving more than one service providers spanning multiple organizational domains.
3 Software Service Manager

The holistic SLA@SOI architecture is described in details in deliverable D.A1a. On the highest level, the architecture is composed of core framework, ServiceManagers (infrastructure and software), deployed service instances with their ManagebilityAgents and MonitoringEventChannels. The core framework encapsulates all functionality related to SLA management. Infrastructure and SoftwareServiceManagers contain all service specific functionality. The deployed service instance is the actual service delivered to the customer and managed by the framework via ManagebilityAgents. MonitoringEventChannels serve as a flexible communication infrastructure that allows the framework to collect information about the service instance’s status. SoftwareServiceManager components provide the capabilities to manage the software layer of the stack. Following section describes the SoftwareServiceManager architecture.

3.1 Architecture

SoftwareServiceManager architecture consists of a block of generic components that cater for managing a variety of software services as well as provide for domain specific extensibility mechanisms to plug in service specific management functionality. This section describes the detailed architecture of the SoftwareServiceManager. The architecture of the SoftwareServiceManager is shown in Figure 1.

**Figure 1: SoftwareServiceManager Architecture**

The following subsections provide elaborate explanation of two aspects of the software service manager. 1) The interactions supported by the SoftwareServiceManager and 2) description of the individual components realizing the functionality exposed via the interfaces supporting the interactions.
3.1.1 Interactions

SoftwareServiceManager supports two interaction stereotypes from SoftwareSLAManager during various points of negotiation, provisioning and runtime of the service lifecycle. These interactions are explained in the following paragraphs.

<<prepare_software_services>> allows SoftwareSLAMangers to search for service implementations provided by SoftwareServiceManager and retrieve information about their dependencies. Furthermore, it provides a unified access point to the provisioning operation of software services including reservation and booking. This interaction is complemented by <<manage_software_services>> which provides provisioning and management functionality. This interaction is represented in the architecture diagram using the IPrepareSoftwareServices interface and provides the following functionality to SoftwareSLAManager:

- Get all available implementations of a particular service type. The implementation descriptors include information about the service dependencies, monitorability, configuration options, and provisioning information such as lead times.

- Create a ServiceBuilder that allows the SLAManager to configure the service instance.

- If necessary, check for the availability of (limited) resources such as software licenses and make a time restricted reservation of these. Inform the caller about the success or failure of the reservation.

- If necessary, book/confirm a previously successful reservation and acquire all necessary resources for the specified time. Inform the caller about the success or failure of the booking.

<<manage_software_services>> enabled SoftwareSLAManager to interact with SoftwareServiceManager to create new software service instances. Additionally, SoftwareSLAManager can execute service lifecycle management related operations through this relationship. This interaction complements the functionality provided through the <<prepare_software_services>> interaction. This interaction is represented in the architecture diagram using the IManageSoftwareServices interface and provides the following functionality to SoftwareSLAManager:

- Trigger the provisioning process that executes the necessary actions to provide access to a booked service instance.

- Release and cleanup of a provisioned service instance. Disables access to a provisioned service instance.

- Change the implementation or dependencies of an instantiated service.

- (re-)configure of the service.

- (re-)configure of the monitoring system.

- Query possible change actions.
3.1.2 Components

SoftwareServiceManager architecture comprises of four main components: SoftwareServiceManagerFacade, SoftwareServiceLifecycleManager and finally the BookingManager and SoftwareLandscape. These components described in details in the following paragraph collectively realize the previously mentioned interactions: \(<\text{prepare\_software\_services}>\) and \(<\text{manage\_software\_services}>\).

SoftwareServiceManagerFacade

The **SoftwareServiceManagerFacade** is the primary subcomponent encapsulating management functionality for services throughout their lifecycle. SoftwareServiceManagerFacade facilitates the \(<\text{manage\_software\_services}>\) and \(<\text{prepare\_software\_services}>\) interaction which is realized through IManageSoftwareServices and IPrepareSoftwareService interface of the SoftwareServiceManager. It follows Gamma's Facade pattern and hides the internal structure of SoftwareServiceManagers from callers. In order to realize the SoftwareServiceManager's functionality, it makes use of the SoftwareServiceLifecycleManager, the SoftwareLandscape, and the BookingManager as well as the required interfaces IManageabilityAgentFacade and IProvisioning.

SoftwareServiceLifecycleManager

The **SoftwareServiceLifecycleManager** component serves as a factory for creating new service builders for various service implementations that SoftwareServiceManager can support and as a management facility for status information of all services. Besides a *createBuilder* operation which initialises a service's life cycle, the SoftwareServiceLifecycleManager allows retrieving the service's status information, its instance's metadata, its ManageabilityAgentFacade, and its ServiceBuilder object.

SoftwareLandscape

The **SoftwareLandscape** keeps track of available service types, service implementations and of provisioned service instances. The externally accessible interface IPrepareSoftwareServices allows the SoftwareSLAManager to query possible service types as well as service implementations for a particular service type. Additionally, the SoftwareLandscape provides metadata about all service implementations. The metadata includes (but is not limited to): dependencies of the implementation to other services, a description of the service realization, information related to provisioning, and available monitoring / manageability configuration options. However, SLAMangers can only read information about the available service implementations. The implementations and their metadata are maintained by the Service Provider. In addition to information about service implementations, the SoftwareLandscape administrates metadata about service instances of the implementations under its control. The metadata of these instances may contain the service endpoint, a reference to the corresponding manageability agent, and external services used.
**BookingManager**

The BookingManager component is responsible for maintaining and managing the software resource booking and reservation related information. For example BookingManager can maintain information about software licenses required for software service provisioning. BookingManager component realizes the *checkCapacity*, reserve and book operations of the IPrepareSoftwareServices & IManageSoftwareServices interface.

### 3.2 Implementation

WP A3 delivered a default implementation of the various building blocks of SoftwareServiceManager. Additionally, extensibility mechanisms are put in place for allowing the use cases to provide domain specific management logic for customization of the SoftwareServiceManager suitable for the native services being considered in the respective use cases. These extensibility mechanisms can also be used by anybody adopting SLA@SOI framework. This section provides a brief overview of the default implementations details, customization capabilities are presented in the next section. The default implementation is driven from two perspectives: 1) generic nature of the software components which can be applied to a variety of domains 2) implementation of the SoftwareServiceManager component for the Open Reference Case demonstrator.

A default implementation of SoftwareServiceManagerFacade is provided which acts as a front-end of the SoftwareServiceManager and mediates access to the other components like SoftwareLandscape, BookingManager etc as well as the domain specific customization logic. SoftwareServiceManagerFacade hides the internal structure and delivers transparent access to the default implementations as well as domain specific custom management logic plugged in by the use cases or other users.

SoftwareLandscape is implemented and integrated with the SoftwareServiceManager. Detailed description of the SoftwareLandscape implementation is provided in section 5. SoftwareLandscape is a generic component and its current implementation sufficient for the variety of use cases.

Default implementation of BookingManager and SoftwareServiceLifecycleManager is provided with respect to the Open Reference Case and this implementation can be overridden by providing domain specific logic.

### 3.3 Domain Specific Customization

As described previously, SoftwareServiceManager delivers extensibility mechanisms which can be used to introduce domain specific customizations to fit the needs of the use case. This section describes how domain specific customizations can be achieved through these extensibility mechanisms. The domain specific customization is enabled through two interfaces which use cases can implement to plug in domain specific service management logic. These interfaces are IProvisioning and IManageabilityAgentFacade. These interfaces are described in the following paragraphs.

**IProvisioning**

The interface IProvisioning is encapsulates actions for deploying, configuring and startup & shutdown related functionality of software services. The functionality defined in IProvisioning is predominantly domain and software service specific. Therefore, the SoftwareServiceManager adopts a plug-in type mechanism
whereby domain specific plug-in (implementing the IProvisioning interface) can be deployed so that SoftwareServiceManager can use these plug-ins to perform software service configuration, startup & shutdown operations. IProvisioning is domain-specific interface that is to be implemented by all use cases employing a SoftwareServiceManager. This interface encapsulates the basic functionality for starting and stopping service instances. Furthermore, it allows the SoftwareServiceManager to access (or create) a domain-specific ManageabilityAgents for an existing service instance. Finally, all implementations of this interface encapsulate the knowledge necessary to determine the endpoints for their services. The IProvisioning interface contains the following methods:

**startServiceInstance**

```java
void startServiceInstance(ServiceBuilder builder,
                          Settings connectionSettings,
                          String notificationChannel)
                          throws ServiceStartupException,
                          MessagingException;
```

Starts a new service instance according to the configuration specified in the builder object. Once the service is available a notification is sent over the channel specified.

**Parameters:**

- **builder** - The builder object used to set up the service instance. The builder must be fully specified, i.e., all dependencies to external services have to be resolved.
- **connectionSettings** - Connection settings for the message-oriented middleware hosting the notification channel.
- **notificationChannel** - The channel used to notify the responsible parties once the service's provisioning has been completed.

**Throws:**

- **ServiceStartupException** - Thrown if an error occurs during the initialization of the provisioning process.
- **MessagingException** - Thrown if the messaging system cannot be reached.

**stopServiceInstance**

```java
void stopServiceInstance(ServiceBuilder builder)
```

Stops a running service instance.

**Parameters:**

- **builder** - The builder object used to create the service instance.
**getEndpoints**

```java
List<Endpoint> getEndpoints(ServiceBuilder builder)
```

Determines the endpoints of a service instance created by the given builder object.

**Parameters:**
- `builder` - The builder object used to create the service instance.

**Returns:**
- Endpoint(s) of the service

**getManageabilityAgent**

```java
IManageabilityAgentFacade getManageabilityAgent(ServiceBuilder builder)
```

If available, this method returns a facade to the manageability system of the given, domain-specific service.

**Parameters:**
- `builder` - The builder object used to create the service instance.

**Returns:**
- ManageabilityAgent of the service instance

**IManageabilityAgentFacade**

This interface provides a generic access point for the SoftwareServiceManager to use case specific ManageabilityAgents. To correctly implement this interface, you need to consider the following: There exists exactly (!) one IManageabilityAgentFacade per service instance. If multiple service instances are managed by a single agent, the requests are dispatched by the Facade. The methods of the Facade are kept on a general level. Basically, all more specific actions of a domain-specific ManageabilityAgent have to be realized through the "executeAction" method.

**getInstance**

```java
ServiceInstance getInstance()
```

**Returns:**
- Returns the ServiceInstance that is controlled by the ManageabilityAgent

**getSensorSubscriptionData**

```java
List<SensorSubscriptionData> getSensorSubscriptionData()
```

**Returns:**
- Returns a list of available sensors and where to find their data
**configureMonitoringSystem**

```java
void configureMonitoringSystem(IMonitoringSystemConfiguration configuration)
```

Applies the given configuration to the monitoring system of the managed ServiceInstance. The given configuration replaces the previous one (if any).

**Parameters:**

*Configuration* – Configuration to be applied

---

**deconfigureMonitoring**

```java
void deconfigureMonitoring()
```

Disable the currently applied monitoring configuration

---

**executeAction**

```java
IEffectorResult executeAction(IEffectorAction action)
```

Execute domain-specific action for the adjustment of a ServiceInstance

**Parameters:**

*action* - Domain-specific implementation of the IEffectorAction interface that contains information about what action to execute and its parameters.

**Returns:**

The result of the action in domain specific format

---

## 4 Enhanced SOA Modelling

New emerging technologies, such as Service Oriented Architectures, Virtualisation, WebServices and Cloud Computing creates whole new business ecosystems, in which business processes depend more than ever on services provided by partner organisations. Often, disruptions in services delivery affect immediately thousands of business customers and consumers. For example, on January, 4th 2010, SALES FORCE, a company offering online enterprise support services, experienced an outage for over one hour which effected 68'000 business customers [26]. Another example would be PAYPAL, a service to process online payments. PAYPAL was down for 4.5 hours worldwide on August, 4th 2009. PAYPAL usually processes 2'000 USD per second for its customers [27].

Disruptions do not only have a financial impact or cause losses in reputation; they may also have legal consequences. In particular key industry sectors, such as energy, gas, oil, pharmacy or finance, have to demonstrate business continuity competence, which is required by regulations and laws. A study to quantify Information and Communication Technologies (IT) business continuity risks at ESSENT NETWERK, a Dutch electricity and gas distributor, revealed, that a four hour outage of an IT service might result in a withdrawal of the licences to operate and would eventually take ESSENT NETWERK out of business [@@TODO].
In a truly Service Oriented World, where mission-critical business processes of enterprises and organisations rely on the service delivery by partner organisations, three aspects will become more important. First, Service Level Agreement do not only establish a mutual agreement on the quality of a service to deliver. They also act as reinsurance policy and risk mitigation strategy of service consumers to back-up their business in case the service provider is not able to deliver a service in agreed quality. The second aspects that will become more important are common standards, regulations and certifications which demonstrate that a service provider is capable of delivering services in quality specified by the respective certification or standardisation body. A good example would be the BS-25999 [28], a standard that defines IT Business Continuity Management.

Business Continuity Management aims to: (1) identify potential threats to an organisations’ business support system, services and operations, (2) assess the business impact of a threat, estimate probabilities and compute risk exposures, (3) determine strategies and responses to these threats, and model an IT business continuity plan to overcome or mitigate a possible business disruption.

However, in Business Continuity Managers struggle to conduct a comprehensive, thorough and valid analysis for Service Oriented Architectures. The authors of the Essent Netwerk [27] study, encountered several severe problems, which hinder their conduction of a comprehensive and thorough Business Continuity Analysis for enterprise grade Service Oriented Architectures. Their problems were:

- **SOA Complexity**: SOA based Enterprise IT landscapes are complex adaptive systems comprising hardware, software, facility and human components. SOA landscapes evolve over time; for example new servers are added or software is upgraded. No single person has a complete understanding of all involved SOA landscapes nor oversees dependencies between single IT elements.

- **Model and Analysis of Temporal Fault Propagation**: Small changes in an operational SOA landscape may have severe consequences. To estimate the business impact, a full understanding of the behaviour and analysis of all interactions, between processes, services, applications and hardware has to be performed. Due to the complexity of SOA landscapes it is hard to connect and analyse the outage of a single element to the overall damage on business processes.

- **Modeling people behaviour**: People are essential elements and resources in business processes, SOA landscapes and are crucial for recovery plans. Furthermore labor costs are usually higher than hardware or software costs. Hence, any Business Continuity management should take interaction with people and people costs into considerations.

- **Model and Tool diversity**: Different modeling techniques are used to describe SOA landscapes. For example, business processes are modeled in work-flow languages whereas software is modeled in Unified Modelling Language (UML) diagrams. This adds additional hurdles to Business continuity analysis.

Wijnia et al. concluded [27] that Business Continuity Management should become part of SOA landscape planning and modeling in the first place. These observations are shared by Moura et al. In "Research challenges of Business-Driven IT management" [29] Moura et al. concluded that there exists a complexity problem and that "the complexity of modeling the interactions of incident, problem, [...] and continuity management has not been tackled yet". The authors also state that the modeling and analysis of "how IT affects the business [...] is a good research problem to think about."
The SAP Business Continuity Management team acknowledges these problems as well and confirmed that existing methodologies and technologies for Business Impact Analysis and Dependency Modeling of SOA based IT systems are not sufficient enough to meet the needs of operational risk management.

Therefore we decided to investigate these problems and to research a modelling and analysis methodology that incorporates operational risk management and business continuity planning into Service Oriented Architecture Modeling. We call this modelling and analysis methodology Enhanced SOA Modeling.

The reminder of this report is organized as follows: The following section provides brief introduction to Business Continuity Management, succeeded by a section where we elaborates and discusses requirements of a Enhanced SOA Modeling Methodology. Finally, we outline the planned research activities for the following year.

4.1 Business Continuity Management

Business Continuity Management is standardised by the British Standards Institution (BSI) and formal defined as follows:

A holistic management process that identifies potential threats to an organization and the impacts to business operations that those threats, if realized, might cause, and which provides a framework for building organizational resilience with the capability for an effective response that safeguards the interests of its key stakeholders, reputation, brand and value-creating activities.[28]

![Figure 2 The BCM lifecycle as depicted in [28]](image)

The BSI standard includes the business continuity management lifecycle as depicted in Figure 2. The business continuity lifecycle is a closed-loop and comprises four groups of activities, which are: understanding the organisation,
determining Business Continuity Strategies, developing and implementing a BCM response, and exercising, maintaining and reviewing BCM arrangements. All four activities are organised by a fifth activity, the BCM Program Management, which initiates business continuity related projects, assigns responsibilities, observes and manages activities, conducts training, and provides documentation.

4.1.1 Understanding the Organisation

This activity aims to provide information that enables Business Continuity Managers to (i) identify critical business processes, stakeholders, assets, resources and internal/external dependencies (ii) identify potential threats to critical business processes and (iii) assess and evaluate potential damages or losses that may be caused by a threat to critical business processes. Business Continuity Managers refer to these activities as Business Impact Analysis (BIA) and Dependency and Risk Analysis (DA).

**Business Impact Analysis:** Business Impact Analysis is a crucial tool used by a business continuity manager to better understand the business of an organisation to make informed decisions. However, for large enterprises operating at global scale this analysis can take a lot of effort. Nowadays business continuity managers tend to gather information about business processes via questionnaires and interviews. This method is time-consuming, cumbersome and error-prone. Large and globally operating enterprises deploy up to hundreds of different business processes, spanning different function areas of an enterprise. Some of those business processes are complex, not all processes are well documented and in some cases only semi-structured. Business processes can change and develop over time or can exist in regional variants. For example, Coca Cola's business process management program includes over 50 cross-functional business processes in 36 variants, involving 6500 users in all functional areas [30].

Business continuity managers are not able to develop a current, accurate and complete analysis solely based on interviews and questionnaires; they require tooling support that aligns business process management and business impact analysis. Disruptions of business processes may have a financial impact, legal consequences or may cause effects on other business values and indicators, such as reputation, customer satisfaction or customer churn rates. Those values are known as business level Key Performance Indicators (KPI)s. Research in management shows an importance of customer focus and customer satisfaction. These nonfinancial indicators are called leading indicators: if customers are not satisfied, they will eventually go to competitors. Losses in those business values are therefore indicators of future business impacts, even if the current financial effect of an incident is negligible. As stated previously, authors of [27] discovered that a disruption in a specific business process of Essent Netwerk does not cause significant financial losses; however, in case of a disruption the government would probably withdraw the licences to operate. Business continuity managers should consider those financial and nonfinancial business values alike.

**Dependency and Risk Analysis:** The business continuity manager does not only has to understand the effects of an adverse incident on a business, he also has to understand dependencies among business processes, dependent resources and possible root-causes of a adverse incident. To carry out a dependency and risk analysis the business continuity manager has to identify stakeholders, assets, resources and relations to external parties on which a business process depends. Furthermore, he needs to understand and analyse the nature and significance of dependencies. For example, the analyst needs to understand how a broken air-conditioning unit may affect the datacentre and servers deployed in this datacentre. Jeff Dean, a Google fellow, stated that there exists a 50% chance per year that a datacentre overheats, and once overheated most servers power down
4.1.2 Determine Business Continuity Strategies

First, the business continuity manager has to understand, estimate, quantify and classify risk probabilities of threats. Second, he has to determine the acceptable minimum level of business process operations to mitigate the business impact. He also has to specify acceptable timeframes (BCM metrics) in which a normal level of operations has to be restored, such that the organisation can continue to deliver products and services. Business Continuity Metrics: Return Time Objective (RTO) and Recovery Point Objective (RPO) are examples of BCM metrics. The RPO defines "the maximum amount of data loss an organization can sustain during an event. It is also the point in time (prior to outage) in which systems and data must be restored to" [32]. RTO defines the "target time for resumption of product, service or activity delivery after an incident" [33]. In addition there exist other BCM related metrics; some are commonly used in IT Business Continuity Management, others are organisation or business dependent. A discussion and overview about BCM related metrics is given in [34][35][36]. Apart from quantitative metrics, some qualitative metrics are of importance as well (e.g. "security") [37]. The business continuity manager should be able to specify measurements (e.g. "time to security breach") to be able to compare and analyse quantitative metrics as well.

4.1.3 Developing and implementing a BCM response

The business continuity manager can put four responses to a risk in place: to remove a risk, to mitigate the adverse effects of a risk, to transfer the risk responsibility to third parties, or to do nothing at all and simply accept the risk. If he decides not to accept the risk, the business continuity manager has to develop a recovery plan. The recovery plan details the steps to be taken to maintain or restore business operations to defined levels of operations within given timeframes.

Recovery Plans: A business process manager should also be able to analyse a recovery plan. He has to prove that a recovery plan is valid. Furthermore he has to make sure that the recovery plan is robust and is less dependent on uncertain estimations made in the dependency and risk model. This could be achieved by a sensitivity analysis. A recovery plan might depend on resources and might be vulnerable to threats itself. Therefore, a business continuity manger should be able to model and analyse dependencies and threats to recovery plans in a similar way as he is able to do with business processes. The business continuity manger also has to make sure that recovery plans are dead-lock or live-lock free.

Service Level Agreements: If the business continuity manager decides to transfer the risk to third parties or if a business process depends on external service providers in the first place, the Business Continuity Manager may specify Service Level Agreements (SLAs). This means, he has to detail objectives encoded in a SLA and he has to specify penalties in case the service provider is not able to fulfil the agreements made. The exact penalties should be derived from the BIA. The manager needs a methodology to translate business level BCM metric and objectives down to individual IT elements BCM objectives. For example, he needs to translate the RTO of a business process to RTOs of external services.
4.1.4 Exercising, maintaining and reviewing BCM arrangements

These activities enable the organisation to demonstrate that business recovery plans are complete, coherent, current and correct. Exercising and reviewing helps business continuity managers to understand the organisation better and gives them opportunities to identify improvements in business recovery plans, business continuity strategies and business impact analyses. Hence, these activities close the Business Continuity Management Lifecycle.

4.2 Enhanced SOA Modeling Requirements

As stated previously several problems have to be addressed:

- **Model and Tool diversity**: Different modeling techniques are used to describe SOA based IT landscapes. For example, business processes are modeled in work-flow languages whereas software is modeled in Unified Modelling Language (UML) diagrams. At the infrastructure and operational layer, the Common Information Model (CIM) is used to model systems, networks, applications and services. For each layer domain specific tools are available that support modeling, analysis and planning. However, these different tools and models are not easy to integrate and existing approaches struggle with the complexity and chaotic behaviour of SOA landscapes and the overwhelming number of business processes and business process variants. As a result, Business Impact Analysis are inadequate, Dependency and Risk Analysis incomplete, Recovery Plans not reliable and Service Level Agreements are not well defined. The underlying research question is: How to design a comprehensive and generic ICT Business Continuity Management, Analysis and Planning model methodology that is suited for complex SOA landscapes and organisations with numerous business processes? The proposed Enhanced SOA Modeling Methodology should (i) provide an integration approach, (ii) be comprehensive, (iii) support Business Impact Analysis, (iv) support Multi-Level Resource Dependencies and Risk Analysis and (v) provide means to analyse and validate response plan as well as support for SLA translation.

- **Integration**: The proposed methodology should provide a mean to integrate existing landscape modeling tools and BCM models to provide a coherent and comprehensive picture on a SOA based IT landscape.

- **Comprehensiveness**: The proposed methodology should provide a means to model resources and resource dependencies, threats and effects, as well as recovery plans.

- **Business Impact Analysis**: The approach under discussion should provide means to analyse impact on business processes. A threat may have different effects on a business process, depending on the process, dependent resources, line of business and organisation. The analysis should consider the perception, behaviour, and response of work force and customers. The business impact analysis should be able to model and analyse the impact on financial and nonfinancial KPIs.

- **Multi-Level Resource / Service Dependency and Risk Modeling**: The dependency models should support arbitrary kinds of layered resources (e.g. business process resources, ICT elements, human resources). Resource behaviour and temporal aspects of dependencies are important. The “ripple” effect of temporal failure propagation through resource layers is one subject
of analysis. Furthermore, it should be possible to model and analyse combined failure modes since threats may influence, foster or trigger one other.

- **Multi-Paradigm-Analysis and Optimisation**: Besides the Business Impact Analysis and the Resource and Dependency Analysis, the enhanced SOA modelling methodology should support the following analysis, planning and optimisation methodologies:
  
  o *Best, Average, Worst-Case Analysis*: Analysis of failure and recovery plan execution time: at best, on average and at most (worst-case). In ICT BCM, the worst-case execution time of a recovery plan is often of particular concern since it is important to know how much time at most is needed to recover business functionality.
  
  o *Dead-Lock*: It should possible to detect dead-lock or live-lock situations in recovery plans. - Single point of failure: Business Continuity Manager should be able to identify and spot single point of failure with regard to business continuity planning in an ICT landscape or recovery plan.
  
  o *What-If-Questions*: Business Continuity Manager should be able to simulate different scenarios and should be able to compare different recovery plans.
  
  o *Sensitivity Analysis*: Business Continuity Manager should be able to analyse variation and uncertainty in the dependency, threat and recovery plan models. The manager should be able to demonstrate that a recovery plan is robust and is valid even for uncertain risk assumptions.
  
  o *SLA Translation and Multi-Objective-Optimisation*: A methodology is needed to translate business level BCM metrics and KPIs to individual ICT elements BCM objectives, for example to translate RTO for a business process to the RTO objective of an external service. Furthermore ICT level BCM metrics need to be optimised; for example to identify the maximum tolerable outage time for given ICT elements with acceptable costs and minimised risks.

It has to be noted, that we do not aim to develop optimisation algorithms or simulation engines, rather it anticipates reusing existing implementations.

### 4.3 Conclusion and Future Work

In this section we proposed a methodology for enhanced SOA modelling and discussed requirements. Currently we are working on a reference architecture that will be used to evaluate our approach. A brief description is given here: The reference architecture centres on a formal Analysis and Planning Model. This central model provides means to model and express resources, dependencies, threats, effects and recovery plans. The model will be supplemented by a set of profiles to express business continuity manager preferences, the "risk appetite" of an organisations, legal constraints, etc. In order to reuse existing planning, analysis and optimisation tools, we utilise the TIPM [39] as an intermediate model. This means, that the Enhance SOA Model will be transformed using model driven technologies into a TIPM instance.

However, the current implementation of our reference architecture lacks support for user defined measurement models, which is needed for business continuity analysis. Therefore, a advanced version of BCM-TIPM has to be developed.

A future research topic might investigate on how BCM related feedback can be visualised in original landscape and business process modelling tools.
Furthermore we will focus on SLA Translation and Multi-Objective Optimisation of BCM-SLA objectives in collaboration with work package A5. The challenge is to traverse resource layers and interconnect behaviour and effect models, business impact with SLA objectives. Furthermore, business values may have to be considered and eventually expressed in monetary value to define penalties. Organisation preference must be taken into account to solve this multi-objective optimisation problem. Optimisation is also needed to detail of recovery plans (e.g. to define the maximal RTO).

5 Landscape Modelling

IT landscapes are typically characterized by a large number of heterogeneous software, hardware and service elements inter-related through a complex set of relationships and dependencies. In order to manage these complex landscapes effectively and efficiently to ensure smooth service operations and minimized service disruptions, these elements and their relationships and dependencies need to be captured and represented in the form of a coherent and consistent model. Through these coherent and consistent landscape models, IT operators and administrators can get a thorough comprehension of the state of IT landscapes for performing various activities during the lifecycle of services being offered by the organization. These models can be utilized for example: to understand the requirements and implications for on-boarding new services into the landscapes, to make better informed decisions during the operations and runtime phase of services, to comprehend the impacts of potential faults and failures in the landscapes, to perform intelligent decisions through effective root cause analysis etc.

In the context of this work package, the objective of the modelling exercise is to devise a meta-model which provides instruments to capture and represent software and service related artefacts (and their relationships and dependencies) of the IT landscapes. These models can then effectively facilitate the overall objective of this work package i.e. SLA Aware Service Management. The objective of this modelling is to facilitate various aspects of service management including service provisioning, monitoring, planning, prediction etc. The meta-model should therefore be able to express and capture information about software and service element attributes and their packaging, configuration, versioning, deployment related activities etc. In addition to the design of a metamodel, this task also aims at providing a default implementation of Software Landscape which realizes the metamodel and provides an entity which can be used to store and query the information captured through these landscape models.

In the context of SLA@SOI framework architecture, this metamodel and the respective SoftwareLandscape entity is part of the SoftwareServiceManager. Service providers can populate the SoftwareLandscape with information about the software and services being offered through the SoftwareServiceManager. This information is then queried by the SoftwareSLAManager & ServiceEvaluation entities during the service provisioning phase to understand the dependencies, requirements and features which need to be satisfied and configured during the provisioning process. Additionally, during the service runtime or adjustment phase, information about the respective actions should be reflected in this model to account for the current state of service deployed. This information push is also enabled through the SoftwareLandscape entity.

The following two sections present detailed description of the landscape metamodel and the implementation details of the SoftwareLandscape element.
5.1 Landscape Metamodel

As described previously, the landscape modelling exercise aims at capturing information about the software and service elements of the IT landscapes. A compact view of the software landscape metamodel is shown in Figure 3. Detailed description of the individual metamodel elements is given in the following sections along with their detailed structure and sub-elements.

![Figure 3 Landscape Metamodel [Compact View]](image)

5.1.1 Landscape

The central element of the landscape metamodel is the Landscape element. A landscape is a collection and aggregation of other model elements representing various services which are being offered by the service provider. There can be only one landscape element in a software landscape. Landscape element contains general attributes capturing information about service provider organization. A Landscape element is an aggregation second level model elements consisting of ServiceTopology, ServiceType, ServiceImplementation, ServiceBuilder and ServiceInstance. Descriptions of these elements are given in the following sections.

5.1.2 ServiceType

A ServiceType element specifies the external interfaces of a service. It basically describes what functionality the service provides. The Landscape element contains two types of ServiceType aggregations. Provided service types refer to the services a service provider can offer to its customers whereas required
service types refer to the services which provided service types are dependent on to deliver their functionality. Landscape element can potentially contain unlimited number of both required and provided service types. Figure 4 shows the ServiceType element attributes. The description of these attributes is explained below:

<table>
<thead>
<tr>
<th>ServiceType</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID : EString</td>
</tr>
<tr>
<td>ServiceTypeName : EString</td>
</tr>
<tr>
<td>Description : EString</td>
</tr>
<tr>
<td>Interfaces : EString</td>
</tr>
</tbody>
</table>

**Figure 4 ServiceType Element**

- **ID**: Unique identifier for this ServiceType
- **ServiceTypeName**: Textual name of this ServiceType
- **Description**: Brief description of the ServiceType elements
- **Interfaces**: set of interfaces that specify its exact functionality by a set of operations that are offered by this service

### 5.1.3 ServiceImplementation

ServiceImplementation (shown in Figure 5) realizes a specific ServiceType. They describe i) specific artefacts and assets (such as software components, or appliances) that are required to instantiate a service, ii) the dependencies of assets and artefacts on other services, and iii) the configurable features of the particular service implementation and its associated monitoring system. For example, the “All-In-One” implementation of the ORC’s services depends on one infrastructure service that hosts its appliance. The threadpool size of its Active BEPL engine can be adjusted according to the services usage and deployment. Furthermore, the monitoring system allows for instrumenting each service operation and extract response times and throughput.
To express such properties, ServiceImplementation contains a set of ImplementationArtefact, a set of ComponentMonitoringFeatures, and some ProvisioningInformation. ComponentMonitoringFeatures specify the capabilities of the monitoring system associated to a service. They contain information about the available sensor, effectors and reasoning components. Section 8.2 provides further details on the monitoring system’s metamodel.

ProvisioningInformation and ImplementationArtefact hold the information necessary to plan and execute the provisioning of a service. ProvisioningInformation contains the LeadTime that is needed to start a particular software system. Furthermore, it specifies the boot order in which multiple ImplementationArtefact are to be started. Each ImplementationArtefact represents a single unit such as appliances or software archives that have to be deployed separately. The metamodel of an ImplementationArtefact contains information about its dependencies and the ConfigurableServiceFeature associated with this artefact. Figure 6 presents a detailed view of the ImplementationArtefact and its elements.
A Dependency refers to ServiceTypes that are necessary to instantiate a given ServiceImplementation. Dependencies are part of ImplementationArtefacts to allow a direct association of a dependency to the artifact that requires it. This is necessary, if multiple artefacts require the same ServiceType with different quality characteristics. In this case, multiple SLAs need to be established for the same service type. Thus, a unique mapping between each SLA and the requiring ImplementationArtefact is necessary. A typical example for such a scenario is multiple dependencies to infrastructure services. If a ServiceImplementation comprises multiple appliances (e.g. one for the application server and one for the database), the model needs to reflect which appliance is to be hosted on which virtual machine.

Furthermore, ConfigurableServiceFeatures describe properties of a service that can be adjusted for each instance. Again, ConfigurableServiceFeatures are directly associated to ImplementationArtefacts in order to avoid ambiguities. ConfigurableServiceFeatures comprise a unique identifier (ID), a configuration type (e.g. property file or environment variable), a pointer to a file (ConfigFile), and an identifier of the parameter to be adjusted (ParameterIdentifier). Identifiers depend on the configuration and file type considered. For example, the identifier of a parameter in an XML file can be an XPath expression.

ImplementationArtefacts are abstract entities that have to be specialized for different domains. In Figure 6, we show subclasses for ResourceArtefacts, DeploymentArtefacts, and DataArtefacts. DeploymentArtefacts are further refined to VirtualAppliances and SoftwareArchives. These elements contain detailed information necessary to deploy the artefact. Please note that ImplementationArtefacts are only used inside a ServiceManager. Thus, the information about the internal structure of a service implementation does not have to be understood by SLAManagers and ServiceEvaluation. For this purpose ServiceImplementations contain explicit operations that aggregate the dependencies and ConfigurableServiceFeatures for external processing.

ResourceArtefact element represents the supplementary resource required by the services during their provisioning, initialization or runtime. DataArtefact element can be used to provide information about the data elements which need to be imported into databased or directory services for example which serves as the
seed data to be used by the services during their initialization or operations. Deployment Artefact elements represent the software or service elements which need to deployed and configured during the service provisioning phase. Deployment Artefact elements are further specialized into Virtual Appliance and Software Archive elements which serve as the concrete model elements used for specification of the deployment information.

Virtual Appliance elements refer to the virtual machine image packages which should be deployed onto hypervisor elements during the service provisioning process. Virtual Appliance element specification includes attributes like hypervisor that should be used for deploying the image and its hypervisor version. Additional attributes include the image format used in the virtual appliance package, the operating system and its version. Software Archive refers to the elements which should be deployed onto application containers for service provisioning. Examples of such archives include WAR, EAR or JAR files which should be deployed into web services or J2EE containers for deploying and provisioning the services.

5.1.4 ServiceBuilder

Service Builders serve as a communication datastructure to be used by SLA Managers, (Software-) Service Managers and Service Evaluation. Service Builders are associated with a Service Implementation for which they construct a new service instance. They are used throughout the whole negotiation and provisioning process and are stepwise enriched with information. Basically, Service Builders serve as configuration objects for new service instances. SLA Managers, Service Evaluation, and Service Managers exchange information on a potential service using the Service Builder. The implementation of the Service Builder follows the Builder pattern of Gamma et al. [41]. For a single Service Implementation, multiple Service Builders can exist that are associated with it.

A Service Builder (shown in Figure 7) is responsible i) for resolving dependencies of a Service Implementation by offers of an external service provider or another SLA Manager, ii) for the configuration of specific service features, and iii) for the configuration of specific monitoring features. For each dependency of a Service Implementation, the Service Builder can hold a Service Binding that maps the dependency to an SLATemplate or one of its specialisations (SLA and Business Product). The SLATemplate represents a contract for a service of the required type. It contains all information necessary to assess and access a service outside of the current SLA Manager's domain. It includes quality constraints and, after the SLA has been agreed, endpoints of the service.

For configuration purposes of a service, Configuration Directives assign new values to Configurable Service Features. The setup of the monitoring system is given in the Monitoring System Configuration. Details about monitoring features and configurations follow in Section 8.2.
A **ServiceInstance** (shown in Figure 8) refers to the instantiated version of a ServiceImplementation. As such, it contains information about the runtime aspects of the deployed services. For example, it contains the Endpoints of a particular service. The Endpoints either refer to a running and accessible service or point to the locations where the service will be available according to the time constraints defined in the corresponding SLA. Each ServiceInstance can contain multiple endpoints. Additionally, a ServiceInstance contains information about the date and time of its creation (InstantiatedOn) and the usual means for identification by machines (ID) and humans (ServiceInstanceName, Description).

## 5.2 Software Landscape

Landscape metamodel is realized in the Software Landscape component of the SoftwareServiceManager. In essence SoftwareLandscape represent a repository of landscape models. SoftwareLandscape provides an interface to store, query and
manipulate the landscape models. Eclipse Modelling Framework (EMF) is used to implement the landscape metamodel in the software landscape.

This functionality is exposed through the SoftwareLandscapeManager entity. SoftwareLandscapeManager provides a CRUD like interface which can be used to manipulate the landscape models. Figure 9 shows the interface for SoftwareLandscapeManager. Detailed interaction semantics and operations are described in the following paragraphs with respect to the create, read, update and delete aspects of the interface.

<table>
<thead>
<tr>
<th>SoftwareLandscapeManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getModelFactory(): SoftwareLandscapeFactory</td>
</tr>
<tr>
<td>+ addToLandscape(obj: EObject) : void</td>
</tr>
<tr>
<td>+ queryAllServiceTypes(): List&lt;ServiceType&gt;</td>
</tr>
<tr>
<td>+ queryServiceTypeById(typeId : String): ServiceType</td>
</tr>
<tr>
<td>+ queryServiceTypeByName(name : String): ServiceType</td>
</tr>
<tr>
<td>+ queryAllServiceImplementations(): List&lt;ServiceImplementation&gt;</td>
</tr>
<tr>
<td>+ queryServiceImplementationByImplId(implId : String): ServiceImplementation</td>
</tr>
<tr>
<td>+ queryServiceImplementationByType(type : StringType): List&lt;ServiceImplementation&gt;</td>
</tr>
<tr>
<td>+ queryServiceImplementationByTypeAndName(typeName : String): List&lt;ServiceImplementation&gt;</td>
</tr>
<tr>
<td>+ queryAllServiceInstances(): List&lt;ServiceInstance&gt;</td>
</tr>
<tr>
<td>+ queryServiceInstanceById(nsId : String): ServiceInstance</td>
</tr>
<tr>
<td>+ queryServiceInstanceByName(nsName : String): ServiceInstance</td>
</tr>
<tr>
<td>+ queryServiceInstancesByImplId(implId : String): List&lt;ServiceInstance&gt;</td>
</tr>
<tr>
<td>+ queryServiceInstancesByImplName(implName : String): List&lt;ServiceInstance&gt;</td>
</tr>
<tr>
<td>+ updateLandscape(): void</td>
</tr>
<tr>
<td>+ deleteFromLandscape(obj: EObject): boolean</td>
</tr>
</tbody>
</table>

**Figure 9 SoftwareLandscapeManager Interface**

### 5.2.1 Create

In order to create new model element in the software landscape, SoftwareLandscapeManager can be used in the following sequence:

- All the model elements can be created through a model factory, `getModelFactory()` method can be invoked to initialize and get an instance of the factory. This factory provides methods for creating all the model elements.
- Once model elements are created, these can be populated with relevant information through the getter/setter methods of individual model elements. These model elements can then be added to the software landscape by invoking `addToLandscape()` method passing the model element as parameter. As mentioned earlier, EMF is used to design the landscape metamodel, therefore, `addToLandscape()` takes the generic `EObject` which is a superclass of all the model elements described in the previous sections.

### 5.2.2 Read

To read model elements `SoftwareLandscapeManager` provides a variety of methods. These methods are described below:

- To query service types, `queryAllServiceTypes()`, `queryServiceTypeById()`, `queryServiceTypeByName()` methods can be used
To query service implementations, queryAllServiceImplementations(), queryServiceImplementationById(), queryServiceImplementationByName(). Moreover, to search for service implementations by passing service type information, queryServiceImplementationByTypeId(), queryServiceImplementationByTypeName(), queryServiceImplementations()

- To search and retrieve service instances, queryAllServiceInstances(), queryServiceInstanceById(), queryServiceInstanceByName(), queryServiceInstanceByImplId(), queryServiceInstanceByImplName()

5.2.3 Update

To update software landscape elements, query functionality can be performed and required changes can be performed on the returned objects. Once all the changes have been performed, updateLandscape() methods can be invoked to make these changes permanent.

5.2.4 Delete

To delete elements from software landscape, respective model elements should be queries using the query methods and then deleteFromLandscape() method can be invoked passing the object to be deleted. This will remove the model element from software landscape.

5.3 Conclusion and Future Work

This section presented the design of landscape metamodel and implementation details of the Software Landscape element which serves as the repository of landscape models. Landscape metamodel described is capable to capturing and representing information about software and service elements of the IT landscapes. The information captured includes software, services, their inter-relationships, packaging and monitoring features. This information is utilized by the SLA managers during SLA planning and service provisioning phases of the service lifecycle. Software Landscape serves as the landscape model repository and is part of the SoftwareServiceManager component of the SLA@SOI framework. In the future, focus should be on devising a methodology to align the software landscape with the infrastructure landscape to provide a consolidated and coherent picture of the entire IT landscape. Moreover, alignment will be performed with the enhanced SOA models to support business continuity management processes.
6 Dynamic Service Discovery and Composition

This document describes the orchestration engine realized to implement the general architecture defined at the end of the first year (see [1]). Such an architecture aims to offer an efficient mechanism to make dynamic binding (i.e. to bind steps of a process at run time) and to provide an execution environment for abstract processes described with a new methodology focused on a better separation between process description and (abstract) component services description (see innovation section).

6.1 Dynamic Orchestration Engine (DOE)

6.1.1 Overview

Our general objective has been to be compliant as more as possible with actual standard for composite services description. For this reason we have assumed to use WS-BPEL language [2] (the standard de facto) as description language and to adopt one of the more diffused open source WS-BPEL engine as starting point of our work. The chosen engine is ActiveBPEL [3] community edition since it has resulted to support in the best way the WS-BPEL specification [4].

WS-BPEL defines the relationships with external web services by means of required and exposed interfaces, assuming that such interfaces are defined by means of documents compliant to the WSDL 1.1 specification. In detail the definition of a WS-BPEL business process follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and port type definition versus binding and address information). A WS-BPEL process represents all partners and interactions with these partners in terms of abstract WSDL interfaces (port types and operations); no references are made to the actual services used by a process instance, in other words WS-BPEL does not make any assumptions about the WSDL binding.

In detail any interaction is associated to a partner link: “The notion of <partnerLinks> is used to directly model peer-to-peer conversational partner relationships. <partnerLinks> define the shape of a relationship with a partner by defining the portTypes used in the interactions in both directions” [2]. The attribute myRole of a partnerLink indicates the role in the interaction of the process executor, while the attribute partnerRole indicates the role of the partner (see Figure 10). Any partner role has an associated interface (port type).

To invoke an operation on the port type of a partner role, a binding information must be available at invocation time. The binding information is represented by an EPR (End Point Reference) linked to the specific partner role and can be provided as part of the business process deployment, but the BPEL specification does not make any assumption about this static binding mechanism. Moreover the specification does not make any assumption about the format of the EPR.

Starting from these considerations we propose a management interface, allowing to associate several binding information to a specific partner role of a WS-BPEL process, in any moment after its deployment (i.e. dynamically). This interface allows to associate to a partner role a constant binding or a conditional binding dependent from the status of the process.
More precisely, it is possible to associate to each partner a set of couples <condition, binding information>. Each couple represents a binding rule. The condition represents the process status that triggers the specific binding. We describe such a condition by means of a set of couples <process variable, value>.

This approach results of interest for efficiency reasons (see innovation section) and also because it is independent from the deployment procedure/descriptors, of the adopted BPEL engine but relies just on standard BPEL concepts. So this solution may be applied also to other engines, possibly using the same interface.

Finally we propose to use the same interface also to complete the description of an abstract processes (see [1]), by extending the concept of binding rule with that of abstract binding rule. An abstract binding rule associates a SLA template document to a specific condition in place of a binding information. This template describe a service that will be searched, negotiated and bound anytime the node must be executed.
6.1.2 Implementation

To support the functionalities discussed above and to offer an appropriate runtime environment we implemented the architecture described below.

ARCHITECTURE

Figure 12: DOE Architecture

The components depicted in grey colour represent new components, while the white ones represent components of ActiveBPEL. Furthermore, the figure distinguishes, by means of different colours, interactions triggered at execution time (green), triggered during the engine initialization (red) or needed to manage the binding configurations of a deployed process partner roles (e.g. to add new binding rules). Such interactions will be better explained in the next sections.

FUNCTIONALITIES EXPLANATION

The main functionalities, of the management interface are the following:

i) Add a (abstract) binding rule to a partner role

ii) Remove an (abstract) binding rule associated to a partner role

iii) Retrieve all (abstract) binding rules of a partner role

ActiveBPEL already offers an interface (defined by a wsdl) for management actions (e.g. to deploy a process, to terminate a process etc.) implemented by an administration web service (AeActiveBpelAdminImpl: see Figure 12). We extended such a web service adding a new interface to it (specified by the WSDL in Appendix B: Admin Interface) represented in the figure below.
Registration of a new binding rule is effected through the administration method:

- `addPartnerRoleBindingRule(processID: String, partnerRole: String, bindingRule: EnsBindingRule): void`

The EnsBindingRule class is automatically created\(^1\) by de-serializing the corresponding xml schema type defined inside the SLA@SOIEngineAdmin.wsdl\(^2\). Upon invocation of `addPartnerRoleBindingRule(...)` the AeActiveBpelAdminImpl asks the AesTypeConversionHelper class for some transformation of input parameters and delegates the invocation to the AeRemoteDebugImpl class which is the remote implementation of the BPEL engine used for administration web service invocations. The mechanism just described is an interaction pattern, adopted by ActiveBPEL, for its standard administration methods, to distinguish the internal parameters representation (i.e. used inside an AeBusinessProcessEngine) from the external one (i.e. used by the administration web service). In our case an EnsBindingRule is transformed into the internal type EnBindingRule, implementing the IEnBindingRule interface, with, basically, the same information content.

Finally the AeRemoteDebugImpl retrieves from the running (singleton) engine (i.e. the AeBusinessProcessEngine) the running (singleton) Binder and delegates to it the adding of the binding rule. If a rule with the same condition is already

---
\(^1\) By axis 1.4 which is the SOAP implementation adopted by ActiveBPEL to hand its web services.

\(^2\) See Appendix C: Admin Interface
associated to the specified partner role of that process it will be overridden with the new one.

![Figure 15: addPartnerRoleBindingRule – sequence (2/2)](image)

We may note that Binder lifecycle is strictly related to the lifecycle of the engine (AeBusinessProcessEngine); it is initialized/destroyed when the engine starts/ends and is made available to the other components by the engine itself. The Binder holds the association between partner roles and binding rules (see Figure 16). The persistence of the binder data is realized by means of JPA\(^3\) (Java Persistence API).

![Figure 16: Class diagram: Binder internal relationships](image)

Registration of a new abstract binding rule is affected through the administration method:

- `addPartnerRoleAbstractBindingRule(processID:String, partnerRole:String, abstractBindingRule:EnsAbstractBindingRule):void`

Basically the interactions involved are the same already described above. In this case, anyway, we need to convert\(^4\) the SLA template contained in the EnsAbstractBindingRule to a different java format before building the internal

---

\(^3\) Toplink implementation

\(^4\) AesTypeConversionHelper implements conversion
type EnBindingRule. The reason is that Axis automatically adopts the Java Bean convention for object de-serialization while SLA@SOI framework adopts a specific Java format. Since, as it will be seen in the next section, the DOE has to send the provided SLA template to other framework components it was necessary to realize such conversion. We implemented this by a specific de-serializer for Axis\textsuperscript{5} able to produce an Xml Bean compliant Java representation for the de-serialized objects and then we uses a parser \cite{5} (5.3 section) included in the SLA@SOI framework, capable to convert a SLA template represented by an Xml Bean into the SLA@SOI specific format (org.slasoi.slamodel.sla.SLATemplate class). Finally, we used a local instance of the SLA@SOI SLA Template Registry \cite{5} to hold templates of abstract rules (see Figure 16).

The basic retrieval methods exposed by the DOE are as follows:

- \texttt{getPartnerRoleBindingRules(processID: String, partnerRole: String): EnsBindingRulesListType}
- \texttt{getPartnerRoleAbstractBindingRules(processID: String, partnerRole: String): EnsAbstractBindingRulesListType}

The interaction flow is analogous to the one already showed for adds methods: if the specified partner role does not exist an exception will be thrown. Details on output types EnsBindingRulesListType and EnsAbstractBindingRulesListType can be found in the administration interface schema (Appendix B: Admin Interface).

The removal of a binding rule is effected by the method:

- \texttt{removePartnerRoleRule(processID: String, partnerRole: String, condition: Map): void}

An exception will be thrown if the specified partner role does not exist or if no rule, with the specified condition, exists.

\section*{RUNTIME ENVIRONMENT}

As said above the DOE must support the execution of processes whose (abstract) binding rules have been specified using the administration interface just described. Since such information is managed by the Binder we intercept the execution of invocation steps and re-direct them to the Binder, that performs the actual service invocation\textsuperscript{6} and returns the control to the engine (see also Figure 12 green wires).

When an invoke activity is executed from within a BPEL process instance, ActiveBPEL uses its internal Web service framework to package the message into a SOAP request and send it over the HTTP protocol to the designated service endpoint address. Instead of using this standard invocation mechanism ActiveBPEL offers the possibility to implement a custom invoke handler\textsuperscript{7} to bypass the creation and transport of SOAP messages over HTTP. The only constraint, for a custom invoke handler, is to implement the IAEInvokeHandler interface exposing the method handleInvoke(AeInvokeActivity). We have adopted such

\textsuperscript{5} Implemented classes: EngXmlBeanSerializer, EngXmlBeanDeserializer.

\textsuperscript{6} The Binder has to solve, eventually, syntactically interface mismatching: such functionality is not implemented in Y2.

\textsuperscript{7} It must be communicated to ActiveBPEL server that it should use the custom invoke handler when an invoke activity with a specified partner role is executed. This is accomplished by modifying a deployment artefact (the Process Deployment Descriptor PDD: see Appendix C: PDD example)
mechanism implementing our custom invoke handler (EngInvokeHandler) to address invoke execution.

Basically the EngInvokeHandler has to retrieve, from the invoke activity, three kinds of information, that the Binder needs to make the invocation:

INFO1
- endpoint specified at deployment time
INFO2
- process and partner role identifiers
- process status (i.e. specific process variables)
INFO3
- request message parts (xml documents)

Such information are used following the logic described in the figure below:

![Figure 17: Run time: interactions (1/2)](image)

INFO2 data are exploited by the Binder to get the partner role binding rule whose condition matches with the process status; if a matching concrete binding rule exists then its binding information will be used to perform the invocation, INFO1 data will be exploited otherwise. In both cases INFO3 data are mandatory since they represent the input message information.

The Binder manages only concrete bindings, while abstract rules are managed by a pluggable component that is triggered before the control is passed to the
Binder. This is realized by the instrumentation layer [1], that intercepts the execution request of the invoke activity, checks if the invoke activity under execution is abstract and, in the case, alerts the plugged component. The plugged management component is in charge to specify a concrete binding rule for the partner role of the activity (see Figure 18). In such way when the execution continues there is the guarantee that at least one concrete binding rule has been associated to the abstract invoke.

![Diagram](image)

**Figure 18: Run time: interactions (2/2)**

In detail the instrumentation, after verified the invoke activity is abstract and that no concrete binding rule was already associated to its partner role, retrieves the matching abstract binding rule (one at least must exist otherwise an exception is thrown) and builds a *Binding Missing Event* (BME) that is passed to the plugged management component.

![Diagram](image)

**Figure 19: BME conceptual model and serialized instance example**

8 A concrete binding rule can be add in any moment after process was deployed.
The described mechanism works independently from the SLA@SOI framework. When the DOE is used in conjunction with the SLA@SOI framework, the BME will be published, in a serialized format (see Figure 19), on the monitoring event channel and it will be received by a PAC (Planning and Adjustment Component), specific management component for SLA@SOI, subscribed to BME events. The PAC component will analyze the SLAT contained in the BME and will find out a matching SLA. The PAC is domain dependent, so different strategies can be adopted to get such SLA. For instance it can use a central SLAT Registry to find a list of matching SLATs, to ordinate such list and finally select and negotiate the best one. In other cases, instead, negotiated SLAs could be already available so that only a selection will be necessary. Anyway the PAC, finally, will extract from the SLA the necessary binding information and will build a concrete binding rule to set by means of DOE administration interface (see Figure 13). We aim to realize a reference implementation of a PAC exploitable from B line for specific specializations.

6.2 Future Work

Some limitations of the current work will form the basis for evolving the work and implementation in Year 3. In detail we aim to extend the mechanism for the automatic dynamic binding SLAT based (i.e. the mechanism for execution of abstract invoke) described above.

Such mechanism, in fact, allows to automatically bind an abstract invoke partner role with one (and only one) specific web service, coherently with the WS-BPEL specification which refers only to a bi-party conversational model [2]. We aims to overcome this limit by introducing the concept of an Abstract Multiple Invoke Activity, that implements a multi-party conversational model, allowing to invoke in one step several partners exposing the same operation. To deploy a process with such an invoke activity, a new type of abstract rule (let’s say ExtendedAbstractBindingRule) must be associated to its partner link. In this way when the PAC receives the BME with such a rule, it will use the specified SLAT to retrieve a list of matching SLAs and will specify the whole list of the binding information, coming from these SLAs, by means of the DOE administration interface. Finally the Binder will recognize that the invoke activity has been bound and, for each endpoint in the list, it will perform the service invocation, gathering the results to return as reply to the invoke under execution.

As evident this improvement will require to work on both the DOE interface and components involved at run time. Other than a technological challenge, this is also an improvement needed to fulfil the requirements coming from B line (see B6 use case).

---

9 SCA (Service Component Architecture) is a technology which has proposed an extension of WS-BPEL to overcome such limit [6] offering a mechanism to implement “multi reference” partner roles.

10 A new method [e.g. addRuleForMultipleBinding(String processID, String partnerRole, AesMultipleBindingRule bindingRule)] must be implemented.
7 Unified SOA Manageability Interface

Herein we present our work on the design of a generic unified manageability interface for managing services through appropriately deployed sensors and effectors. The generic manageability interface extends the work achieved in the project’s first year, and aligns it with the design of the overall SLA@SOI framework.

The unified manageability interface enables the dynamic management of the lifecycle of the service: from initial deployment and startup, to advanced runtime activities such as monitoring and adjustment.

In the SLA@SOI project, services are not required to directly implement the unified manageability interface, but a generic manageability agent is provided. A manageability agent is a generic component that supplies a gateway to managing one or more services, hiding the domain-specific details of the management itself.

7.1 The Unified Manageability Interface

The SLA@SOI manageability agent is a component that implements the IManageabilityAgent interface. This interface encapsulates the basic functionality for starting and stopping service instances. Furthermore, it provides access to, or allows to create, domain-specific management hooks for our services. These hooks are implementations of the IManageabilityFacade interface (see Figure 20).


Figure 20: The Unified Manageability Interface

7.1.1 IManageabilityAgent

The IManageabilityAgent interface contains the following methods:

```java
void startServiceInstance(ServiceBuilder builder, Settings connectionSettings,
```

...
This method starts a new service instance according to the configuration given in the builder object. Concrete details regarding the ServiceBuilder object can be found in Deliverable D.A1a "Framework architecture (full lifecycle)" Section 4.3.5. Once the service is available a notification event is sent over an XMPP event bus using the settings in connectionSettings, and the channel given in notificationChannel. The method throws an error if a problem arises during the initialization of the provisioning process (ServiceStartupException) or if the XMPP bus cannot be reached.

```java
void stopServiceInstance(ServiceBuilder builder)
```

This method stops the service that was deployed and started using the builder object, making it unavailable.

```java
List<Endpoint> getEndpoints(ServiceBuilder builder)
```

This method uses the builder object to determine the endpoints of the created service instance.

```java
IManageabilityAgentFacade getManagibilityAgent(ServiceBuilder builder)
```

This method is used to gain access to a domain-specific implementation of the IManageabilityAgentFacade interface for managing a specific service instance. The builder object that was used to deploy and start the service specifies the service instance that needs to be managed.

### 7.1.2 IManageabilityAgentFacade

Components that implement the IManageabilityAgentFacade interface provide the domain-specific knowledge required to perform advanced management on a given service instance. Nevertheless, the interface maintains a generic level. It provides methods for configuring the service-specific sensors that will produce the data needed for monitoring the service. It also presents a generic “executeAction” method for performing adjustment activities on the service. The interface provides the following methods:
ServiceInstance getInstance()

This method returns the ServiceInstance that is controlled by the specific manageability agent façade. The ServiceInstance is a part of the Service Construction Model, which can be found at Deliverable D.A1a “Framework architecture (full lifecycle)” Section 4.3. The ServiceInstance contains information regarding an instance of a service implementation. In practice, it contains the endpoints of a particular service, the date and time of creation, and the usual means to identify the service both in a machine- and human-readable fashion.

List<SensorSubscriptionData> getSensorSubscriptionData()

This method returns sensor subscription data for all the sensors that are at the moment active on the service instance being managed. In particular, each SensorSubscriptionData object provides all the information needed to subscribe on the XMPP event bus to the events that are produced at run time by a given sensor.

void configureMonitoringSystem(IMonitoringSystemConfiguration config)

This method applies a configuration to the monitoring system the service instance. If a previous configuration is already in place it is replaced entirely. The monitoring configuration is a class that contains two kinds of information: sensor configurations (given through class SensorConfiguration), and configurations for reasoning components (given through class ReasonerConfiguration). The effect of the method is to configure the two parties so that they can communicate at run time. The sensors will collect the data needed for performing advanced monitoring analyses, and send them to the reasoners.

void deconfigureMonitoring()

This method removes any monitoring configuration that may be active on the service instance.

IEffectorResult executeAction(IEffectorAction action)

This method executes domain-specific adjustment actions on the service instance. It takes an IEffectorAction containing both which domain-specific adjustment action that has to be performed, and the parameters that need to be passed to
this action. It returns an IEffectResult indicating if the action was successful, as well as any domain-specific payloads that there may be.

### 7.2 Integration with the SLA@SOI Framework

The Manageability Agent bridges the generic SLA@SOI framework with the domain-specific implementation of a service instance (see Figure 21). The SLA@SOI framework interacts with the Manageability Agent through a Service Manager.

![Figure 21: The Manageability Agent in the context of the SLA@SOI framework.](image)

The Service Manager calls the Manageability Agent through the native_service_management interaction. This interaction is implemented through the generic IManageabilityAgent and IManageabilityAgentFacade interfaces we introduced in the previous section.

During the deployment and setup phase the Manageability Agent receives, through a domain-specific façade implementation, a configuration request from the Service Manager. The request is transformed into domain-specific sensor and effector configurations and transferred to the service instance through interactions configure_sensor and configure_adjustment.

At run time the data that are collected are sent to the event bus. Any components that have subscribed to the data will receive them. Interested components can discover how to subscribe using the appropriate method provided by the façade. Should the SLA@SOI framework decide that some sort of adjustment is required the Service Manager will send an appropriate executeAction request through interaction native_service_management. This request is transformed into a domain-specific request that is forwarded to the specific service instance.

### 7.3 Managing the DOE

We provide an implementation of the Manageability Agent that provides the SLA@SOI framework with DOE-specific implementations of the
IManageabilityAgentFacade interface. This allows the system to deploy and manage the process instances inside the DOE.

The DOE currently provides a domain-specific SOAP-based administration interface. The interface provides remote methods for deploying new processes, for configuring sensors through instrumentation directives, and for configuring dynamic binding rules for an executing process.

Figure 22 illustrates how the Manageability Agent is used by the Service Manager to deploy a new process into the DOE, and then translate generic monitoring and adjustment configurations into DOE-specific ones.

Figure 22: Managing the DOE

The Service Manager starts by calling method startServiceInstance on the Manageability Agent, and passing it a builder object. The manageability agent reads the builder object and discovers that the service it has to start is a BPEL process. To deploy and manage the new service it creates a new DOEFacade object, i.e., a DOE-specific implementation of the IManageabilityAgentFacade interface. At this point, there are a few optional method calls that may need to be performed on the DOE. First of all, if the required process is not already deployed in the DOE, the façade needs to perform the deployment itself. Deployment is performed calling the deployBPR method present in the DOE’s administration interface. The method takes, as its only parameter, an archive of the process (bpr file) to be deployed. The bpr file is found as an artefact inside the builder object. Second, if the builder contains concrete or abstract dynamic binding rules, they are added to the DOE calling methods addPartnerRoleBindingRule and addPartnerRoleAbstractBindingRule respectively. The parameters they need are contained within the builder object, and consist in the process’ id (i.e., the name used during its initial deployment), the partner role for which we are providing a binding rule, and the concrete or abstract binding rule itself.

Further run-time interactions between the Service Manager and the Manageability Agent are performed using a DOE-specific façade, which is obtained calling method getManageabilityAgentFacade on the Manageability Agent. In Figure 22 the ServiceManager issues a new monitoring configuration by calling method configureMonitoringSystem on the façade. The MonitoringSystemConfiguration
object that is received only contains sensor and reasoning component configurations. There is no explicit effector configuration, since changing the concrete and/or abstract binding rules associated with a given partnerRole is seen as a run-time adjustment action, and as such it needs to be requested through method executeAction on the façade. The request for a monitoring configuration produces calls to the DOE for configuring sensors.

The sensors are configured calling method configureSensor, which requires as input the process’ id (i.e., the name used during initial deployment), an operation id (i.e., a unique identifier for the BPEL activity to instrument), a correlationKey and a correlationValue, and a list of target eprs. The correlation key and value are used to ensure support for multi-tenancy within the DOE. In practice, the instrumentation is only executed on those process instances that are executed by the client with which the original SLA was negotiated. The correlation key is the name of an XML element that can be found within the message that initiates the process execution, while the values is what we expect to fine therein at run time. The list of target eprs is a list of event buses to which the collected data needs to be sent to during execution.

At run time it may be the case that adjustment is required by the Service Manager. In this case the Service Manager must call method executeAction on the façade. This request contains a reference to the kind of action that needs to be performed. The DOE currently only supports dynamic binding so it gets translated into an addPartnerRoleBindingRule or an addPartnerRoleAbstractBindingRule call on the DOE’s administration interface. The needed parameters are in the builder object, and they contain the process’ id (i.e., the process name used during the initial deployment), the partnerRole, and the new binding rule.

### 7.4 Conclusions and Future Work

We have designed a generic and unified interface for managing services that bridges the gap between the domain-agnostic SLA@SOI framework and the technicalities of the specific service instances we want to manage. We have also provided a concrete implementation of the proposed APIs for the Dynamic Orchestration Engine.

In the project’s next year we will continue to expand and enrich the management interface, and its generic data types, to provide simpler, and at the same time richer, QoS-driven monitoring and control of service instances. In particular, we will extend our interfaces to align them to, and fully support, the concepts defined in the design-time modelling for management work being achieved in the project’s A6.6 task: “Manageability modeling and generation for (composite) business services”.

### 8 Advanced SOA Management

#### 8.1 Monitoring System Overview

The purpose of the Monitoring System (illustrated in Figure 23) for SOA Management is to compliment the ServiceManager component by providing the means to monitor provisioned service instances in the context of requested Agreement Terms (contained in an SLA).
Figure 23  Overview of Monitoring System in SLA@SOI

The **Planning and Optimization Component** (POC) is the local executive controller for a **ServiceManager**. It is responsible for assessing and customizing SLA offers, evaluating available service implementations and planning optimal service provisioning and monitoring strategies. The POC generates a suitable execution plan for monitoring (based upon a configuration obtained from the **MonitoringManager** component) and passes this to the **Provisioning and Adjustment Component** (PAC).

The **Provisioning and Adjustment Component** (PAC) collects information from the Low Level Monitoring System, analyzes the incoming events and decides if a problem has occurred or it is about to occur, identifies the root cause and if possible decides and triggers the best corrective or proactive action. In case the problem cannot be solved at a local level, PAC escalates the issue to a higher level component, namely the POC. In case of an SLA violation, Adjustment can trigger re-planning, re-configuration and/or alerting to higher-level SLA. These
capabilities are considered to be important in order to guarantee best user perception preserving underlining resources.

The **MonitoringManager** (MM) coordinates the generation of a monitoring configuration of the system. It decides, for an SLA model instance it receives, which is the most appropriate monitoring configuration according to configurable selection criteria. A monitoring configuration describes which components to configure and how their configurations can be used to obtain results of Guarantee Term monitoring.

The **Low Level Monitoring Manager** is a central entity for storing and processing monitoring data. It collects raw observations, processes them, computes derived metrics, evaluates the rules, stores the history and offers all this data to other components (accessible through the ServiceManager). It implements the monitoring part of a *ProvisioningRequest*, containing constraint based rules (time and data driven evaluations) and *ServiceInstance* specific Sensor related configurations.

There are three types of Monitoring Features in the Advanced Monitoring System. First, **Sensors** collect information from a service instance. Their designs and implementations are very much domain-specific. It can be injected into the service instance, e.g., service instrumentation, or it can be outside the service instance intercepting service operation invocations. A sensor can send the collected information, to the communication infrastructure or other components can request (query) information from it. There can be many kinds of sensors, depending on the kind of information they want to collect, but all of them should implement a common interface. The interface provides methods for starting, stopping, and configuring a sensor.

Second, **Effectors** are components for configuring service instance behaviour. Their designs and implementations are very much domain-specific. An effector can be injected into a service instance, e.g., service instrumentation, or can interface a service configuration interface. There can be many kinds of effectors, depending on the service instance to be controlled, but all of them should implement a common interface. The interface should provide methods for configuring a service.

The third type of monitoring feature is a **Reasoning Component Gateway (RCG)**. An RCG provides the interface for accessing a Reasoning Engine. A reasoning engine (or short name as Reasoner) performs a computation based upon a series of inputs provided by the events or messages sent from a sensor and/or effector. An example RCG may be `completion_time` which accepts events from sensors detecting both request and responses to a service operation.

### 8.2 Monitoring Manager

The Monitoring Manager (MM) coordinates the generation of the monitoring system configuration for use by the **Planning and Optimisation Component (POC)**. It selects, for a given set of Guarantee Terms of an SLA, which are the most appropriate monitoring configurations according to a configurable selection criteria. A monitoring configuration specifies which components are to be configured and the interdependency of monitoring results between monitoring components. Monitoring Components (MC) receiving monitoring configurations are generalised as one of three types being; **Sensors**, **Effectors** and **Reasoning Component Gateways** as described in section 8.1. Each type of component has a series of monitoring features which describe their capabilities (e.g. provides the average `completion_time` of service requests or provides the result of comparing two values if one is less than the other). The MM is used only by the POC (as illustrated in Figure 24).
Given an SLA model and a set of monitoring features, the POC requests the MM to "check for monitorability". The goal of the monitoring manager is to take the SLA model, extract the Agreement and Guarantee States and match each Guarantee State expression with the monitoring components through their advertised features. The core process of the Monitoring Manager is described in more detail in the following section.

### 8.2.1 Core Process

A high-level flow is described as follows and illustrated in Figure 25. The process takes as input a SLA Model and a series of Monitoring Features. Firstly, the MM extracts the Guarantee States from Agreement Terms of the SLA model. The terms are in turn parsed into an efficient and formal Abstract Syntax Tree (AST) for the expressions of the states.

The AST is then used as input to select each Guarantee State expression (by traversal of the AST) of each state and match each left-hand side (lhs), operator and right-hand side (rhs) of the expression with appropriate monitoring features. The matching algorithms are discussed in section 8.2.3. Following selection, the delegate components form a **SelectedComponents** list, which in turn, are used to generate a complete **Monitoring System Configuration (MCS)** result for an agreement. If no suitable monitoring configuration can be formed (i.e. not all monitoring requirements could be matched) then an empty configuration is returned for a particular guarantee term.

The result of the Monitoring Manager process is an MCS representing the configuration of selected Monitoring Components which reason or provide events to monitor the Agreement Terms of the SLA passed. For input and output formats please refer to section 8.2.4.
8.2.2 Parsing Agreements and Guarantees

The Monitoring Manager abstracts the Guaranteed States (a guarantee made by one of the parties involved in the agreement) that a certain state of affairs will hold for the service. We abstract these states from the Agreement terms and parse the terms using a grammar which is based upon the Backus Normal Form (BNF) specification of the SLA model work package A1 deliverable D.A1a. The grammar for the parser is currently only based upon the Agreement Term and Guaranteed State expressions. A sample part of the grammar is listed in Figure 26.
The grammar is used as input to the Java Compiler Compiler (JAVACC) [7] which generates compiler source code to accept and parse source files specified in a defined grammar language. The generated compiler can be reused by Monitorability Agents who translate the SLA terms in to their own language specification (e.g. from SLA@SOI model expressions to the EVEREST Event calculus). For an example of this the reader may refer to the work package A5 deliverable D.A5a.

8.2.3 Monitor Component Selection

The principle for monitoring component selection is loosely based upon the initial results as detailed in [8][9][10]. The Year 2 Monitoring Manager now consists of two core algorithms for monitor selection. Firstly, the MonitorSelection algorithm (Figure 27) takes as input the TermAST generated from parsed Agreement Terms (section 8.2.2) and a set of candidate Monitoring Component Features. The algorithm extracts the lhs, operator and rhs expressions of each term starting from a root Guaranteed Term (Guaranteed State).

The MonitorSelection algorithm in turn, uses a SelectMonitor algorithm (Figure 28) to match the required types and operations (or terms) to the monitoring component features. It is envisaged that the SelectMonitor algorithm will be superseded in the 3rd year of the project with more sophisticated selection routines, most likely based upon some criteria passed with the Monitoring Features of preferential selection of Monitoring Components.

If any part of an Guarantee Term is not fulfilled (e.g. if no suitable monitoring component configurations can be established) an empty (null) MCS is returned for the Guarantee Term assessed. Otherwise, the MCS is configured as described in section 8.2.4.
Requirement: Given a set of input_types and a monitor term, select the first monitor that matches the term or event types required.

Input:
1. InputTypes – A set of types (e.g. Number, Event etc)
2. Term – A term or operation to be monitored (e.g. completion_time or <=)
3. Features – A list of Monitoring Component Features

Output: A monitor component offering the types and operation

Algorithm:
1. For each MonitoringFeature in Features
   a. Select FeaturedMonitors that equal Term
2. For each Monitor in FeaturedMonitors
   a. For each Type in InputTypes
   b. IF Monitor has Types
      i. Select SelectedMonitor
3. Select the first SelectedMonitor matching term and types
4. Return Monitor

Figure 27 MonitorSelection Algorithm
8.2.4 Input and Output Models

SLA Model (Input)

The input SLA model is as defined in the work package A1 deliverable D.A1a. The input dependencies for Year 2 are focused only on the Agreement Term sections where both Guaranteed States and Actions are defined. Note that Guaranteed Actions are not supported in Year 2 (see section 8.2.6). A sample part of an SLA model passed to the Monitoring Manager by the POC is illustrated in Figure 29.

```
VariableDeclr arrival_rate_vdec = new VariableDeclr(
    new ID("ARRIVAL_RATE_1"),
    new FunctionalExpr(common.arrival_rate, new ValueExpr[]{
        service_ref_vdec.getVar() }));
arrival_rate_vdec.setDescr("'ARRIVE_RATE'");
AgreementTerm loadTerm = new AgreementTerm(
    new ID("LoadTerm"),
    new TypeConstraintExpr( // precondition
        new FunctionalExpr(core.and, new ValueExpr[]{}),
        new SimpleDomainExpr(new CONST("FRIDAY", core.day_is),
            core.equals, null), null,
        new Guaranteed[]{
            new Guaranteed.State(
                new ID("CapacityState"),
                new TypeConstraintExpr(arrival_rate_vdec.getVar(),
                    new SimpleDomainExpr(new CONST("10", units.tx_per_s),
                        core.less_than),
                    new CONST("5", units.percentage))));
loadTerm.setDescr("Load Capacity Agreement Term");
.....
```

Figure 29 Partial SLA Java Object Code for an AgreementTerm

Monitoring Features (Input)

Component Monitoring Features (modelled in the Service Construction Model in work package A1 deliverable D.A1a) are specified for the type of component offered in monitoring aspects of a service or term expression. For example, the SENSOR description illustrated in Figure 30, specifies two monitoring measures; one for events reporting cpu_load and one the number of logged_users.

```
ComponentMonitoringFeatures
   MonitoringFeature
        @componentType=SENSOR
        @uuid=550-112-3523
        @type=cpu_load
        basic=primitive
        @type=NUMBER
   MonitoringFeature
        @type=logged_users
        basic=primitive
        @type=NUMBER
```

Figure 30 Sensor Component with Monitoring Features
Note that both sensor features return a basic, primitive, type of a number.

Another example of a component monitoring feature is for REASONER (or RCG - section 8.1) type components. REASONER features are described by a type (the term or operator performed), one or more input parameters and one output. The example in Figure 31 illustrates an RCG with two features. One providing a \textit{less\_then} comparison of two input parameter numbers. The other providing a MTTR (Mean-Time-To-Repair) computation output based upon request and response input events.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{reasoner_component监控特性.png}
\caption{Figure 31 Reasoner Component with Monitoring Features}
\end{figure}

\textbf{Monitoring System Configuration (Output)}

The Monitoring System Configuration (MSC) is also defined in the Service Construction Model (see deliverable D.A1a). There exists one MSC (illustrated in Figure 32) per SLA instance. The MSC contains a list of Components representing Sensor, Effector or Reasoners selected to support the Guarantee Terms of the Agreements in an SLA. Each Component in an MSC contains one or more component configurations for each of the different components. For example, an MSC can contain a Reasoner component which has component configurations for two Sensor components and one additional Reasoner component. A Sensor component configuration contains a MonitoringFeature (that which was advertised in selection for the Sensor component) and one or more OutputReceiver(s). An output receiver is another component which expects the result (as an event or value) to perform its own function. A Reasoner component configuration also specifies one or more OutputReceivers but a Specification replaces the MonitoringFeature with a list of Guarantee States required for reasoning by the component.
8.2.5 Implementation and Testing

The MonitoringManager Java implementation is provided in the `org.slasoi.gslam.monitoring` set of packages. The package structure and related implementation file topics are illustrated in Figure 33.

![Figure 32 Example MonitoringSystemConfiguration Structure](image)

![Figure 33 The MonitoringManager Implementation Packages](image)
The MonitoringManager Java interfaces are defined in a separate group of packages, under `org.slasoi.gslam.core.builder` and `org.slasoi.gslam.core.monitoring` as illustrated in Figure 34.

![Figure 34 The MonitoringManager Core Interface Packages](image)

To use the packages above the MonitoringManager can either be instantiated directly (through the MonitoringManagerImpl class) or by using the MonitoringManagerBuilder for which an interface is defined in the `gslam.core.builder` package. We now present some usage scenarios and code.

**Usage:**

```java
MonitoringManagerImpl (org.slasoi.gslam.monitoring.manager.impl)

MonitoringSystemConfiguration checkMonitorability
     (SLA model, ComponentMonitoringFeatures[] cmfs)
```

Given an SLA model object instance and an array of ComponentMonitoringFeatures, the checkMonitorability method builds a configuration of monitoring components supporting the Agreement and Guarantee terms of the SLA. The configuration is returned as an instance of a MonitoringSystemConfiguration object.

**Input Parameters:**

- `model` - The SLA model object instance. The model must be fully specified, i.e., all dependencies to external service interfaces have to be resolved.
- `cmfs` – An array of ComponentMonitoringFeatures selected as candidate monitoring components for the service specified in the SLA model.

**Output Parameters:**

- `MonitoringSystemConfiguration` - a configuration of monitoring components supporting the Agreement and Guarantee terms of the SLA.

**Testing with the MonitoringManagerBuilder:**

Test code is available in the `org.slasoi.gslam.monitoring.manager.demos.A4-A6` packages. The samples illustrate using the MonitoringManager builder to
create an instance of the MonitoringManagerImpl described previously. Below is an example of instantiating and call the MonitoringManager using the builder.

```java
import org.slasoi.gslam.core.builder.IMonitoringManagerBuilder;
import org.slasoi.gslam.core.monitoring.IMonitoringManager;
import org.slasoi.monitoring.common.features.ComponentMonitoringFeatures;

public class Test {
    public static void main(String[] args) {
        // create an object instance of the B3 SLA Template
        SLA sla = new B3SLAT();

        // create some sample monitoring features
        ComponentMonitoringFeatures[] cmfs = EFeatures.buildTest();

        // create a new builder and and instance of a Monitoring Manager
        IMonitoringManagerBuilder mmb = new MonitoringManagerBuilder();
        IMonitoringManager mm = mmb.create();

        // checkMonitorability and obtain a system configuration
        MonitoringSystemConfiguration config = 
            mm.checkMonitorability(sla, cmfs);
    }
}
```

### 8.2.6 Current Limitations

Some limitations of the current work are noted as follows. These limitations will form the basis for evolving the work and implementation in Year 3 and handling more complex SLA agreement terms and conditions.

- Preconditions (of Agreement Terms) are extracted but not assessed or resolved to components in the Monitoring Configuration.
- Guaranteed Actions (a set of actions that one of the parties to the SLA is obligated to perform) are not supported in the selection process and Monitoring System Configuration.
- Event types required as input by the Reasoning Component Gateways are currently specified in a local MM map repository.
- Parameter value types required by Terms are specified in a local MM map repository.
- Monitor Component Selection is currently based upon a simple “first-in-list” selection process.

### 8.2.7 Future Work

With reference to the selection process limitations highlighted in section 8.2.6, the Monitoring Manager currently provides only a simple selection process (matching monitoring requirements and features by types on a FIFO “first-in-first-out” basis). Future work will explore more optimised selection criteria, possibly passed in an extended specification for SLA models. Guaranteed Actions will also be parsed and, to support Guaranteed Action prerequisites, appropriate Reasoning Components provided in the Monitoring System Configuration. We
also aim to integrate the event and parameter types for standard terms in the SLA@SOI registry to relieve the maintenance of maps within the MonitoringManager.
8.3 Process Monitoring

8.3.1 Overview

Process monitoring (or processes-level monitoring) concerns the monitoring of these services that implement complex business processes, specified, e.g., in BPEL or other orchestration languages. The work undertaken in SLA@SOI on this topic, started in year 1 and proceeded through year 2, with the goal of extending the monitoring capabilities of SLA@SOI with a dedicated reasoning component able to address the specific requirements of process level monitoring.

This section reports the extensions and novel developments that have been undertaken during the 2nd year of the project on the process level monitoring framework described in Deliverable DA3.a, “SLA-Aware Service Management”, Section 4.2.2. The extensions reported here have been driven, on the one side, by the progress in the development of the whole SOA monitoring and management framework in the project; on the other side, by the development of the industrial case B6, “E-Government”, which has been used for experimenting the process monitoring framework.

The most important extension that has been implemented has solved one of the most serious limitations of the previous version of the framework, namely the limited capability to do event correlation. In particular, the Y1 process monitoring framework was supporting only two kinds of properties to be monitored: the “instance level” ones, which are evaluated on each single instance of the process to be monitored, and the “class level” ones, which aggregate events for all instances of a given process. The first kind of property allows one to monitor, e.g., whether any execution of a given process (e.g., the time to fulfil an e-government procedure of a given citizen) takes more than a given amount of time. The second kind of properties allows one to monitor, e.g., the average execution time of all instances of a process, and to check that it does not overpass a given threshold (e.g., whether the average time for fulfilling a given e-government procedure meets some KPI). The Y1 framework was not able to express in a convenient way all the properties in between these two extreme cases, when events need to be aggregated only for a given subset of instances of a process. This is needed, e.g., when different SLA guaranteed terms are defined for different classes of citizen that require the same procedure, and hence the average completion time needs to be computed only for the process instances corresponding to citizen in a given class. Moreover the Y1 framework was not able to correlate events coming from different processes, which is a typical situation, e.g., in e-government scenarios, where there are usually different parallel procedures by different stakeholders concurring to one common goal.

The second important extension with respect to the Y1 framework is a more flexible implementation that (1) facilitates language extensions and (2) permits the usage of the framework in isolation, as well as its integration in complex software landscapes. This extension has been necessary in order to push the integration of the framework within the SLA@SOI platform and to facilitate the usage of this component in the different industrial scenarios.

In the next subsection we will describe in more detail the new language that has been defined in order to support the powerful modelling of event correlations. In the following subsection we will describe implementation issues, while the final subsection of this section describes how the proposed framework has been integrated within the SLA@SOI platform.
8.3.2 Language

As explained in the previous subsection, the key feature of the new version of the process monitoring framework is the capability to define complex correlations among events. This capability is reflected into a new language for expressing monitoring properties, namely RTML v2. This subsection describes the correlation properties that the new language is required to be able to capture, as well as the proposed language.

The different forms of event correlation RTML v2 is required to model include:

- **Cross-instance correlation**

  This form of correlation gives the possibility to specify properties that correlate different instances of the same process. The evaluation of the property is done correlating events belonging to the execution of different instances of the same process. The events hence belong to the same domain data – the one associated to the monitored process. The events are selected by specifying conditions on these domain data that define (1) which process instances need to be aggregated, and (2) how this aggregation is performed. An example of property specifying cross-instance correlation is:

  \[
  \text{for each user, the average execution time of service } S \text{ must be less than } 10\text{ms.}
  \]

  This property requires the generation of one monitor for each user that is involved in the (domain data associated to the) process; each monitor aggregated the execution time of all the instances of the process corresponding to one specific user, and checks that the average time is less than 10ms. In addition to the event type and to the process type, the user identifier, which is one of the elements of the domain data of the process, is used to correlate all the events pertaining to each one of the monitors.

  One extreme case of cross-instance correlation includes all instances of a given process; in this case, the aggregation is done only on the basis of the event type and of the process type; this is the only case that was supported by RTML v1.

- **Cross-process correlation**

  This form of correlation gives the possibility to specify properties across different processes. The evaluation of the property is done correlating events belonging to different processes (or sub-processes or services within a process). An example is:

  \[
  \text{the average time between the completion of } S_1 \text{ and of the completion of } S_2 \text{ must be less than } 10\text{ms.}
  \]

  In this example, we assume that S1 and S2 are services belonging to two different processes, or to two different sub-processes of the same process.

  The key aspect here is that the two services have different data domains, since they are in different processes. In the easiest case, the elements used to correlate the services is shared by the two domain data, e.g., S1 and S2 share user identifier and protocol identifier, and these are sufficient to correlate them for the computation of the time between the two completions. In other cases, this is not possible: S1 and S2 may share the user identifier, but they may use different session identifiers S11 and S12, and hence the correlation cannot be unique. In this case, only the parent
process P is able to correlate SI1 and SI2, and hence the two services S1 and S2.

- **Cross-layer correlation**
  
  While the monitoring framework that is described in this session focuses on a specific layer of a service-based application, namely the software service layer, the properties to be monitored often depend on properties and events that are defined at different layers. The evaluation of the property hence requires the correlation of events belonging to the process with events belonging to other layers. An example is the following one:

  *when the system load SL is less than 80%, the average execution time of S must be less than 10ms.*

  In this case we need to correlate service S with the load of the system that executes S. Similarly to what happens for cross-process correlation, S and SL do not share the same domain data; this aspect is actually emphasized by the fact that S and SL express properties of different layers of the system. In order to be able to correlate S and SL, the monitor should not only be able to access events of the infrastructure, such as system load, but also to properties that associate a given infrastructure to a specific service, e.g., it should be able to determine on which system (e.g., virtual machine) a specific service is executed.

  The capability of expressing all these correlations is the key feature of RTML v2. The language should also inherit from RTML v1 the capability to express different types of properties to be monitored, and in particular:

  - **Boolean properties:** i.e., properties that, in each moment in time, are either true or false;
  - **Time related properties:** i.e., properties that express duration or time intervals;
  - **Statistic properties:** these properties have a numeric value that represent, for instance the count of a given event, or a percentage, or also a sum or an average of a quantity that is correlated across the instances of a given process.

  These types of properties need to be combined and nested in arbitrary ways.

Figure 35 provides the grammar of the RTML v2 language. A RTML v2 program contains a list of event declarations and of monitor declarations. Events are defined by an identifier and by a list of correlation parameters. These parameters are used to select a given subset among all the events of a certain type. Of course, a mechanism is needed to "ground" these events, and in particular, the declarations of correlation parameters, to the actual structure of the events received by the monitor. As this is implementation-specific, the language is completely agnostic with respect to this grounding of events.

Similarly to events, monitors are also defined by an identifier and by a list of correlation parameters. Moreover, monitors are associated to an expression that defines the property to be monitored. In the grammar in Figure 35, most of the operators within the expression are inherited from RTML v1, and have been already explained in the previous deliverable. Moreover, as already commented, the language is extensible, and it is hence easy to add new operators. For this reason, here we do not explain the meaning of these operators, but we rather focus on the main innovation of the language with respect to RTML v1, namely usage of correlation parameters.
In the grammar for expressions, correlation parameters appear in two different places. First, they appear as restrictions associated to (identifiers of) events and monitors: this allows for selecting only a subset of the events of a certain type, according to the values of the correlation parameters. Second, they are used to introduce new correlation parameters associated to the aggregation operators.

```
program := (eventdecl | monitordecl)*
eventdecl := 'EVENT' ID [' correlationParameterDeclaration '] '
monitordecl := 'MONITOR' ID [' correlationParameterDeclaration '] := expr ;
expr := bool |
num |
bool '?' expr ':' expr
bool := ID (' correlationParametersRestriction ') |
ID (' correlationParametersRestriction ') . query |
bool '||' bool |
bool '&&' bool |
num '=' num |
num '<' num |
'once' '(' bool ')' |
history '(' bool ')' |
bool 'since' bool |
'OR' [' correlationParametersDeclaration '] '(' bool ')' |
'AND' [' correlationParametersDeclaration '] '(' bool ')' |
TRUE |
FALSE
num := ID (' correlationParametersRestriction ') |
ID (' correlationParametersRestriction ') . query |
num OP num |
'time' '(' bool ')' |
'time' '(' bool '; bool ')' |
'COUNT' [' correlationParametersDeclaration '] '(' bool ')' |
'AVERAGE' [' correlationParametersDeclaration '] '(' num ')' |
NUMBER
correlationParametersDeclaration := ID [', correlationParametersDeclaration]
query := ID [', query]
correlationParametersRestriction := correlationParameterRestriction [',
correlationParametersRestriction]
correlationParameterRestriction := ID '=' STRING |
ID '=' NUMBER |
ID '=' correlatorReference |
ID '=' TRUE |
ID '=' FALSE
correlatorReference := '$' ID
```

Figure 35: RTML v2 language definition.

We start by illustrating the first usage. Assume that event

```
EVENT S_completed[uid, sid]
```

captures the completion of an execution of service S associated to user uid and session sid, and assume that S_completed.time is the execution time for S. Then:
defines the events corresponding to the completion of all service executions for user ‘Alice’. Formula:

\[ \text{AVERAGE } S_{\text{completed}}(uid = \text{Alice}).time \]

computes the average completion time for all executions of service S for user ‘Alice’. Finally

\[ \text{MONITOR } S_{\text{avg completion time}}[\text{user}] := \text{AVERAGE } S_{\text{completed}}(uid = $\text{user}).time \]

defines a monitor that computes the average completion time for all executions of service S for a generic user. Of course, at run-time, we expect to have different running instances of this monitor, one for each user of service S.

We illustrate now the second usage of correlation parameters in monitor expressions, namely the introduction of additional correlation parameters associated to aggregation operations. Let us assume that we want to compute the worst (i.e., maximum) average completion time for service S over all users. We can define this monitor on top of \( S_{\text{avg completion time}}(\text{user}) \), as follows:

\[ \text{MONITOR } S_{\text{worst avg completion time}} := \text{MAX } [u] S_{\text{avg completion time}}(\text{user} = u) \]

That is, correlation parameter \( u \) is introduced in order to range on all users for which we monitor the average completion time.

As a final example of the usage of correlation parameters, let’s assume that there is a process P that executes two services S1 and S2, and that we want to measure the average time between the completion of S1 and the completion of S2. If S1 and S2 share a session identifier, this property is easy to define the monitor as follows:

\[ \text{AVERAGE}[\text{sid}] \text{time}(S1_{\text{completed}}(\text{sid}).time, \ S2_{\text{completed}}(\text{sid}).time) \]

If they do not share a session identifier, we could use a similar formula, namely:

\[ \text{time}(S1_{\text{completed}}(\$s1id).time, \ S2_{\text{completed}}(\$s2id).time) \]

We need however to associate the correlation parameters \( $s1id \) and \( $s2id \). We can exploit, for instance, the following events:

\[ \text{EVENT P_spans_S1}[\text{pid},\text{sid}] \]
\[ \text{EVENT P_spans_S2}[\text{pid},\text{sid}] \]

which capture the invocation of the services from the process, and which define the relation between the process identifier \( \text{pid} \) and the service identifiers \( \text{sid} \). More precisely, the property we are interested in can be defined as follows:

\[ \text{AVERAGE}[\text{pid},\text{s1id},\text{s2id}] \]
\[ (P_{\text{spans}}_S1(\text{pid}=$\text{sid},\text{sid}=$s1id) \ & \ P_{\text{spans}}_S2(\text{pid}=$\text{sid},\text{sid}=$s2id)) \rightarrow \]
\[ \text{time}(S1_{\text{completed}}(\$s1id).time, \ S2_{\text{completed}}(\$s2id).time) \]

This guarantees that the time interval is computed only for corresponding pairs of execution of services S1 and S2.

### 8.3.3 Implementation

As discussed in the overview of this session, the main principle that guided the design and implementation of the RTML v2 framework has been flexibility. In particular, we have designed the framework so that it can work both in isolation and as an integrated part of the SLA@SOI platform. Moreover, the integration within the SLA@SOI platform has been made flexible, so that the relations and interplays of RTML v2 with the other monitoring engines and with the other
components of the system can be adapted, e.g., to the specific needs of the different application scenarios.

More in detail, the principles that have been adapted to the definition of the architecture are:

- The monitor engine is agnostic with respect to the source of the events and to the generation of these events, in order to facilitate the adoption in different settings and with different roles.

- Separation of rule translation and rule execution: in order to facilitate the extension to the RTML language, or also the replacement of the RTML language with other property specification languages, the monitor lifecycle is separated in two steps, namely the translation of the rules into executable code implementing them (rule translator), and the execution of these rules (rule runtime). The rule translator should fully automate the process of transforming the input rules into executable code. The rule runtime should be as agnostic as possible with respect to the specific features of the rule specification language.

- Prominence of correlation aspects within the run-time. As correlation is the main concern of the proposed approach, it needs to be separated from the other aspects of the monitoring process (e.g., management of the different monitors and management of the notifications). This way, the correlation manager can be used as a stable core component, while the other components can be adapted to specific contexts (e.g., the notification management can be modified as prescribed by the usage scenario).

Form a high level, the usage of the framework we foresee is the following:

- The monitor domain – meaning the events and monitors for a specific usage context – are modeled in RTML v2 (or an alternative language).
- The monitor domain specification is then translated into a set of specific classes which are deployed in a monitor engine.
- The engine is responsible of feeding these classes with the expected events captured by service-specific captors (the components that in the SLA@SOI framework are called sensors) and manage the monitor life cycle.

The defined high-level architecture is depicted in Figure 36.
Figure 36: RTML v2 architecture.

The integrated monitoring framework consists of three components:

- **The Monitor Engine**: this is the core component of the framework, and implements all the monitor capabilities. The monitor engine console consists of two main sub-components, namely the Rule Translator, and the Rule Runtime.

The Rule Translator consists of two sub-components, corresponding respectively to the parsing of RTML properties and to the emission of executable monitors in terms of Java classes. Also this partition of the Rule Translator has been done to facilitate extensions to RTML v2, or adaptation of the framework to other languages.

Each monitor rule is translated in two Java classes. The "monitor class descriptor" class provides a static description of the monitor, such as the property to be monitored and the events the monitor is interested to reason on; this class will be instantiated only once by the engine. The "monitor" class contains the dynamic aspects of the monitor, such as its status and the correlation set; this class will be instantiated dynamically, possibly several times, once for each instance of each monitor associated to specific correlation values.

The Rule Runtime consists of three sub-components: the Correlation Manager has the responsibility of associating monitored events to the specific monitor instances that are waiting for them; the Monitor Handler is responsible of evolving monitors depending on the received events, as well as to create new monitor instances when needed, or to terminate completed instances; the Notification Manager, finally, is responsible of managing the notification of the status of the monitored properties, both in a pull and in a push modality.

More precisely, the behaviour of the engine, when it receives an event, is the following. (1) It gets the set of monitor classes interested in the event type (the engine uses the "monitor class descriptor" for this purpose). (2) For each class, it gets the monitor instances having a correlation set that
matches the received event. (3) If there is no matching monitor instance, then it create a new one and initializes it with the new correlation. (4) It sends the event to the associated (new or old) monitor instances. (5) These monitor instances update their status according to the received event. This may also trigger the update of other associated monitor instances. (6) The monitor updates may trigger notifications of state changes or of property violations.

- The **Service Container**: this is an abstraction of all the rest of the service-based application. The only relevant aspect is that we assume that there are service and process models from which we can define events to be captured. The service and process models are instantiated in service/process instances that are executed by service handlers. Monitor captors (sensors in SLA@SOI) are capable to extract the relevant events from the execution of these instances, and to push them on a monitor bus. A Bus Listener within the Monitor Engine is then responsible of accessing these events. As already stated, this component is abstract, and need to be instantiated to the specific usage context of the monitoring engine.

- The **Monitor Console**: this component allows for managing and controlling the monitor engine. More precisely, it gives the possibility to interact with the engine, to deploy monitors, to send events/messages and view monitor instances. It also allows for accessing to status of the running monitor instances. This component is realized as an API, in order to facilitate adaptation and reuse.

### 8.3.4 Integration in SLA@SOI

The integration of RTML v2 within SLA@SOI has exploited the flexible design and implementation of the framework.

The component of the framework integrated in the SLA@SOI platform is the Monitor Runtime. The Service Container is implemented by the SLA@SOI platform, while the implementation of the Monitor Console is delegated to the specific industrial case that adopts the monitoring framework.

For what concerns the specification of the monitoring rules, we support both the SLA and SLA Template as starting point to generate the monitors.

From an architectural perspective, the Monitor Runtime is integrated in SLA@SOI as a ReasoningComponentGateway. A wrapper fulfilling the SLA@SOI specification for this component has hence been implemented. More precisely this wrapper has to perform three tasks: (1) to translate the SLA and SLA Template to the format supported by the RTML v2 framework; (2) to translate the SLA@SOI event format to the event format of our framework, and to dispatch these events to the Monitor Runtime; and (3) to translate the monitor result events from the format of the RTML v2 framework to the SLA@SOI format.

Translation (1) of the SLA and SLA Template to the RTML v2 format consists in two steps. The first step translates the guaranteed state and action present in the
SLA and SLA Template. In this step, for each guaranteed state or action, one or more monitoring Java classes are generated and deployed on the Monitor Runtime. This generation and deployment of Java classes is done at run-time, whenever a new SLA to be monitored is passed to the rule translator. The second step consist in the subscription thought the internal SLA@SOI monitoring event channel to the proper events, thus enabling the computation of the formulas belonging to the guaranteed states and actions.

Translation (2) of the SLA@SOI event to the RTML v2 event format is required because the information contained in the two event formats is compatible but is represented with different data structures. The translation however is pretty straightforward and just copies data from a data structure to one other.

When a monitored formula is violated, a Monitor Result Event compliant with the SLA@SOI specification must be create and push to the SLA@SOI framework. The translation between the RTML v2 and the SLA@SOI format is again a simple process, since it corresponds to the compilation of a predefined XML data structure. The XML data structure is than pushed back to the monitor event channel, since this is the approach adopted by SLA@SOI to signal violations.

![Diagram](image)

**Figure 37: Architecture of the SLA@SOI wrapper.**

Figure 37 describes the architecture of the Reasoning Component Gateway wrapping the RTML v2 Runtime Monitor. The proposed architecture is coherent with the architecture defined for the RTML v2 Monitor Engine (see Figure 36), and exploits the flexibility degrees which have guided its definition. In more detail, the integration within the SLA@SOI framework required a redefinition of the Rule
Translator and the Bus Lister. A specific notification plugin has been implemented in order to enable the notification of the monitoring result to the Monitoring Event Channel; this issue required the introduction of the SLARepository, which is a component in charge for maintaining the relation between the guaranteed state or action to the generated monitoring class.

### 8.3.5 Future Work

The future work for what concerns process level monitoring will target the validation of the proposed solutions for event correlation. This validation will be performed on the SLA@SOI industrial cases, with a specific focus on the e-government case. The validation will evaluate in particular (1) the capability of the proposed notations and mechanisms to capture all the relevant cases of event correlation and (2) the capability of the monitoring engine to handle these event correlation cases in an efficient way. If needed, extensions and modifications of notations, mechanisms and engines will be provided.

### 8.4 Adjustment: Autonomic Adjustment

#### 8.4.1 Overview

In section 8.2, the monitoring solution in SLA@SOI at the software level has been described. This SLA level monitoring system can detect SLA violations but does not always provide sufficient information for deciding the most appropriate response to a problem. Such decisions often require additional diagnostic information that explains why a violation has occurred and can, therefore, indicate what would be an appropriate response action to it. This is the main objective of the Adjustment Component described in this section, thus helping the service providers to guarantee their SLA commitments not only by detecting SLA violations once they have occurred but further by anticipating these violations to trigger the appropriate preventing actions.

The Adjustment functionality aims to minimize the adverse impact on the business level of incidents and problems that are caused by errors within the lower layers (software and infrastructure), and prevent recurrence of incidents related to these errors. This module is responsible for investigating the problem, identifying the root cause and resolving the issue. In case of an SLA violation, the Adjustment module can trigger re-planning, re-configuration and/or alerting to higher-level SLA. These capabilities are considered to be important in order to guarantee best user perception preserving underlining resources.

Besides the corrective actions to be taken upon a SLA Violation, another important feature from the Service Provider point of view is the possibility of preventing possible breaches of Agreements. Based on the recent behavior of a service, the Software Prediction Component (see work package A6 deliverable D.A6a) can predict the probability of an SLA violation in the near future, and start countermeasures if appropriate. Using this information, the Adjustment Component can help to proactively avoid SLA Violations.

The importance of the Adjustment is clear from a business perspective. Most Guarantee Terms set in a Service Level Agreement are subject to direct financial penalties or indirect financial repercussions if not met. Hence the relevance of trying to avoid the malfunctioning of the service, or, in case a violation occurs, to restore the normal situation in the quickest way possible.

**Autonomic overview**
Autonomic computing is an initiative started in 2001 by Paul Horn, Vice President of IBM [11], aiming to develop systems capable of self-management. As a solution increasing complexity of the system and communication infrastructure he suggested to design and build future systems and infrastructures capable of running themselves, adjusting to varying circumstances.

For this approach, he proposed to take the massively complex systems of the human body as a model. The human body performs a set of tasks such as controlling temperature or sweating, the rate of heartbeat, breathing, etc., involving a lot of independent but interlinked entities (glands, organs, etc.), being controlled by the autonomic nervous system, hence the name "Autonomic Computing".

There is an agreement in defining an autonomic system [12] as the one that manages itself without external intervention, able to adapt its operation following the evolution of its operational context and its internal state, even in case of unpredictable environmental changes, to satisfy high-level system management requirements and specifications.

Self-management [13] is considered to be the essence of Autonomic Computing. IBM frequently cites four aspects of self-management: self-configuration, self-optimization, self-healing and self-protecting:

- **Self-configuration**: components and systems must be able to adapt their configuration in an automated and seamless way according to high-level policies. The rest of the system detects the change in the context and adjusts automatically.
- **Self-optimization**: components and systems will continually seek opportunities to improve their own performance and efficiency.
- **Self-healing**: that is, automated detection, diagnosis and repair of localized SW/HW problems.
- **Self-protection**: the system must automated defence itself against malicious attacks or cascading failures. It will make use of early warning to anticipate and prevent wide failures.

IBM has used this definition to build the MAPE-K architecture [14] (Monitor-Analyze-Plan-Execute-Knowledge), as shown in Figure 38.

![Figure 38: MAPE-K Architecture](image-url)
The architecture based on the autonomic element (AE). The AE has to know the environment and how to influence it in order to keep it in optimal conditions, without the need of any external operation.

The autonomic architecture implements an intelligent control loop. For a system component to be self-managing, it must have an automated method to collect the details it needs from the system; to analyze those details to determine if something needs to change; to create a plan, or sequence of actions, that specifies the necessary changes; and to perform those actions. When these functions can be automated, an intelligent control loop is formed.

The Autonomic Element is composed of four parts that share knowledge:
- The **Monitor** function provides the mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from a managed resource.
- The **Analyze** function provides the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.
- The **Plan** function provides the mechanisms that construct the actions needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
- The **Execute** function provides the mechanisms that control the execution of a plan with considerations for dynamic updates.
- **Knowledge**: The autonomic element employs knowledge to interpret the information from the environment and to perform the appropriate actions. It forms a space of understanding among all the blocks. It is defined using semantic technologies.

Apart from the IBM's pioneer work on this area, it is worth mentioned the FOCALE architecture [15], and the four SAC projects: ANA [16][17], BIONETS [18][19], CASCADAS [20][21] and HAGGLE [22][23].

**8.4.2 Design**

The Autonomic Management of SOA components aims to build systems with self-CHOP capabilities (self-CHOP stands for self-Configuration, self-Healing, self-Optimization and self-Protection[40]). That is, systems able to configure themselves, able to make an auto diagnosis of themselves and recover from failures, able to find the optimized way to execute themselves and able to protect themselves of possible external threatens.

In order to apply Autonomic Techniques to SOA components, it becomes clear that the resulting system should be highly scalable, which made an agent-based architecture very suitable. The main building blocks of the proposed architecture are depicted in Figure 39.
The main actors in this architecture are Managed Elements and Agents. Managed Elements are defined as components with direct access to the resources. Examples of resources are not only software elements from the software landscape (SOA components, e.g. ORC, the Open Reference Case) but any generic infrastructure resource. These Managed Elements (MEs) are controlled by an Agent, which provides the ME with autonomic capabilities. Agent is capable of querying the static and dynamic properties of the Managed Element. Figure 39 shows how an agent extends a Managed Element, so the behavior of an agent could be controlled by another agent.

An agent can have several tasks assigned to it, e.g. to receive monitoring events, to analyze incoming information or to execute adjustment actions. A Task evaluates rules and, when the pre-conditions are met, executes the post-conditions, which in turn are Actions to be performed on the Managed Elements. An Action is an implementation of a generic Command, and the actions to be executed are queued in a Job Manager facility in order to prevent congestion in the resources.

By means of using a generic agent and task implementation, some activities like configuring the agent or the tasks, starting or stopping them all could be implemented once and only those domain-specific activities (namely the Actions) need to be further worked out.

In order to allow the communication between agents, asynchronous messaging through a publish/subscribe bus is used. To that end, Agents implement a Messaging interface to be able to send and receive messages from another Agent. A bus channel can be assigned to each message type, so agents subscribe to those channels associated to messages needed by the tasks they are executing.

Figure 39: High-level view of the proposed architecture
8.4.3 Implementation

The architecture described above provides a unified management interface for agents that can be reused within the project. In particular, it will be used in the Provisioning and Adjustment Components inside the SLA Managers. Being the PAC domain specific, there will be several implementations: reference implementations focused on the Open Reference Case (ORC), both software and infrastructure, as well as implementations for the industrial use cases.

The idea behind the implementation is to keep the generic components clearly separated from the domain-specific ones, and the latest as configurable as possible.

This section focuses on the actual implementation of the components of the proposed architecture that have been developed during the second year of the project. In order to avoid a very long and technical section, only the implementation of the main components (Agent, Task and Configuration) is described in some detail. For more complete information, please refer to the appropriate Javadoc documentation.

Agent

Basically, an Agent is a pool of tasks (see Figure 40). The interface IAgent defines the basic characteristics an agent should implement: start, stop and configuration methods. The Agent class, besides implementing IAgent, also implements the Runnable interface so it can be executed in its own thread. The agent also maintains a map of the Managed Elements under its control, so it can send control commands to them.

The agent has access to an instance of some communication mechanism (namely a publish/subscribe bus). Through this communication channel, the agent receives commands (e.g. shutdown, restart), as well as events from external systems (like a monitoring system).

When the Agent is created, it reads its configuration from an xml file through an AgentConfiguration object. The configuration must contain the agent configuration channel, messaging settings and tasks configurations. The latter is used to instruct the agent about the tasks it is in charge of.

Details about configuration of agents and tasks will be provided below in a specific section.
Task

A Task is an abstract class that encapsulates a specific functionality (see Figure 40). The basic characteristics of a task, namely start(), stop() and configure() methods, are defined in the interface ITask. A Task corresponds to a specific action, such as reading events from the event bus or performing the analysis of the incoming events, while an Agent can execute one or more tasks at the same time.

When the Task is created, it receives its configuration from a TaskConfiguration object. The configuration contains the task identifier, the actual class to be instantiated and, optionally, a list of parameters needed by the task.

Details about configuration of tasks will be provided in the next section.
For this architecture to be reusable in generic and domain-specific components, it needs to be highly configurable. For this purpose, both agents and tasks, read its own configuration from an xml file.

Typically, this configuration file is composed by the configuration of at least one agent, represented by the AgentConfiguration class. Each agent can configure the tasks assigned to it. Since it is expected that some components will need specific tasks (inherited from the abstract Task class), the TaskConfiguration contains not only the identifier of the task but also the class name of the Task to be instantiated, and the parameters needed for the actual configuration of the task.

The parameters are stored in a generic hash list, where the key is a name that uniquely identifies the parameter.

When the system is started, a Configuration object is created, which reads the xml configuration file and deserializes it. The deserialization process is done using xstream library, and produces the corresponding AgentConfiguration objects, which in turn include a set of TaskConfigurations. Then, agents are created with the specified configuration.

Tasks are instantiated to the indicated class name, using reflection. This allows an easy extension of the system: if a component needs to create a specific kind of task class, then it can be added to an Agent just including the new class name in the configuration file, without modifying existing code. The only requisites are the class should inherit from the abstract Task class, and must provide a default constructor so RTTI (Run Time Type Identification) can be applied.

**AgentConfiguration and TaskConfiguration**

![Task Class Diagram](image-url)
The following code snippet shows an example of an xml file to configure an agent and its tasks. In this example, an agent is created and two tasks assigned. The first one, the EventTranslationTask, that translates events of the specified format (ManageabilityAgentMessage) using the given translator (MAMessageTranslator). The second task connects to the event bus specified in the properties file, and keeps listening to the designated channel.
In this section, we briefly outline some of the basic tasks that have been implemented as part of the architecture. These tasks provide common functionality that will probably be used by any component implementing the architecture. If a new type of task is needed, it can be extended and easily assigned to the agent, as described in the previous section.

```xml
<AgentConfiguration>
  <agentId>Manager</agentId>
  <TaskConfiguration>
    <taskId>EventTranslationTask</taskId>
    <className>
      org.slasoi.gslam.pac.EventTranslationTask
    </className>
    <parameters>
      <entry>
        <string>ManageabilityAgentMessage</string>
        <string>org.slasoi.gslam.pac.MAMessageTranslator</string>
      </entry>
    </parameters>
  </TaskConfiguration>
  <TaskConfiguration>
    <taskId>ConnectionEventBus</taskId>
    <className>
      org.slasoi.gslam.pac.EventBusHandlingTask
    </className>
    <parameters>
      <entry>
        <string>BUS_PROPERTIES_FILE</string>
        <string>eventbus.properties</string>
      </entry>
      <entry>
        <string>CHANNEL1</string>
        <string>PROVISIONING_RESULT_EVENT_CHANNEL</string>
      </entry>
    </parameters>
  </TaskConfiguration>
</AgentConfiguration>

Basic tasks

In this section, we briefly outline some of the basic tasks that have been implemented as part of the architecture. These tasks provide common functionality that will probably be used by any component implementing the architecture. If a new type of task is needed, it can be extended and easily assigned to the agent, as described in the previous section.
**EventBusHandlingTask**

This task connects to a publish/subscribe messaging bus, and keeps listening to one or more channels. The messages received through the bus will typically inform about the status of the Managed Element: whether it has been provisioned correctly or not, if it is running without problems or if some parameter is out of bounds.

**EventTranslationTasks**

Events received through the event bus will usually need a translation to some format understandable by the application before being analyzed. This translation is carried out by ad-hoc EventTranslationTasks. Each EventTranslationTask has attached a message type and methods to serialize and deserialize events. Therefore, it allows the incoming messages to be converted to the internal event format, and also the conversion of internal events to be sent to another component.

**AnalysisTask**

This task analyzes incoming data and extracts a conclusion and possibly a suggestion of action to be triggered. The Analysis Tasks relies on a rule engine in order to determine the impact or potential impact of the received events and to determine the action or set of actions to take.

Drools [24] has been chosen as the rule engine, since it presents characteristics that fulfils the requirements of the architecture, such as sliding windows, possibility of correlation, rule packages, possibility of defining priority for each rule, complex event processing or temporal reasoning. Furthermore, rules can be changed on the fly, and it provides auditing functionality, which is an interested feature to keep a record of how decisions were made.

**PersistencyTask**

This task is responsible for the permanent storage of data. It decides when to store data in the database without disturbing the normal functioning of the system, and also implements a mechanism for accessing the historical database when needed. The implementation of this task uses hibernate [25] with sql, which facilitates the storage and retrieval of Java objects through object/relational mapping.

### 8.4.4 Autonomic adjustment in SLA@SOI

Previous section has described the proposed architecture for the Autonomic Management of SOA Components. The agent-based system has been used as part of the SLA@SOI framework for the functionality related to the SLA enforcement. In particular, inside the A3 work package, for the adjustment of the software layer. To this end, an agent has been deployed in the Provisioning and Adjustment Component (PAC) of the Software SLA Manager. The agent manages the underlying Software Service Manager, allowing the provisioning of the services following a given SLA, as well as ensuring that the service runtime operation fulfils the corresponding Agreement.
With this objective, the agent starts an EventBusHandlingTask, that keeps listening to the event bus, to the channel where the monitoring infrastructure sends its data. The incoming information corresponds to two type of data: SLA Violations and SLA Prediction Events. The former are created when the SLA Level Monitoring System finds out that one or more GuaranteeTerms of a given SLA have been violated. This conclusion is reached through the comparison of the instrumentation information (raw monitoring data) with the guarantees written in the SLA. On the other hand, Prediction Events are triggered when the software prediction system deduces that there is certain probability that a violation of one of the GuaranteeTerms of an SLA occurs.

Once these events are read by the EventBusHandlingTask, they are then passed to an AnalysisTask, which, using a drools engine, inspects the incoming information and decides if some action needs to be triggered. To this end, specific rules have been implemented for the adjustment at the software level. Example of actions include: sending of notifications to the Business Manager, where the economical impact of the possible violations are studied; reconfiguration of the software (BPEL) engine; restart of the service, or even a renegotiation of the agreement.

Finally, the triggering of actions is planned and executed, using the <<manage_software_services>> interaction offered by the Software Service Manager, the <<control/track>> interaction with the Business Manager.

For illustration purposes, an example of adjustment rule is shown here:

```java
//Detects a violation indicating that the response time of a given service does not fulfill the SLA, and restarts the service.

rule "Violation on Response time"
when
    $violation : SLAViolationEvent( type == "response time",
    $serviceBuilder : serviceBuilder, $responsetime : value);
    $ssm : SoftwareServiceManager();
    $bm: BusinessManager();
then
    $ssm.executeAction(stopServiceInstance($serviceBuilder));
    $ssm.executeAction(startServiceInstance($serviceBuilder));
    $bm.notifyViolation($violation);
retract($violation);
end
```

This example shows a rule that acts upon the reception of a violation on the guaranteed response time of an SLA. In that case, the service is restarted and the business layer alerted to evaluate the economical impact of the violation.

A similar process is followed with the prediction data, using a dedicated Analysis task to evaluate the data. In this case, the information serves to prevent a violation of the SLA signed with the customer.
Therefore, the proposed architecture has been applied to allow the Monitoring, Analysis, Planning and Execution components of a typical autonomic system, while the Knowledge component is implemented using a PersistencyTask to store in a knowledge database the different statuses of a service, as well as the results of the analysis and actions triggered. Hence, the proposed architecture and implementation provides of autonomic capabilities to the software layer for adjustment purposes.

D.A5a (SLA Foundations) gives more details about the usage of the architecture in the different Provisioning and Adjustment Components of the SLA@SOI framework.

### 8.4.5 Future Work

Adjustment in Task A3.5 has been proved as a suitable architecture to be used in the different components throughout the project, in particular for the SLA enforcement based both in monitoring and prediction events. Future work will further research novel algorithms in order to analyze the economical impact of the incidents and problems that are caused by errors within the lower layers (software and infrastructure).

### 9 Multiprovider/Multidomain Service Management

#### 9.1 Overview

Internet of Things, Architecture of Participation, and Cloud Computing are some of current trends in ICT that are expected to have strong impacts on service delivery evolution. Architecture of participation, embodied, for instance, by the so-called Web2.0, encourages users assuming an active role in the provision of contents and services (prosumers), and enables alternative business models to accommodate the long tail of the market. In parallel, Telecom Operators are deploying solutions aligned with Web2.0 principles to enable user-generated services through the exposure and mash-up of network features and data.

Value-added services like automated discovery, negotiation and service composition, favors the appearance of service aggregators and service brokers. A service aggregator plays the role of a service provider that offers an aggregated business function view of discrete separate business functions. That is, it consumes multiple services from different providers and aggregates their results behind the facade of a service offering a single business function. Thus the service aggregator operates as a service provider to its customers and as a service requestor to its suppliers.

In such a dynamic marketplace, where the final service delivered to the customer is composed by services of different providers, the responsibility on the well functioning of the service is spread over the different entities involved. In this context, the signing of Service Level Agreements between the different actors (Service Providers-Service Aggregator, Service Aggregator-Customer), establish the limits and responsibilities of the parties, as well as the quality under which the services (the atomic and the composite ones) must be delivered.
For the SLA@SOI framework (described in work package A1 deliverable D.A1a) to be successful in today's business environment, a clarification of how to use it in a multiprovider-multidomain environment is of vital importance. This is the main goal of this task: to analyze and describe how the framework will allow the SLA management on whole end2end services in composition and brokering processes from different service providers.

Both B5 and B6 use cases have requirements for a multiprovider environment, but any B-line workpackage has requirements for a multidomain environment, at least for the second year of the project. For this reason, we show here how the framework can be applied to deal with a multi-provider environment, and leave to Y3 the deep analysis of the multidomain case. In order to better explain the usage of the SLA@SOI architecture, we have chosen a scenario approach, that is, the adoption of the framework is shown through a revision of the each of the most important scenarios of the SLA management process.

### 9.2 Scenarios

This section provides a detailed analysis of the SLA management in an environment where more than one provider is involved. First, the main scenarios of the SLA management are described. Each scenario description is first decomposed in the main steps, followed by an explanation of how these steps can be accomplished with the SLA@SOI architecture. To that end, the components and interactions of the framework, described in deliverable D.A1a will be mentioned.

For the sake of clarity, and without any loss of generality, the scenarios will be described with the following actors:

- **end-customer**: end-user that consumes services offered by the Service Aggregator or Service Broker.
- **3rd party Service Provider A (SP A)**: Service Provider offering its services to the Service Aggregator or Service Broker.
- **3rd party Service Provider B (SP B)**: Another Service Provider offering its services to the Service Aggregator or Service Broker.
- **Service Aggregator or Service Broker**: Service Provider that consumes services of Service Providers A and B, makes a new composite service and offers these new services to end-customers. In the following, SP Agg. will be used to denote the Service Aggregator or Service Broker. In order to be generic, in the following we assume that the Service Aggregator can also have infrastructure of its own, and therefore be able to combine its own services with the ones offered by SPs A and B.

Each one of the Service Providers (A, B and Agg.) is supposed to be running an instance of the SLA@SOI framework.  

It is worth mentioning here that, even in the more basic multiprovider scenario, more than one SLA is involved. Apart from the SLA signed between the end-customer and its service provider (the Service Aggregator), the later will sign an SLA with the 3rd parties for each one of the external services it consumes. Similarly, each one of the service providers will describe its available services using SLA templates.

---

11 In case any of the providers does not implement an SLA@SOI framework, it must at least implement the interfaces needed to interact with the framework.
The first four scenarios cover all activities that need to be performed by the Service Agg. prior to the negotiation of a service. That includes 3rd party management, 3rd party service discovery and the activities connected to the design and development of the composite services. Then, scenarios with regard to negotiation, provisioning and run-time are analyzed.

9.2.1 Third-party registration

This scenario deals with the registration of SP A and B as 3rd parties in the framework of the Service Agg. The registration includes the assignation of user and password credentials, so the 3rd parties can later on interact with the Service Agg. in a secure way.

Framework components involved:

- BusinessManager of the Service Agg., SP A and SP B.

Interactions involved:

- <<provider_relations>>, that allows the registration and modification of the information of a 3rd party, as well as the provider authentication.

9.2.2 Third party services discovery

This scenario covers all activities that need to be performed by a Service Provider that acts as a Service Aggregator or Service Broker to discover 3rd party services that may be used to compose a new product.

ServiceProvider interested in selling its services or consuming services from a 3rd party can deploy its own advertisement bus. ServiceProviders interested in advertising themselves publish events in the bus, while those ServiceProviders interested in buying 3rd party services subscribe to advertisement events. The sequence of the interactions is the following:

1. ServiceAgg. deploy its own advertisement bus, and subscribe to advertisement events, using the <<publish/subscribe advertisements>> interaction of the SLAManager.
2. Service Agg. SLAMs subscribe to advertisement events. Policies will be used to describe security requirements. Business Manager provides the SLAMs in the framework with policies (<<control/track>> interaction), and the SLAMs would then know who is a trusted 3rd party and who is not.
3. 3rd party SLAMs can publish templates in the advertisement bus. The bus coordinates (host, port,...) are indicated either directly via config options,  

---

12 For the sake of clarity, we have chosen here an scenario where the Service Agg. owns the bus, but any of the providers could deploy the advertisement bus. If a 3rd party is the one that owns the bus, the Service Agg. will have to subscribe to advertisements, but the overall procedure is similar to the one described in the text.
or through some other mechanism where the Business Manager is involved (that is, Business Manager indicates to lower level SLAMs where to publish events, through the policies exchanged in the <<control/track>> interaction).

4. 3rd party SLAMs publish their templates using the Service Advertisement mechanism, as well as the coordinates of the SLAM (<<publish/subscribe advertisements>> interaction).

5. SLAMs of the Service Agg. receive and store 3rd party templates. The Provider ID, that indicates the endpoint for negotiating a specific SLAT is propagated together with the SLAT, and it is stored in the SLA Template Registry as metadata. It is mandatory to include this endpoint when the SLAT is registered.

Following this approach, a Service Provider would only be aware of the new templates being advertised. That is, a new actor in the marketplace (a new company) cannot access previous templates of other Service Providers. Similarly, if the Service Agg. fails to listen to the bus during a certain period, it will miss the advertisements publish during that period. In order to avoid inconsistencies, two alternative solutions can be applied:

- Service Providers submit advertisements periodically -- just like it happens on TV and radio.
- The Service Agg. can introduce itself sending a message to the bus requesting specific service types, and the 3rd parties will publish again the advertisements.

When the 3rd party wants to cancel one of the advertized templates, it sends a cancellation message to the bus. It should be noted that the service discovery mechanism described here is just an extension of the communication of new templates between SLAMs of the same framework instance in a monoprovider environment. That is, the framework itself provides mechanisms for the publication of templates, that can be used either internally between SLA Managers of the same provider, either between SLAMs of different framework instances.

**Framework components involved:**

- Business, software and infrastructure SLAManagers, in particular the following sub-components:
  - SLA Registry
  - Service Advertisement Bus
  - SyntaxCoverter, which implements the interface to receive the publishing policies.

**Interactions involved:**

- <<control/track>>
- <<publish/subscribe advertisements>>

### 9.2.3 Product offering

This scenario covers all activities that need to be performed by a Service Aggregator or Service Broker to prepare an actual offering. This might include
issues such as creating needed information in data stores, registration of components, the design of the product and its publication in the product catalog, making it accessible to possible customers.

As a result of the previous scenario, the templates describing all the available services, both own and 3rd party services, are stored in the SLAT registry of the Business SLAManager. Service Aggregator chooses one or more of these SLATs and builds a new product on top of them. To this end, and based on the individual templates, the SLAT of the final product must be defined, containing the description of the service, prices, guarantee terms and also penalties to be applied in case of malfunctioning of the service.

Also at this stage, the Service Aggregator defines the business rules that apply to the final service. For instance, margins for the negotiation process, different offers attached to the product or promotions.

Given that all the templates are stored in the SLAT registry, the above described steps are not different to the ones in the monoprovider case. That is, the SLA@SOI framework provides an homogeneous way to compose and publish products of the same or different providers. Further information about the process of publication can be found in deliverable D.A2a.

**Framework components involved:**
- Business Manager
- Business SLA Manager

**Interactions involved:**
- <<negotiate/query/coordinate>>

### 9.2.4 Product discovery

Prior to the purchase of products, the customer needs to access the information about the available products. As explained in the previous section, the information about products is stored in the Product Catalogue in the Business Manager, and the description of products and details about the offered conditions are included in the corresponding SLA Templates. This is the same situation than in the monoprovider case, and therefore the interaction <<query_product_catalog>> can be also used in a multiprovider scenario to search for products.

This interaction is divided into two: <<getProducts>>, to retrieve the product information, and <<getTemplates>>, to obtain the business SLA templates corresponding to a given product.

Further information about the product discovery can be found in deliverable D.A2a.

**Framework components involved:**
- Business Manager
- Business SLA Manager

**Interactions involved:**
- <<query_product_catalog>>
9.2.5 Negotiation

Once the 3rd party templates are discovered, negotiation can take place using the mechanisms defined for the monoprovieder case. The SLA@SOI framework implements the <<negotiate>> interaction that bundles together the different interactions needed for the negotiation of SLAs. In the monodomain case, when only one framework object is instantiated, there are two conceptually different types of negotiation: between the external customer and the provider (namely, the Business SLA Manager), and between SLAMs of the same framework instance. In the latter case, one of the SLAMs plays the role of provider, being the other one the customer. This allows the usage of the <<negotiate>> interaction to address both types of negotiation process.

In a multidomain scenario, a third type of negotiation should be considered: the one between the different providers. At the end, this translates into a negotiation process between SLAMs of different frameworks. The architecture states that a SLAM can initiate negotiation with any other SLAM, be it internal to the framework or external. In case of the external framework instance, then it must interact with the business level of that framework instance. In any case, the <<negotiate>> interaction is used for negotiation purposes.

For the use cases that means that also a use case can be implemented from a business-centric perspective, where the negotiation only takes place at the business level (that is, involving only Business SLA Managers), as required by B5. As an example, the sequence of operations involved in a negotiation process between the Service Aggregator and 3rd parties SP A and SP B is described here:

1. Business SLAM of Service Aggregator invokes <<negotiate:negotiate>> interaction on the Business SLAM of SP A.
2. Business SLAM of SP A launches the internal negotiation process, invoking the <<negotiate:negotiate>> interaction on its software SLAM, which at the end invokes the <<negotiate:negotiate>> interaction of the infrastructure SLAM.
3. In a similar way, Business SLAM of Service Aggregator invokes <<negotiate:negotiate>> interaction on the Business SLAM of SP B.
4. A negotiation process internal to SP B framework is then triggered, involving its software and infrastructure SLAMs.

Therefore, the negotiation procedures provided by the SLA@SOI framework are valid in a scenario with more than one provider.

Although already considered in the monoprovieder case, security considerations are mentioned here due to their special relevance when different providers are involved. As explained above, the negotiation takes place either between an external customer and a SLAM, or between SLAMs (either of the same framework instance or of different ones). Regardless the existence of one or more parties, the id of the provider SLAM is stored as part of the SLA Templates of the offered services, and it will be used to initiate the negotiation process. That raises the question whether the provider id set on the SLAT can be trusted or not. This issue can be resolved at negotiation time using x.509 certificates, or other methods for authentication, so it can always be confirmed with certainty that the 3rd party is who it claims to be, since that will be guaranteed by the trusted Certification Authorities.

Framework components involved:
9.2.6 Provisioning

Once the customer and the Service Aggregator have reached an agreement, they sign an SLA establishing the exact conditions under which the service is delivered. The next step is the provisioning of the involved services.

Unlike the monoprovider case, where the provisioning of the services only involves the infrastructure of one provider, in the multiprovider scenario the provisioning must run across the different providers involved. To this extend, it is worth mentioning that the Service Aggregator acts as a service provider to the customer, and as a customer for the 3rd party service providers. In particular, when the customer requests the provisioning of the aggregated service, the Service Aggregator will request the provisioning of the atomic services to the corresponding parties.

The SLA@SOI framework implements the substereotype <<provision>> of the <<coordinate>> interaction to trigger the provisioning, and it should be applicable to the multiprovider scenario. In this case, the sequence of the operations is the following:

1. Customer invokes <<coordinate:provision>> interface to the Business SLA Manager of the Service Aggregator.
2. The Business SLA Manager of the Service Aggregator triggers the provisioning of 3rd party services using the <<coordinate:provision>> interface to the SLAMs in the 3rd party services.
3. The Business SLA Manager of the Service Aggregator triggers the provisioning of its own services using the <<coordinate:provision>> interface of the underlying SLAMs of its framework.

It should be noted that the procedure is the same than in the monoprovider scenario, when dealing with a composite product. In that case, the provisioning should be triggered for each one of the atomic services, and using the same interactions. The only difference is that the recipient of the provisioning requests is a SLAM inside the same framework, while in the multiprovider scenario it is an SLAM inside the framework of an external provider. But, since the endpoint is stored in the SLAT itself, the procedure is the same regardless the scenario involves one or more providers: the provisioning is triggered to the endpoint indicated in the template.

Framework components involved:

- Business SLA Manager
- Software SLA Manager
- Infrastructure SLA Manager

Interactions involved:

- <<negotiate/query/coordinate>>
9.2.7 Run-time Operations

Once the services have been successfully provisioned and the whole product has been activated, the customer can start using the purchased product. The GuaranteeTerms embedded in the signed SLA establish the conditions of the normal functioning of the product. When the product is composed of one or more Third Party services, the SLA(s) between the Service Aggregator and the Third Party Provider(s) also contain GuaranteeTerms that govern the normal operation of each of the services. The GuaranteeTerms can contain penalties to be applied in case of malfunctioning of an atomic service or the whole product.

This section describes the mechanisms that play an important role during the product usage: the monitoring of the service, the adjustment and penalty application and the retrieval of information through automatic reports.

Monitoring

The Service Aggregator must ensure that the product is delivered to the customer under the conditions agreed in the Guarantee Terms of SLA. Therefore, the aggregator monitors the quality parameters of the product being used by the end-customer. The measured parameters are then compared with the values of the GuaranteeTerms of the signed SLA, in order to detect SLA violations.

The monitoring process depends on the kind of services involved:

1. Services provided by the Service Aggregator. Monitoring takes place at infrastructure, software and business layers of the SP Agg., each of which has their own Monitoring and Adjustment modules. As soon as an SLA violation is detected, it is communicated to the upper layer, and the appropriate actions take place in order to restore the normal functioning of the system.

2. Services provided by Third Parties. In this case, the Service Aggregator does not have access to the external low level infrastructure nor to Third Parties service management environment. The Service Agg. must monitor the service in the customer’s endpoint.

The results of the monitoring process are stored in an historical database for analysis purposes

Framework components involved:

- Business SLA Manager
- Software SLA Manager
- Infrastructure SLA Manager
- Software Service Manager
- Infrastructure Service Manager
- Manageability Agents
- Low Level Monitoring System
- Monitoring Event Channel

Interactions involved:

- <<publish_event>>
- <<subscribe_to_event>>
**Adjustment and Penalties**

As explained in the previous section, all the layers involved in the product, along with third party domain, provide monitoring information. If the monitoring information states that a given condition of the SLA has been breached, the adjustment component must identify the cause of that violation and take some corrective actions. The Service Aggregator does not have access to the third party infrastructure, therefore only really corrective actions (e.g. reprovisioning) can be taken on failure of one of its own services. When dealing with the malfunctioning of a 3rd party service, actions are taken at business level and include procedures as renegotiation of the agreement, termination, change to an analogue service offered by another provider, or application of penalties.

It is worth mentioning the importance of the Penalty Management from the Service Aggregator point of view. Penalties apply not only in the relationship with the customer, but also with the Third Party providers. From a business perspective, the Service Aggregator will develop policies aiming to maximize the benefits, and minimize the penalty impact. Therefore, one of the main aspects to be taken into account when implementing the business policies is the propagation of the penalty to be paid to a Customer, to penalties to the Third Party Providers delivering the atomic services of the aggregated product.

1. In case a SLA violation is detected, an automated process of adjustment takes place in order to restore the normal functioning of the service. The adjustment process is different depending on where the problem is located:
   a. Malfunctioning of a service owned by the Service Aggregator. In this case, the Adjustment modules of the involved layers (Infrastructure, Software and/or Business) automatically take the appropriate corrective actions, through the `<<manage_software_services>>` or `<<manage_infrastructure_services>>` interactions.
   b. Malfunctioning of a Third-Party service. In this case, the Agg. has no means to apply the needed adjustment in the service or infrastructure layer of the Third Party. The only possible adjustment comes from the modification of the internals of the product delivered to the Customer:
      - SP Agg. selects another Third Party (SP C) Provider offering a service with the same functionality, and checks that the SLA of this new service is compatible with the SLA between the Customer and the Aggregator.
      - Service Aggregator triggers the provisioning of the new service to the Third Party Provider C (``coordinate:provision`` interaction).
      - Once the provision is finished, Aggregator modifies the aggregation: the new service is used instead the malfunctioning one.
      - Service Aggregator notifies the original Third Party owner of the malfunctioning service the finalization of the usage of that service (``<negotiate>``)
   c. In case problems arise both in services owned by the Aggregator and Third Party-services, both a) and b) must be applied.
   d. Malfunctioning of the Composition. If the atomic services are running with the correct QoS, the problem may be located in the
way the composition has been done by the Service Aggregator. In that case, the composition should be redefined and redeployed (<<manage_software_services>>).

2. In case a GuaranteeTerm with an attached penalty is violated, the defined penalty must be applied. The definition of a GuaranteeTerm includes the party (Service Provide or Customer) that is obliged to fulfil the given term. Therefore, a penalty can affect the Customer, the Service Aggregator or the Third Party Provider:
   a. SLA between Service Aggregator and a Third Party Provider. If a GuaranteeTerm of this SLA is violated, the penalty must be applied to the Third Party (if the term has ServiceProvider as the obliged party) or to the Service Aggregator (if the obliged party is the Customer).
   b. SLA between Aggregator and Customer. If a GuaranteeTerm of this SLA is violated, the penalty must be applied to the Service Aggregator (if the term has ServiceProvider as the obliged party) or to the end Customer (if the obliged party in the GuaranteeTerm definition is the Customer).
   c. When a violation in the SLA between the Aggregator and the Customer is associated to a violation in the SLA signed with a Third Party, both a) and b) must be applied.

3. In case a penalty is applied, the corresponding report must be automatically sent to the other party:
   a. SLA between Service Aggregator and Customer. The Customer Manager component informs the end Customer of the GuaranteeTerm that has been violated, as well as the penalty application.
   b. SLA between Service Aggregator and Third Party Provider. The Service Aggregator informs the Third Party of the GuaranteeTerm that has been violated, and the associated consequences (application of a penalty). The 3rd party can also be monitoring its own services, and analyzing its observations to confirm that the violations notified by the Service Aggregator have indeed been produced. With different monitoring systems own by different parties, a discrepancy between measurements is possible. Due to the economical implications (penalties) of the violations, this becomes an important issue in a multidomain environment. For this reason, workpackage A2 has developed a non-repudiation evidences mechanism, that guarantees the trustworthiness of the components involved in the monitoring of the SLAs.

Framework components involved:
- Business Manager
- Business SLA Manager
- Software SLA Manager
- Infrastructure SLA Manager

Interactions involved:
- <<manage_software_services>>
- <<manage_infrastructure_services>>
- <<control/track>>
**Reporting**

One of the objectives of SLA@SOI project is maintain the quality of the products offered to the customer according to the level of quality agreed inside the SLA. In order to transparently fulfil this requirement, a procedure must be implemented so the customer can check the behaviour of the products contracted once the services are up and running.

The only difference with respect to the monoprovider case, is that when more than one provider is involved some data may need to be collected from the third parties. In any case, the interaction to collect information, either from the lower levels, either from a third party, is the <<control/track>> interaction.

**Framework components involved:**
- Business Manager
- Business SLA Manager
- Software SLA Manager
- Infrastructure SLA Manager
- Low Level Monitoring System

**Interactions involved:**
- <<customer_relations:report>>
- <<control/track>>

9.2.8 **Decommissioning**

The conditions and processes to terminate an agreement are specified in the SLA. The SLA establishes as well the expiration date indicating the period in which the agreement is valid. When this date is reached, the normal termination procedure takes place, which is in charge of freeing all the allocated resources for that given service, and updating the corresponding registries. When Third Party services are involved, the termination procedure must trigger the finalization of the service to the Third Party Provider.

Apart from the normal termination procedure, the termination process could be launched following a bilateral or a unilateral decision. In any case, the SLA should specify the foreseen reasons to end an agreement, either by the customer or by the service provider and the methods and periods to send the corresponding notification, as well as possible fees or expenses due to services being delivered up to the effective date of termination.

A concrete example of a special termination procedure is a GuaranteeTerm of the SLA specifying that, in case of violation, the agreement must be automatically ended, instead of the Service Provider trying to recover the normal situation.

The steps that fulfil the termination process are the following:

1. Notification of the end of the agreement. Two different cases may be distinguished:
   a. In case of a normal termination procedure, and previously to the expiration date, the Service Aggregator notifies the Customer the end of the service delivery. This notification can include an offer to prolong or renegotiate the agreement.
b. In case that one of the parties wishes to finalize the agreement before the expiration date, it must send a notification to the other party, in the format specified in the SLA.

c. In case of an automatic termination procedure, the Service Aggregator notifies the Customer the end of the agreement, specifying the reasons that lead to that situation.

2. The Service Aggregator launches the finalization of the services that compound the final product. The concrete process depends on the kind of services involved, but it is always realized through the \textlangle
\textnegotiate/query/coordinate\textrangle interaction:

a. Own services: SP Agg. commands the service finalization to the Business Layer, which takes care of the finalization in the lower layers. The software and infrastructure resources are released, and the registries updated. Automatic monitoring and adjustment processes are instructed to not manage the terminated service.

b. Third Party services: Service Aggregator commands the service finalization to the Third Party Provider involved. The Third Party frees its resources and takes the corresponding management actions.

c. Composite services: Service Aggregator frees the aggregation, and launches steps a) or b) for each one of the atomic services. Automatic monitoring and adjustment processes are instructed to not manage the terminated aggregation.

\textbf{Framework components involved:}

- Business SLA Manager
- Software SLA Manager
- Infrastructure SLA Manager
- Software Service Manager
- Infrastructure Service Manager

\textbf{Interactions involved:}

- \textlangle
\textnegotiate/query/coordinate\textrangle

\section*{9.3 Framework extensions and implementation}

As explained in the previous section, the overall architecture of the SLA@SOI framework was designed from the beginning to be able to be used in a multiprovider environment. This assertion is mainly supported by the following facts:

- A multiprovider environment is a playground where providers can play a double facet, being either customer to other providers either provider to end-user customers or to other service providers.

- The negotiation and provisioning interactions are realized using the endpoint reference stored as part of the SLA Template. This applies both
to internal templates, where the endpoint corresponds to another SLAManager, as well to external templates, where the endpoint corresponds to another SLAM of an external framework instance.

- The mechanism for SLAT publication, through an event advertisement bus, is the same regardless the SLAT is to be used by other SLAManagers inside the same framework instance or by an external framework.

This symmetry between internal and external SLAMs makes it possible that the framework can be easily used as it is in a multidomain marketplace. Following the analysis of the scenarios presented in the previous section, the main impact of the inclusion of external providers would be at the business layer. To that end, the database model designed as part of A2 workpackage (see deliverable D.A2a) has been extended so to include a registry of the 3rd parties. The symmetry is kept using a generic party element, and its associated party-role. A party is defined as any entity interacting in the marketplace, being the party-role the framework owner (the provider that owns the instance of the framework), a customer or a provider. Either an organization or an individual could be parties in this model. In the later case, their credentials (username and password) will also be stored in the database. Since this has been included as part of the A2 database model, further details can be found in the A2 deliverable D.A2a.

The feature is extended with the addition of the <<provider_relations>> interaction, which allows the registration of 3rd parties into the framework, the modification of the registration information and authentication prior to the negotiation interaction. The Provider Relationship Management Component has been implemented in the Business Manager, storing all the information concerning 3rd party providers, and offering webservice interfaces for the registration and authentication. The symmetry in the database model explains the similarity between this component and the Customer Relationship Management Component.

Another important feature to be added in the multiprovider case is the possibility to detect that a given service of a 3rd party is malfunctioning, and substitute that for another 3rd party service, modifying the aggregated or composed service. To this extend, specific rules have been added into the Business Adjustment Component. The reason to allocate these rules at the top-level business component is that, regardless of the number of SLAMs a framework can instantiate, the uniqueness of the Business Manager makes it a single point receiving reports of all the problems in the underlying layers (through the <<control/track>> interaction), and therefore able to access all the information about the performance of a given service or of a given 3rd party provider. The decision about when to move to another 3rd party provider can be based in simple preconditions, as a persistent high number of violations, or in a deeper economical analysis, calculating the difference between the incoming due to that service and the penalties paid due to the violations. Once the decision is taken, the procedure to trigger is the one described in previous sections. More technical details are included, for consistency, in the A2 deliverable D.A2a.

9.4 Future Work

Future work in Task A3.6 will focus on three main aspects: on one hand, collecting a detailed feedback from the usecases that are using the multiprovider functionality (mainly B5 and B6), extending the implementation following the possible new requirements. On the other hand, the current analysis will be extended to cope with the multidomain environment.
10 Conclusion

This document presented the work carried out in the work package A3 in Y2. Broadly speaking, A3 addressed the SLA-aware service management and therefore, the focus of the activities was on the manageability aspect of the software/service layer of the IT stack. A3 designed the SoftwareServiceManager building block of the SLA@SOI framework. A default implementation of the SoftwareServiceManager was provided with extensibility mechanisms to carry out domain specific customization for various use cases within SLA@SOI. In addition to this, various tasks focused on other aspects of the SLA@SOI framework.

A number of topics were addressed in Y2 in the work package A3. Regarding modelling, task A3.1 focused on enhanced SOA modelling for enabling business continuity management support whereas task A3 whereas task A3.2 focused on landscape metamodelling and a default repository implementation for storing and query the landscape models. Dynamic service discovery and binding was the focus of work in task A3.3 and a Dynamic Orchestration Engine was implemented which is capable of performing dynamic service bindings to ensure SLA guarantees. Task A3.4 addressed the unified SOA manageability issues and designed a unified interface to manage various SOA landscape artefacts.

Advanced SOA management focused on service & process monitoring and service adjustment. Architecture and implementation details of the Monitoring Manager was described which provided functionality to translate SLA guarantee terms into low level monitoring rules to be observed during runtime for detecting potential SLA violations. Based on this dynamic service adjustment needs to be performed to bring software and service elements into compliance with the established SLAs. Details of this autonomic service adjustment engine were presented as well. Finally, multiprovider/multidomain service management was addressed in task A3.6 which augments the SLA@SOI framework to be applied to multiprovider scenarios.
11 References

[1] SLA-aware Service Management M12 Deliverable, 

   http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[3] ActiveVOS, 
   http://www.activevos.com/community-open-source.php


   https://javacc.dev.java.net/


[16] Project ANA Web Site: http://www.ana-project.org/


[18] BIONETS Web Site: http://www.bionets.eu/


[22] Project HAGGLE Web Site: http://www.haggleproject.org


[27] TODO


Glossary

The following list shows the most important entries of the SLA@SOI glossary. Note that terms that are specific for the current document and not part of the overall project wide glossary are marked with an asterix *.

Agreement Initiator

An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.

Agreement Offer

An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.

Agreement Responder

The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.

Agreement Template

An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.

Agreement Term

Agreement terms define the content of a service level agreement.

Business Service

A business service is exposed/invoked via at least some non IT elements.

Business Manager

A specialization of service provider: person that defines the SLATs of products and joins available services in a product.

External Service

External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.

Framework Administrator

A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.

Guarantee Term

Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.

Hybrid Service

A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).

Infrastructure Manager

A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.
Infrastructure Provider: A specific kind of service provider that focuses on the provisioning of infrastructure services.

Infrastructure Service: An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.

Internal Service: Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.

IT Service: An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.

Offered Service: An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.

Operation Level Agreements: A specification of the conditions under which an internal service or a component is to be used by its “customer”.

Service: A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks.

Service Concreteness: The stage a service reaches over time from a fully abstract type to actually instantiated.

Service Consumer: Person(s) who actually consume/use the provided services. Typically they belong to the service customer.

Service Customer: Someone (person or group) who orders/buys services and defines and agrees the service level targets.

Service Description Term: Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.

Service Exposure: Services can be exposed either internally (within the same administrative domain) or externally.

Service Implementation: A service implementation is a possible concrete realization of a given service type.

Service Instance: A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.

Service Interface Type: Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface.

Service Level Consequence: An action that takes place in the event that a service level objective is not met.

Service Level Agreement: An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may
specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or availability. Entities can dynamically establish and manage agreements via Web service interfaces.

**Service Level Objective**
Service Level Objective represents the quality of service aspect of the agreement. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.

**Service Provider**
An organization supplying services to one or more internal customers or external customers.

**SLA Manager**
A specialization of service provider: person/system that is responsible for managing SLATs and SLA relationships.

**Software Designer**
A specialization of software provider: person that designs/develops the architecture and components of a specific SLA based application.

**Software Manager**
A specialization of service provider: person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.

**Software Provider**
An organization producing software components which might be used by a service provider to assemble actual services.

**Software Service**
A software service is a specific IT service which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.

**Software Component**
Software components are the entities produced at design-time by a software provider.

**Service Type**
A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.
Appendix A: Abbreviations

AOP  Aspect Oriented Programming
BM   Business Manager
B-SLAM Business SLA Manager
EMF  Eclipse Modelling Framework
IE   Interaction Event
Infr-SLAM Infrastructure SLA Manager
Infr-SM Infrastructure Service Manager
IoC  Inversion of Control
LLMS Low Level Monitoring System
MA   Manageability Agent
MRE  Monitoring Result Event
NFP  Non-functional property
ORC  Open Reference Case
OVF  Open Virtualization Format
QoS  Quality of Service
POJO Plain Old Java Objects
SE   Service Evaluation
SLA  Service Level Agreement
SLAM SLA Manager
SLAT Service Level Agreement Template
SLMS SLA Level Monitoring System
SM   Service Manager
SOA  Service Oriented Architecture
SW-SLAM Software SLA Manager
SW-SM Software Service Manager
TOGAF The Open Group Architecture Framework

Appendix B: Admin Interface

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://docs.sla@soi/wsdl/enginebpeladmin/enginebpeladmin.wsdl"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:implProject="http://docs.sla@soi/wsdl/enginebpeladmin/enginebpeladmin.wsdl"
xmlns:tnsProject="http://docs.sla@soi/schema/enginebpeladmin/enginebpeladmin.xsd"
xmlns:slasoi="http://www.slaatsoi.eu/slamodel"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema elementFormDefault="qualified" targetNamespace="http://docs.sla@soi/schema/enginebpeladmin/enginebpeladmin.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="addPartnerRoleAbstractBindingRuleInput" type="tnsProject:EnsAddAbstractBindingRuleRequestType"/>
      <complexType name="EnsAddAbstractBindingRuleRequestType">
        <sequence>
          <element name="processID" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </wsdl:types>
</wsdl:definitions>
```
<element name="partnerRoleName" type="xsd:string"/>
<element name="bindingRule" type="tnsProject:EnsBindingRule"/>
</sequence>
</complexType>
<complexType name="EnsBindingRule">
<sequence>
<element name="condition" nillable="true" type="apachesoap:Map"/>
<element name="wsdlAddress" type="xsd:string"/>
<element name="serviceName" type="xsd:string"/>
<element name="portName" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="EnsAbstractBindingRule">
<sequence>
<element name="condition" type="apachesoap:Map"/>
<element ref="slasoi:SLATemplate"/>
<element name="bpelVariable" type="xsd:string"/>
<element name="isMultiple" type="xsd:boolean"/>
</sequence>
</complexType>
<element name="removePartnerRoleRuleInput" type="tnsProject:EnsRemoveRuleRequestType"/>
<complexType name="EnsRemoveBindingRuleRequestType">
<sequence>
<element name="processID" type="xsd:string"/>
<element name="partnerRoleName" type="xsd:string"/>
<element name="ruleCondition" nillable="true" type="apachesoap:Map"/>
</sequence>
</complexType>
<element name="getPartnerRoleBindingRulesInput" type="tnsProject:EnsGetBindingRulesRequestType"/>
<element name="getPartnerRoleAbstractBindingRulesInput" type="tnsProject:EnsGetRulesRequestType"/>
<complexType name="EnsGetRulesRequestType">
<sequence>
<element name="processID" type="xsd:string"/>
<element name="partnerRoleName" type="xsd:string"/>
</sequence>
</complexType>
<element name="getPartnerRoleAbstractBindingRulesOutput" type="tnsProject:EnsAbstractBindingRulesListType"/>
<complexType name="EnsAbstractBindingRulesListType">
<sequence>
<element name="totalRulesCount" type="xsd:int"/>
<element name="empty" type="xsd:boolean"/>
<element name="abstractBindingRules" nillable="true" type="implProject:ArrayOf_tnsProject_EnsAbstractBindingRule"/>
</sequence>
</complexType>
<element name="getPartnerRoleBindingRulesOutput" type="tnsProject:EnsBindingRulesListType"/>
<complexType name="EnsBindingRulesListType">
<sequence>
<element name="totalRulesCount" type="xsd:int"/>
<element name="empty" type="xsd:boolean"/>
<element name="bindingRules" nillable="true" type="implProject:ArrayOf_tnsProject_EnsBindingRule"/>
</sequence>
</complexType>
</schema>
<schema elementFormDefault="qualified" targetNamespace="http://docs.active-endpoints.wsdl/activebpeladmin/2007/01/activebpeladmin.wsdl" xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://types.admin.bpel.axis.rt.activebpel.org"/>
<import namespace="http://xml.apache.org/xml-soap"/>
<complexType name="ArrayOf_tnsProject_EnsBindingRule">
<sequence>
<element maxOccurs="unbounded" minOccurs="0" name="item"/>
<complexType name="ArrayOf_tnsProject_GetPartnerRoleAbstractBindingRulesInput">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:EnsAbstractBindingRule"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_GetPartnerRoleBindingRulesInput">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:EnsBindingRule"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectGetPartnerRoleAbstractBindingRulesRequest">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:GetPartnerRoleAbstractBindingRulesInput"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectGetPartnerRoleBindingRulesRequest">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:GetPartnerRoleBindingRulesInput"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectGetPartnerRoleAbstractBindingRulesResponse">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:GetPartnerRoleAbstractBindingRulesResponse"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectGetPartnerRoleBindingRulesResponse">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:GetPartnerRoleBindingRulesResponse"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectAddPartnerRoleBindingRuleRequest">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:AddPartnerRoleBindingRuleRequest"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectAddPartnerRoleBindingRuleResponse">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:AddPartnerRoleBindingRuleResponse"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectRemovePartnerRoleRuleRequest">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:RemovePartnerRoleRuleRequest"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectRemovePartnerRoleRuleResponse">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:RemovePartnerRoleRuleResponse"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectAddPartnerRoleAbstractBindingRuleRequest">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:AddPartnerRoleAbstractBindingRuleRequest"/>
  </sequence>
</complexType>

<complexType name="ArrayOf_tnsProject_ImplementProjectAddPartnerRoleAbstractBindingRuleResponse">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="0" name="Item" type="tnsProject:AddPartnerRoleAbstractBindingRuleResponse"/>
  </sequence>
</complexType>
Appendix C: PDD example

```xml
<wsdl:operation>
  <wsdl:input message="implProj:project:GetPartnerRoleAbstractBindingRulesRequest" name="GetPartnerRoleAbstractBindingRulesRequest"/>
</wsdl:operation>
</wsdl:portType>
</wsdl:definitions>

Appendix C: PDD example

```xml
<pdd:process>
  ...
  <pdd:partnerLink name="PaymentServicePLT">
    <pdd:myRole allowedRoles="" binding="MSG" service="PaymentService"/>
  </pdd:partnerLink>
  <pdd:partnerLink name="cardValidation">
    <pdd:partnerRole endpointReference="static">
      invokeHandler="java:it.eng.binding.impl.EngInvokeHandler"
      <wsa:EndpointReference
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:ns5="http://cardValidationService.webservices.org/"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
        <wsa:Address>
          http://localhost:8081/Store1/CardValidationService?wsdl
        </wsa:Address>
        <wsa:Metadata>
          <wsa:ServiceName PortName="CardValidationWsIfPort">
            ns5:CardValidationWsService
          </wsa:ServiceName>
        </wsa:Metadata>
      </wsa:EndpointReference>
    </pdd:partnerRole>
  </pdd:partnerLink>
</pdd:partnerLinks>
</pdd:process>
```