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

Deliverable DA3.a
SLA-Aware Service Management

Keywords:
Service Management, Service Monitoring, Service composition

Due date of deliverable: 31st May, 2009
Actual submission to EC date:

Start date of project: 1st June 2008
Duration: 36 months

Lead contractor for this deliverable: SAP
Revision: 1.0 (24th June, 2009)

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
</table>

Page 1 of 116
Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>SAP AG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Gregor Berginc (XLAB)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Mark Evenson (TA)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Joe Butler (Intel)</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>22nd April, 2009</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>10th May, 2009</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>1st June, 2009</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>8th June, 2009</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>19th June, 2009</td>
</tr>
</tbody>
</table>

Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP</td>
<td>Hui Li, Tariq Ellahi</td>
</tr>
<tr>
<td>Engineering</td>
<td>Paolo Zampognaro</td>
</tr>
<tr>
<td>CITY</td>
<td>Marco Commuzzi</td>
</tr>
<tr>
<td>PMI</td>
<td>Sam Guinea</td>
</tr>
<tr>
<td>FBK</td>
<td>Annapaola Marconi, Michele Trainotti</td>
</tr>
<tr>
<td>FZI</td>
<td>Christof Momm</td>
</tr>
<tr>
<td>TID</td>
<td>Beatriz Fuentes, Alfonso Castro</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.

Document History

| Version | Date       | Author | Changes                                                        |
|---------|------------|--------|                                                               |
| 0.1     | 22/04/2009 | WP A3  | First draft ready for internal reviews.                        |
| 0.2     | 15/05/2009 | WP A3  | Changes made according to first reviewer’s comments            |
| 0.3     | 05/06/2009 | WP A3  | Changes made according to second reviewer’s comments            |
| 1.0     | 24/06/2009 | WP A3  | Finalized version after PMT reviews                            |
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 Y1. Since, the overall direction of Y1 activities within SLA@SOI was ORC based adhoc demonstrator, focus of A3 activities was also restricted to the adhoc demonstrator. 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 meta-model 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 focused on dynamic service binding and composition related activities. The scope of this activity was to enable compositions in dynamic and changing environment where service compositions are performed during runtime and bindings are carried out driven by SLAs and quality of service guarantees.

Last but not least, A3’s work focused on runtime monitoring and management of SOA based application landscapes. An event based non-intrusive monitoring approach was adopted within A3 with an objective of monitoring atomic services, composite process as well as composition engines. Additionally, analysis and event correlation was also investigated for SLA violation detection purposes. On the manageability front, A3 pursued the design of a unified manageability infrastructure with special emphasis on unified manageability interface to enable management application to retrieve monitoring information as well as for control the elements being managed. Additionally, efforts were conducted to enable autonomic behaviour to the manageability infrastructure.
# Table of Contents

1 Introduction.................................................................................................................. 7
2 Modelling...................................................................................................................... 8
2.1 Service Component Modelling.................................................................................. 8
  2.1.1 State Of The Art ............................................................................................. 9
  2.1.2 SOA Component Assembly Model .................................................................. 10
  2.1.3 SOA Policy Model ......................................................................................... 12
  2.1.4 SOA Component Behaviour Model .............................................................. 14
  2.1.5 Summary and Open Issues ............................................................................ 14
2.2 Landscape Modelling ............................................................................................... 15
  2.2.1 Landscape Meta-model.................................................................................... 16
  2.2.2 Software Landscape Model Examples ......................................................... 18
3 Dynamic Service Discovery & Composition............................................................... 20
  3.1 Definitions & approaches...................................................................................... 20
  3.2 Problem Statement ............................................................................................... 22
  3.3 Solution.................................................................................................................. 22
    3.3.1 Methodology ............................................................................................... 22
    3.3.2 Requirements & Architecture ..................................................................... 26
4 Service Monitoring ..................................................................................................... 32
  4.1 Service Instrumentation ......................................................................................... 32
    4.1.1 Process Instrumentation .............................................................................. 33
    4.1.2 Service Instrumentation .............................................................................. 36
  4.2 Service Monitoring Architecture ........................................................................... 38
    4.2.1 Atomic Service Monitoring ......................................................................... 39
    4.2.2 Process Monitoring ...................................................................................... 47
5 Service Manageability................................................................................................. 55
  5.1 Service Manageability Infrastructure ..................................................................... 55
    5.1.1 Foundations................................................................................................. 57
    5.1.2 Requirements............................................................................................... 62
    5.1.3 State of the Art ............................................................................................ 63
    5.1.4 Standard-independent Design of the Manageability Infrastructure .......... 68
    5.1.5 Adding Support for Management Standards ............................................. 86
    5.1.6 Implementation ........................................................................................... 93
    5.1.7 Using the Manageability Interface within ORC / Ad Hoc Demonstrator .... 95
  5.2 Autonomic Service Management .......................................................................... 100
    5.2.1 Principles of Autonomic Systems ............................................................... 100
    5.2.2 Autonomic-based Adjustment ..................................................................... 102
    5.2.3 Initial Architecture of Adjustment module ................................................. 105
6 Conclusions................................................................................................................ 106
  6.1 Summary............................................................................................................... 106
  6.2 Outlook on Future Work......................................................................................... 107
7 References.................................................................................................................. 108
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The SOA Component Model based on SCA Assembly</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>The SOA Policy Model based on SCA Policy Framework</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>the SOA Component Behavior Model based on Palladio’s service effect specification</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Software Landscape Meta-Model</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>software landscape example 1. Appliance with one virtual machine.</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>software Landscape Example 2. Appliance packaged with three virtual machines</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Characteristics of main approaches to dynamic composition</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>more than one deployed process for the same offered service</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>one deployed process for each offered service: results of negotiation are binding information and not different processes</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>Manual dynamic binding vs automatic dynamic binding</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>Deploying an abstract process</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>SLAT discovery</td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>SLAT selection capabilities</td>
<td>27</td>
</tr>
<tr>
<td>14</td>
<td>SLA Negotiation</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>dynamic automatic composition: discovery, (local) selection, negotiation</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>dynamic manual composition</td>
<td>30</td>
</tr>
<tr>
<td>17</td>
<td>conceptual architecture</td>
<td>31</td>
</tr>
<tr>
<td>18</td>
<td>The four events created during a standard process-service interaction</td>
<td>33</td>
</tr>
<tr>
<td>19</td>
<td>The Process Instrumentation Infrastructure</td>
<td>34</td>
</tr>
<tr>
<td>20</td>
<td>Example of a Service Invocation Event specification</td>
<td>35</td>
</tr>
<tr>
<td>21</td>
<td>Example of Process Status Event specification</td>
<td>36</td>
</tr>
<tr>
<td>22</td>
<td>Overview of Service Instrumentation in the AXIS container</td>
<td>37</td>
</tr>
<tr>
<td>23</td>
<td>Example of Service Invocation Event specification</td>
<td>38</td>
</tr>
<tr>
<td>24</td>
<td>Overall architecture of SLA@SOI SLA Monitoring Framework</td>
<td>39</td>
</tr>
<tr>
<td>25</td>
<td>Example of SLA Guarantee Term for SLA@SOI Ad-Hoc demonstrator</td>
<td>43</td>
</tr>
<tr>
<td>26</td>
<td>The architecture of monitor engine (stand-alone application)</td>
<td>45</td>
</tr>
<tr>
<td>27</td>
<td>Adapting CITY Monitor to the SLA@SOI SLA Monitoring framework.</td>
<td>47</td>
</tr>
<tr>
<td>28</td>
<td>Process Monitoring Engine / Framework</td>
<td>49</td>
</tr>
<tr>
<td>29</td>
<td>Monitor and Property Java Interfaces</td>
<td>50</td>
</tr>
<tr>
<td>30</td>
<td>Monitor Interactions</td>
<td>52</td>
</tr>
<tr>
<td>31</td>
<td>Pseudo code instance monitor</td>
<td>52</td>
</tr>
<tr>
<td>32</td>
<td>SLA/Service Stack</td>
<td>56</td>
</tr>
<tr>
<td>33</td>
<td>Unified Management View</td>
<td>57</td>
</tr>
<tr>
<td>34</td>
<td>Models of a Management Architecture</td>
<td>58</td>
</tr>
<tr>
<td>35</td>
<td>Mapping of Management Architecture Models to Models of Manageability Infrastructure</td>
<td>59</td>
</tr>
<tr>
<td>36</td>
<td>General Architecture of a Manageable Application System</td>
<td>60</td>
</tr>
<tr>
<td>37</td>
<td>Generalized Approach to Instrumenting Software Components</td>
<td>61</td>
</tr>
<tr>
<td>38</td>
<td>Refined Architecture of a Manageable Application System</td>
<td>61</td>
</tr>
<tr>
<td>39</td>
<td>Managed SCA Composite</td>
<td>68</td>
</tr>
<tr>
<td>40</td>
<td>Manageable SCA Application with one Manager</td>
<td>69</td>
</tr>
<tr>
<td>41</td>
<td>Manageable SCA Application with severnal Manager</td>
<td>69</td>
</tr>
<tr>
<td>42</td>
<td>Metamodel for Manageable SCA Applications</td>
<td>70</td>
</tr>
<tr>
<td>43</td>
<td>Approach to Manageability and Instrumentation Design</td>
<td>71</td>
</tr>
<tr>
<td>44</td>
<td>Basic Structure of Events</td>
<td>72</td>
</tr>
</tbody>
</table>
Figure 45: ServiceOperationCall Events ........................................................ 72
Figure 46: ReferenceOperationCall Events .................................................... 73
Figure 47: Loop Events .............................................................................. 73
Figure 48: Conditional Flow and Variable Events ............................................ 73
Figure 49: Communication between different Sensors .................................... 74
Figure 50: Sample Instrumentation Design ................................................... 75
Figure 51: Sample OperationCall Events ....................................................... 75
Figure 52: Sample VariableData Events ......................................................... 76
Figure 53: Role of the Manageability Data Model ........................................... 77
Figure 54: Granularity of Operations ............................................................. 78
Figure 55: Manageability Data Model ............................................................ 79
Figure 56: Manageability Interface Design ..................................................... 80
Figure 57: Role of Management Agents ........................................................ 82
Figure 58: Internal Design of the Management Agent ................................... 84
Figure 59: Sample Interaction between Instrumentation and Agent ................. 85
Figure 60: Support for Multiple Management Standards ............................... 87
Figure 61: Selected Combination of Standards .............................................. 88
Figure 62: CIM Extension for Manageability Data Model .............................. 89
Figure 63: Extended Manageability Infrastructure Design .............................. 91
Figure 64: Extended Management Agent Design ......................................... 92
Figure 65: Definition Loader Implementation .............................................. 93
Figure 66: Payment Service ........................................................................ 95
Figure 67: Sample SLA for PaymentService .................................................. 96
Figure 68: Scenario 1 ................................................................................ 98
Figure 69: Results ..................................................................................... 98
Figure 70: Scenario 2 ................................................................................ 99
Figure 71: Scenario 3 ................................................................................ 99
Figure 72: Results ................................................................................... 100
Figure 73: MAPE-K Architecture ................................................................ 101
Figure 74: Block diagram for a classical closed control loop ......................... 103
Figure 75: Adjustment Control Loop ............................................................ 104
Figure 76: Simplified Adjustment Architecture .......................................... 105
Figure 77: Event Format XML Schema ........................................................ 115

List of Tables

Table 1 - Built-in operations for specification and computation of service guarantee terms ................................................................. 41
1 Introduction

The IT ecosystem has been undergoing a transformation in the recent years. This transformation is leading the IT ecosystem from a product based landscape to a service based IT landscape. This means that the IT elements which were previously being bought by businesses are now offered by service providers in a service based fashion. For example, infrastructure providers offer infrastructure services aka Infrastructure as a Service (IaaS), software is delivered as software services aka Software as a Service (SaaS). The complete solution used by business and enterprises is hence composed of pieces from various service providers. Although, it is possible for a single provider organization to offer all the services. As businesses adopt this new IT landscape and procure and consume services to power their business operations and processes, it becomes critical for the service providers to continuously improve the services being delivered to the customers. Business continuity is the foremost objective of the organizations and service disruptions can compromise this objective leading to financial loss as well as business reputation and customer base. To effectively cope with this, service customers would like to have some quality of service guarantees from the service providers agreed in the form of service level agreements (SLA). Service providers, on the other hand should continuously strive to ensure that the objectives stipulated in the SLAs must be sustained. An effective, agile and accurate SLA management framework becomes a top most requirement in the upcoming service based economy.

SLA@SOI aims at addressing the SLA management activities from a holistic perspective taking into account the requirements and responsibilities of various stakeholder involved in the service delivery. SLA@SOI focuses on end-to-end scenarios undertaking SLA management across the layers of the IT execution stack. Starting from the top level business aspects to software services and down to virtual and physical infrastructure services. Workpackage A3 specifically focuses on the software services layer of this execution stack. WP A3 tasks aim at designing and developing innovative methodologies to facilitate the software services management related aspects. In this regard, A3 divides the service management issues into the following working tracks all focusing on a specific aspect of service manageability. Together, these three working tracks provides a complete service manageability infrastructure. Detailed description of these track are giving in later sections but briefly, these are described below:

Modelling: In order to build an effective and robust manageability infrastructure, elaborate comprehension of the elements which are being managed can prove a critical factor. In this context, modelling activities focus on two work streams, firstly, modelling the SOA based applications, services and software components to capture the non-functional aspects of the SOA artefacts as well as the specification of the behaviour of the software components. Secondly, modelling attempts to capture the knowledge of the software services related elements and their relationships and dependencies. The aspects captured under this work stream include service / software packaging and deployment to be utilized during the service provisioning phase of the lifecycle. This knowledge provides reasoning capabilities to the manageability infrastructure. Additionally, such knowledge can greatly increase the comprehension of the systems under management.
Service Discovery & Composition: One of the activities within WP A3 is the investigation of dynamic service discovery and composition of services. The task aims at identifying methodologies for integrating dynamic service discovery and composition into the holistic SLA management approach undertaken by SLA@SOI consortium.

Monitoring: In order to deliver the services in compliance to the SLA established, a monitoring apparatus serves as an indispensable element in the manageability infrastructure. In this context, A3 focuses on the monitoring and analysis of the services, either atomic and composite (i.e. processes). The instrumentation of the service execution environment is a paramount element that enables monitoring activities. Hence, A3 addresses the service, process, and execution engine instrumentation for effective data collection for analysis of the compliance of the service execution with the established SLAs. WP A3 follows an event based monitoring approach, hence an event schema is designed by A3. Such event schema is not only used within the WP, but it is also the basis of monitoring in other WPs as well (i.e. Infrastructure monitoring within WP A4).

Manageability: Finally, the last element in the service management is the enforcement activities to ensure SLA compliance at the software service layer. In this regard activities focus on two distinct but related and complementary issues. First a unified and standards based manageability interface for retrieving management information from the SOA based applications. Secondly, other activities focus on adding autonomic behaviour to the service manageability infrastructure. This enables the manageability infrastructure to apply autonomic adjustments to the software services for ensuring SLA compliance.

This deliverable document is organized into the following structure: Section 2 focuses on an elaborate description of the modelling activities. Dynamic service discovery and composition is discussed in section 3. Detailed description of the instrumentation, monitoring and analysis aspects are presented in section 4. Finally, section 5 concentrates on the uniform manageability interface and autonomic service management activities. Sections 6 and 7 conclude the document with a summary and an outlook on future activities.

## 2 Modelling

### 2.1 Service Component Modelling

This section specifies the SOA component model, or equivalently the software component model in the SLA@SOI project. The definition and specification are based on the assembly and policy model in the Service Component Architecture (SCA) [10]. Additionally, we extend the SCA in two ways for enabling design time analysis and prediction. On one hand, a component behavior model is defined to describe the relationship between services and references (or provided and required service interfaces) of a component. On the other hand, methods have to be developed for attaching non-functional properties (NFPs) to the SOA artifacts. We make proposals for the above mentioned two problems and raise a few open issues to be addressed in the project. The SOA component model specified in this document integrates into the system meta model. For details on the system meta-model see A1 deliverable core meta-model.
2.1.1 State Of The Art

The most popular and widely-used modelling language for software systems is OMG’s UML [57]. Together with the Meta Object Facility (MOF) specification [58] it provides the foundation for OMG’s Model Driven Architecture. UML also provides profiles to tailor the language to specific areas, such as the MARTE profile for modelling and analysis of real time systems [59].

UML is an indispensable part of the general modelling practice; however, its profiles and it are not ideal choices as the main modelling approach for SOA-based component architecture. Firstly, it is generic and does not provide the capability to describe all SOA-related artefacts. Secondly, some profiles such as MARTE are designed for real time systems and are too heavyweight for technical IT services. Nevertheless, MARTE does include a comprehensive modelling framework for Non Functional Properties (NFPs) which we can reuse a tailored subset in our system meta-model.

A model driven approach to constructing services and business processes is at the heart of SOA. This approach provides a foundation needed for greater efficiency and flexibility, and the Service Component Architecture (SCA) standardizes some of the key metadata for service modelling elements. It consolidates a standard approach for a SOA lifecycle, from development to composition and to deployment. The SCA standardization is now being carried out in OASIS, supported by leading industrial partners [60]. The specification consists of an assembly model and a policy framework as the foundation, which is independent of languages and technology. Additionally it also provides detailed specifications for different implementations and bindings. We propose to use SCA as the main modelling approach for SOA software components, and augment/extend it to support design time predictions.
2.1.2 SOA Component Assembly Model

Figure 1: The SOA Component Model based on SCA Assembly

Figure 1 shows the SOA component assembly model in a UML class diagram. It illustrates the SOA artefacts and their relationships. The modelling elements are defined as follows:

**Component**: Components are the basic elements of business function in an SCA assembly. They are configured instances of implementations which provide and consume services. The attributes of a component are shown in the Figure, most of which are self-explanatory. The attribute *name* is required, and others are optional. The important attributes that need explanations are:

- **requires**: a list of policy intents. See the Policy Model.
- **policySets**: a list of concrete policy sets. See the Policy Model.
- **constrainingType**: constrained by this constrainingType.
- **ServiceBehaviorSpecs (optional):** attach a component behaviour model.

**Service:** Service is the functionality provided by a component or composite. A service has zero or one **interface**, which describes the **operations** provided by the service. A service supports asynchronous communications using **callbacks**. A service has one or more **binding** elements as children. If no bindings are specified, the bindings specified by the implementation are in effect.

**Reference:** Reference is the functionality required from a service by another component or composite. It shares common attributes with a service, such as operation, requires, and policySets. It contains extra attributes such as target, which is a list of one or more of target service URI's, depending on the multiplicity setting.

**Interface:** An interface defines one or more operations providing business functions. Each operation has zero or one **request (input) message** and zero or one **response (output) message**. SCA currently supports **Java interfaces**, **WSDL 1.1 portTypes**, and **WSDL 2.0 interfaces**.

- Local and remote interfaces
- Bidirectional interfaces (callbackInterface)
- Conversational interfaces (stateful)

**Binding:** Binding are used by services and references to describe the access mechanisms, such as web service, SCA service, stateless session EJB, data base stored procedure, EIS service.

**Property:** Property elements are used to configure data values of the implementation of a component. Each property element provides a value for the named property, which is passed to the implementation.

- **source:** an XPath expression pointing to a property of the containing composite from which the value of this component property is obtained.

**Implementation:** Component implementations are concrete implementations of business functions for a component. SCA allows a wide range of **ImplementationType**, such as java, BPEL, or C++. Syntax:

- **implementation.java**, **implementation.bpel**, **implementation.ejb**, **implementation.spring**, **implementation.composite**.
**Wire**: Wires within a composite connect source component references to target component services. It is an alternative by using target attribute of a reference.

**Composite**: A composite is used to assemble SCA elements in logical groupings. It is the basic unit of composition within an SCA domain. Composites contain zero or more properties, services, components, references, wires, and included composites.

- **Promotion**: Composite services involve promotion of one service of one of the components within the composite. Composite references involve the promotion of one or more references of one or more components. Each promoted reference indicates that the component reference must be resolved by services outside the composite.

**ConstrainingType**: A constrainingType is expressed as an element which has services, references, and properties with intents applied to them. The constrainingType is independent of any implementation. Specifically it cannot contain bindings, policySets, property values, or default wiring information.

### 2.1.3 SOA Policy Model

![Figure 2: The SOA Policy Model based on SCA Policy Framework](image)
Figure 2 shows the SOA policy model based on SCA policy framework. **Policy** describes some capability or constraint that can be applied to service components (called implementation policies) or to the interactions between service components represented by services and references (called interaction policies). The key concepts are:

- **Intent**: An intent describes the requirements of components, services, and references. It is an abstract assertion about a specific QoS characteristic independent of any particular implementation technology. Examples are: authentication, integrity, confidentiality, reliability. An intent can be made **qualifiable** by adding a qualifier to the intent name, such as authentication.message, or authentication.transport.

- **PolicySet**: A policySet element defines a set of concrete policies that apply to some binding type or implementation type. It can use WS-Policy and WS-PolicyAttachment.

- **IntentMap**: As its name suggests, intent maps contain concrete policies that are used to realize a specific intent that is provided by the policySet.

*Intents* and *PolicySets* are attached to SCA constructs using the following keywords:

- **@requires**: list of intents
- **@policySets**: list of policySets

In the SCA policy framework specifies some security and reliability policies:

- **Security**: authentication, confidentiality, integrity
- **Reliability**: atLeastOnce, atMostOnce, ordered, exactlyOnce.

**SCA Policy Framework and the NFP model:**
We can see that the policy framework defined in SCA mainly addresses configuration policies and qualitative NFPs (security and reliability). It constitutes a subset of our NFP specification, which includes also quantitative NFPs. We can reuse the SCA policy framework to express qualitative NFPs and make other extensions for quantitative NFPs.

**SCA Extension for Quantitative NFPs:**
Resembling how policies are attached to SCA constructs, we can also use certain keywords to attach quantitative NFPs:

- **@qNFP (optional)**: requires the following quantitative NFPs to be specified.
- **@qNFPSets (optional)**: a set of concrete quantitative NFP values.
- **@ServiceBehaviorSpecs (optional)**: to attach component behaviour model.
2.1.4 SOA Component Behaviour Model

For enabling design time predictions we need the component developer to specify the internal control flows that connects internal and external service calls, and attach resource demand information on it. For this purpose we need a component behaviour model for describing the behavior of each provided service of a component. A basis model could be drawn from Palladio’s service effect specification (RDSEFF) [61], since such a behaviour model is already available and in use. Figure 3 shows the SOA component behaviour model with several basic actions such as branch, loop, and fork. One key point is to attach parametric resource demand to the actions, which should be compliant with the NFP specification in [SMM08]. The component behaviour model is optional and subject to future discussions. At the moment the component behaviour is specified in the QoS prediction model in A6.

2.1.5 Summary and Open Issues

We plan to extend SCA with quantitative NFPs, which are defined in the core meta model in A1. When applicable it is needed for the specification and attachment of component behaviour to component. This will most likely be specified in the QoS prediction model in A6.
2.2 Landscape Modelling

The second aspect of modelling undertaken by the WP A3 is modelling the IT landscape elements. The objective of this practise is to capture information about the software and service related elements and their relationships and dependencies. The task aims at capturing both planned as well as running landscapes. 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 packaging, configuration, versioning, deployment etc. Despite, the objective of having a meta-model to capture thorough and comprehensive information, the current version of the meta-model is a simplified version and we aim to extend the meta-model in the following years to be able to facilitate the scenarios and aspects mentioned previously.

In its current form, the meta-model is able to capture information about the software services, software elements required to provision these services, the packaging information for deployment purposes.

Figure 4: Software Landscape Meta-Model
2.2.1 Landscape Meta-model

We have designed a meta-model to capture information about the software landscape. Figure 4 shows the meta-model for expressing and capturing information about the software landscape in SLA@SOI. As mentioned, this is a simplified version aimed at catering for the requirements of the first year ad-hoc demonstrator and is subject to changes. The detailed discussion of the meta-model elements is described in the following paragraphs.

In its current incarnation, the meta-model focuses on the packaging, provisioning and deployment aspects of software services. As can be seen in the diagram, **SoftwareService** is the main element of the meta-model. **SoftwareService** is subclassed into two types of software services: **AtomicService** and **CompositeService**. **SoftwareElement** represents the software resources that materialize and facilitate the execution of both atomic as well as composite services. There are two types of software elements captured in this meta-model: **ApplicationSoftware** which represents the software that powers the service. To be precise, **ApplicationSoftware** entities are elements which materialize **AtomicService** entities and implement the application / business logic which gets executed to deliver the service functionality. **ApplicationSoftware** elements might depend on other software elements developed by third parties, hence, there is a self-relationship specified in the meta-mode. The second type of entities the meta-model captures are the execution software elements. **ExecutionSoftware** represents elements which provide execution environment for the **ApplicationSoftware** or **SoftwareService** elements. Examples of these elements include web servers, application servers, middleware, composition engines, database etc. **ExecutionSoftware** elements could be dependant on other elements for example ActiveBPEL depends on Tomcat servlet container. Hence, there is a required self-relationship for the **ExecutionSoftware** element.

This part of software landscape meta-model represents the service anatomy / make up as well as the execution dependencies. Using this information, the manageability framework can comprehend the provisioning requirements and dependencies of a service.

The second aspect that the meta-model represents is packaging and deployment aspect. **VirtualMachine** element represents the installation and execution container for the **SoftwareElement** entities. Multiple **SoftwareElement** entities can be installed and deployed onto the same **VirtualMachine** as well as these can be distributed on multiple **VirtualMachine** elements. The meta-model provides flexibility to the service providers to design the service deployment layout according to the functional and non-functional requirements of the customers. For example, one virtual machine would be sufficient to provide for the requirements of small installations, whereas, the deployment layout can span to multiple virtual machines each with individual parts of the service to provide scalable service landscapes. The last element of the software landscape meta-model is an **Appliance** element. An **appliance** represents a set of virtual machines packaged together which can be deployed onto the hardware resources. An appliance with one virtual machine containing all the software artefacts (**ApplicationSoftware** & **ExecutionSoftware**) is a valid example of an appliance. **Appliance** represents the deployment artefact. Service providers can install software elements onto virtual machines and package together which can
then be used during the service provisioning phase of the service lifecycle. Sample model instances are presented on the next pages.
2.2.2 Software Landscape Model Examples

Figure 5 shows an example model instance representing software landscape. It is a very simple landscape model for the Payment Service. All the service artefacts and software elements (both application software and execution software) are installed onto a single virtual machine and packaged into an appliance which can be provisioned.
Payment service is a composite service. This service is composed of two atomic services: Debit service and Card Validation service. All the services required application software represented by ORCSoftware. Payment service requires a process execution engine ActiveBPEL which in turn requires Axis2 webservice container. Since Debit service and Card validation services are also web services, these require execution software as well, therefore, there is a dependency shown in the model from these services to the Axis2 ExecutionSoftware element. A database is required by the services and the software. The database used in the example is Derby and a dependency link from the ORCSoftware to Derby represents this dependency. All these software elements are installed on a single virtual machine represented as PaymentServiceVM in the model diagram. installedOn links show this relationship. Finally, the virtual machine is packaged into an appliance shown as PaymentServiceAppliance.

Figure 5: software landscape example 1. Appliance with one virtual machine

Figure 6: software Landscape Example 2. Appliance packaged with three virtual machines
Figure 6 shows another example of the landscape with same service Payment Service. In this model diagram, the landscape is spread onto three virtual machines. The composite service and its required execution software is installed on a separate virtual machine, while a Derby database is installed on its own virtual machine. Atomic services and the required software elements are installed on the third virtual machine. All three virtual machines are packaged into the appliance payment service appliance.

These are simplistic landscape models showing individual appliances. In real world situations, the landscape model will capture information about multiple services packaged into various possible appliances which can be used for service provisioning.

3 Dynamic Service Discovery & Composition

This section aims to describe the relationship between dynamic composition, dynamic discovery, and Service/SLA Management and a methodology on how to integrate dynamic service discovery/composition to a holistic SLA management approach. In detail in a first step focus is given to look at existing composition techniques and successively to extend conceptually the chosen technique to incorporate the notion of SLA.

3.1 Definitions & approaches

“Web service composition involves compiling value added services from elementary or atomic services to provide functionalities that were not available or defined before” [6]

This definition of a composite service assumes two “kind of specifications”: on one side the specification of the binding information for the component services and, on the other side, the specification of a control and data flow among the services (the process definition). Moreover, traditionally, web service composition is classified into:

- manual/automatic
- static/dynamic

The distinction between static and dynamic composition concerns the “time” when Web services are composed while manual and automatic refers to the actor which “realizes” the composition. Static composition takes place during design-time when the architecture and the design of the software system are planned. The service components to be used are chosen, linked together and finally compiled and deployed.

This perhaps works fine as long as the Web service environment – business partners and service components – does not or only rarely change. If the environment changes\(^1\), it may be needed to update the used services or change the process definition or make both things on the fly, i.e. after the process has already been designed and deployed (for example when the composite service must be executed). We talk of dynamic composition in the case it is exclusively

\(^1\) e.g. Other business provide new services or the old services are replaced by other ones.
allowed to update the binding information of used services or in the case it is exclusively allowed to modify the process definition or both things are supported. Manual composition means that composition is performed by means of users who have access to the elementary services, while automatic implies that a software agent performs composition based on some predefined algorithms.

These two possibilities (static/dynamic, manual/automatic) represent “orthogonal” approaches applicable to the specifications of a composition. More than one combination is possible, also if the more consolidated and envisaged solutions belong, generally, to the categories below:

- Dynamic Workflow composition (known also as Template based) [6][7]
  - process specification: (static, manual)
  - binding specification: (dynamic, manual/automatic)

- Dynamic AI planning composition
  - process specification: (dynamic, automatic)
  - binding specification: (dynamic, automatic)

**Figure 7 Characteristics of main approaches to dynamic composition**

In the first one the process definition is realized statically as often as not manually by means of specific language. The opportunity of dynamicity, which is the research challenge in this category of solutions, is related to the binding specification that could be delayed during one of the phases after the deployment and realized automatically or manually. In detail, in workflow composition with “dynamic and automatic” binding the process description contains, for some (one at least) invoked services, placeholders instead of specific service bindings. Such process description is at a high level model of the process which we will refer to as an abstract process. Each placeholder can be associated with either:

- a service request\(^2\) or
- a set of already available and suitable services

The binding information is “implicitly” expressed and will be used to automatically search and select the desired service realization. Naturally the run time platform must be flexible enough to support these automatic actions.

In workflow composition with “dynamic and manual” binding it must be possible to specify the service realization to use, for an invoked service, on a process already deployed.

In the AI planning category, differently, the approaches are required to generate, dynamically and automatically, also the process definition other than the binding specification. Most methods in such category are related to AI planning and deductive theorem proving and they are generally referred as “AI Planning” [7].

In the context of SLA@SOI project the focus is mainly on processes where interactions are basically static, as it results from an analysis of the industrial use cases. In other words from a check of use cases specifications, no need of dynamic changes of control and data flow raised until now. For this reason we will propose a dynamic template based composition approach (see Figure 7).

\(^2\) A service request defines the functional/quality constraints for the desired service realization.

\[ e.g. \ SR(s) = SRC_{f1}(s) \land SRC_{f2}(s) \land \ldots SRC_{fn}(s) \land SRC_{q1}(s) \land SRC_{q2}(s) \land \ldots SRC_{qm}(s) \]

\[ SR = \text{Service Request}, \ s = \text{service}, \ SRC_{f} = \text{Service Request functional constraint}, \ SRC_{q} = \text{Service Request quality constraint} \]
### 3.2 Problem Statement

The SLA management impacts on SOA traditional approaches mainly for two reasons:

1. A simple/composite service, ready to be delivered, is always a negotiated service (not only discovered)
2. A simple/composite service, ready to be delivered, is identified by a Service Level Agreement (SLA) document shared by the provider and the customer (not only by an endpoint)

As secondary aspect in a business context a service provider has a special interest to avoid waste of resources provisioning a negotiated service.

On the base of these considerations, we can state our goal regarding dynamic composition as:

*To support the dynamic automatic/manual composition of negotiated services (by means of a dynamic template based approach—see Figure 7) and to simplify the provisioning of a composite service satisfying a negotiated agreement.*

### 3.3 Solution

#### 3.3.1 Methodology

**Dynamic automatic composition**

**Process Description Language**

We propose to use a standard process description language to describe an abstract process adopting “conventional endpoints” to indicate that a specific invoked service is unbound. This conventional endpoint represents the placeholder described in section 3.1. A conventional endpoint, in detail, uses standard URI satisfying an additional syntactic template (specified from the SLA@SOI project) to distinguish abstract service endpoints from concrete ones.

**Service Request**

Service request associated to the conventional endpoint must contain information to execute, automatically, the activities needed to achieve the information identifying a service realization. In the traditional SOA applications these activities are the Discovery and Selection and the searched/selected information fit with the service endpoint.

In SLA aware SOA applications and in our project the service selection can not be considered concluded until service has been negotiated. In fact the output of Negotiation is the description of use terms and condition of the negotiated service, containing also the information for the delivery of the service itself. The document containing such a description is the Service Level Agreement (SLA) and it identifies a deliverable service in our project.

The service request description, then, must contain constraints/rules which allows the automatic discovery, selection and negotiation of a service. In SLA@SOI, such a description is associated to a SLA Template document (SLAT). Coherently with
the SLA and Registry Model. A SLAT is used to represent the requirements that the SLA to establish should satisfy: functional and quality requirements about the activity to perform, terms related to required monitoring capabilities, required legal assurance and responsibility, rewards and penalties.

The adoption of SLA/SLAT to identify/describe service realizations instead of endpoint/WSDL is a peculiarity of our approach.

**Conclusion**

In SLA@SOI to deploy an abstract process it must be provided: a standard process description which uses conventional endpoint to recognize that an invoked service is unbound. A WSDL and a (queried) SLAT must be provided for each unbound invoked service. A WSDL and a SLA must be provided for each bound invoked service.

**Dynamic manual composition**

We propose a platform which offers an interface to set a SLA (or a SLAT) for an invoked service of a deployed process. As it will be clear from the next bullet this binding configuration action can involve all the instances belonging to a certain category or a specific process instance. Moreover it will be clarified the importance, from a business perspective, to offer such a composition possibility.

- **Simplified provisioning of a composite service satisfying a negotiated SLA.**

To better understand the reasons of a simplified provisioning let's give a look at the service lifecycle assumed in SLA@SOI.

After the development phase a service provider may offer the same service to different customers allowing them to customize some functional or quality characteristics of the offered service. After a negotiation phase a customer and the provider formalize the description of the service to provide in a service level agreement (SLA). The provisioning is the set of actions performed by the provider to activate the agreed service. Operations refer to the phase of service execution.

Very roughly, with the current state of the art technologies and standards a common way to realize these phases is to: define, in the development phase, a process, customize it to satisfy the signed (overall and eventually component) SLAs and finally deploy inside an engine (see Fig.

---

1 See section “2.1.3 SLA Templates” in D.A5a-SLA Foundations and Management.doc
Figure 8: more than one deployed process for the same offered service

We think this approach is far than optimal as fundamentally you deploy several time the same process often changing just the component services binding to guarantee a different negotiated QoS. To avoid this waste, we propose to anticipate the deployment of an unbound workflow before the provisioning phase (for instance in the context of the Offering as showed in Fig 8) and then dynamically submit to the workflow engine specific binding information for any customer.

Figure 9: one deployed process for each offered service: results of negotiation are binding information and not different processes

In such a way it’s possible to deploy one single unbounded process description file delegating the setting of different configuration (i.e. binding information
specification), for each customer, to the Provisioning or even to the Execution phase. To support this provisioning simplification our solution must enable:

- the deployment of unbound processes;
- the configuration of binding for a specific process instance or for all instances belonging to a certain category;

Concerning the first point we have already talked about the idea of using a standard process description language adopting conventional endpoints to indicate that a specific invoked component service is unbound.

About the second one we have just talked (see bullet above) of the idea to offer a platform which makes possible to set/change manually the binding configuration on a deployed process. In detail we want allow changing the binding of a specific process instance or of a set of process instances (for example all the instances that will be created or have already been created for a specific customer). A category of process instances will be defined by a set of constraints on variables defined in the unbound workflow.

The decision to support manual dynamic binding (other than the automatic one) is due to the wish to allow provider “to choose” when to make binding. This opportunity is relevant from a business point of view as its better clarified by the figure sketched below. In such figure the service life cycle phases are indicated on the x-axis to highlight the different binding time opportunities, while qualitative traces (trends) for information of interest are drawn in the plane (The traces are put together, in the same plane, only for convenience reason avoiding to produce distinct drawings and since different measure units can be associated to each of such information no specific indication has been associated to the y-axis).

![Figure 10 Manual dynamic binding vs automatic dynamic binding](image)
Information details:

- **Execution guarantee**
  It represents the probability, for the provider, to honour the guarantees subscribed for the whole service and exposed in the agreement.

- **Nightly service exploitation**
  It represents the probability to use the last published services which could offer higher quality or minor cost.

- **Execution overhead**
  It represents the time which must be added to the execution duration of a composite service made exclusively of bound invoked services.

If the provider wants to be sure to execute the composite service (high execution guarantee) and strong execution time constraints (low execution overhead) are specified in the agreed SLA, then, he prefers to specify a service realization, for each invoked service, before the operations phase. Therefore during the negotiation of the composite service, he negotiates (assuming the role of a Customer) also for the component services. Finally he will specify the binding information (see manual dynamic binding in the figure 7). On the other hand if the provider would offer great quality performances (high nightly service exploitation) and the overhead execution time is not a problem for negotiated composite service then, probably, he prefers a dynamic automatic composition at invocation or execution time (see automatic dynamic binding in the figure).

### 3.3.2 Requirements & Architecture

**Dynamic automatic composition use cases**

**Use cases during Design & Development**

**Figure 11: Deploying an abstract process**

**Functionality**

- **Name:** deploy abstract process description
- **Input:** standard abstract process description, SLAT of the whole service, WSDLs and SLAs /SLATs for each invoked service.
- **Output:** None
- **Side effect:** None
- **Rationale:** Included in DOW
- **Comment:** A SLA is provided for each bound invoked service. A (queried) SLAT is provided for each unbound invoked service.

---

4 i.e. For “invocation time” we mean here the process instance invocation time while for “execution time” the execution instant of an invoke activity.
Component: Orchestration engine

Use cases during Operations

![Diagram of SLAT discovery](image)

**Figure 12: SLAT discovery**

**Functionality**
Name: (offered) SLATs discovery  
Input: (queried) SLA Template  
Output: List of SLA Templates (of service offers) that match with the input  
Side effect: None  
Rationale: Included in DOW  
Comment: Currently we assume this functionality is completely implemented inside task A5.2  
Component: SLAT Registry

![Diagram of SLAT selection capabilities](image)

**Figure 13: SLAT selection capabilities**

**Functionality**
Name: local SLAT selection  
Input: List of discovered SLA Templates  
Output: SLA Template with the best rank.  
Side effect: None  
Rationale: Included in DOW  
Comment: The list of discovered SLA Templates is ordered on the base of order criterion specified in the queried SLAT  
Component: SLAT Selection

**Functionality**
Name: global SLAT selection
**Input:** Lists of discovered SLA Templates, SLA of the whole service  
**Output:** List of selected SLA Templates (a SLA Template for each unbound invoked services of the process)  
**Side effect:** None  
**Rationale:** Included in DOW  
**Comment:** The selection is based on a global optimization criterion expressed in the SLA of the whole service.  
**Component:** SLAT Selection

![Figure 14: SLA Negotiation](image)

**Functionality**  
**Name:** negotiate SLA of an unbound invoked service  
**Input:** SLA Template from selection, SLA of the whole process  
**Output:** SLA of the invoked service  
**Side effect:** None  
**Rationale:** Included in DOW  
**Comment:** The negotiation has to: translate properties from SLA in constraints of properties of the selected SLA template, prepare an offer and start the negotiation process. The translation or the negotiation could fail, in such case a new selected SLA Template must be provided to this functionality.  
**Component:** SLA Negotiation

In Figure 15 is sketched an activity diagram to better clarify the interactions among the components which implement the functionalities we have just...
The adaptation activity takes in charge to prepare all the conditions to make the selected service ready to be invoked. The engine achieves this objective by specifying the binding information for the invoked service (of the process instance under execution) using own functionality. Such functionality is available also to an external actor to manually set the binding therefore we prefer describe it in the next section.
Dynamic manual composition use cases

**Figure 16: dynamic manual composition**

**Functionality**
- **Name**: set invoke binding (of a process instance)
- **Input**: unbound invoked service ID, process instance ID, SLA
- **Output**: void
- **Side effect**: None
- **Rationale**: Included in DOW
- **Comment**: This functionality is available to specify the binding of an invoked service for an executing process instance. It is used also by the engine itself as explained in the previous section.
- **Component**: Orchestration engine

**Functionality**
- **Name**: set invoke binding (of a process instances category)
- **Input**: unbound invoked service ID, category, constraint, SLA(SLAT)
- **Output**: void
- **Side effect**: None
- **Rationale**: Included in DOW
- **Comment**: This functionality enables to specify the binding of an invoked service for a category of process instances that will be created or have already been created.
- **Component**: SLA Negotiation

**Architecture**

**Architectural and technological requirements**

- We aim to be compliant with standards as much as possible then we will use standard BPEL4WS as process description language assuming conventional endpoint to indicate unbound invoked service.
- A BPEL compliant runtime platform is not able to support the execution of an abstract invocation. Most common solution to this problem is to invoke a “Proxy” instead of actual service implementation. Any service proxy implements the service needed by a specific abstract invocation maintaining the description (a SLAT for us) of the needed service and, when invoked, performs Discovery, Selection, Negotiation actions. For each abstract invocation a service proxy must be instantiated and deployed. This solution enables to continue using a standard BPEL engine but produces a huge waste of resources.
We propose a solution based on an extension, as limited as possible, of a standard BPEL engine to avoid this waste. In detail the idea is to extend the engine by means of one component (Binder) which is responsible:
- to hold the SLAT of the composite service and the SLA/SLAT for each invoked service of all deployed processes.
- to retrieve the SLAT for an unbound invoked service and invoke the discovery functionality.
- to invoke the selection functionalities.
- to invoke the negotiation functionality.

Besides it exposes the binding configuration functionalities, described in the use cases section, which enables the dynamic manual composition (see Figure 177 for a conceptual architecture).

Figure 17: conceptual architecture

- Finally, Binder component can be reused also in an alternative (less efficient but more standard) architecture based on one centralized proxy service, without the need to extend any BPEL engine.
4 Service Monitoring

Runtime system monitoring, as opposed to static system analysis and testing is often the only meaningful way to perform verification of service based systems, in which both the involved software services and infrastructural elements may change dynamically according to contextual factors, such as the system load or the availability of new components. When SLAs regulate the provisioning of service based systems, then the properties expressed in SLAs become the properties that need to be monitored during the service execution.

Within SLA@SOI, we consider an event-based approach to service monitoring, in which monitoring properties, derived from SLAs, are checked against events, e.g. service calls and responses, describing the evolution of the service execution environment. Therefore, besides monitoring engines, i.e. reasoning tools to check the satisfaction of monitoring properties, service monitoring also requires the instrumentation of the service execution environment. Such instrumentation should be able to capture the events relevant for monitoring and to make them available to the core reasoning engines of the service monitoring framework.

Section 4.1 discusses the instrumentation of the service execution environment, focusing on the Open Reference Case (ORC) services used in the context of the Year 1 demonstrator. In particular, we show the instrumentation at both the level of atomic and composite (i.e. BPEL business processes) services. Section 4.2 discusses the approaches to event-based service monitoring used within the project. More specifically, we adopt the monitoring approach based on Event Calculus [66] developed at CITY University for the monitoring of atomic services [62][63][65], whereas composite services are monitored through the approach developed at FBK, based on the language RTML [67]. The objective is to show how the two different approaches for monitoring can be plugged in into the overall service monitoring framework of the project, using the events produced by the instrumentation of the service execution environment.

4.1 Service Instrumentation

Our monitoring approaches predicate over events that are generated by our runtime execution framework. The generation of these events is treated as a crosscutting concern and implemented through instrumentation. We currently support the generation of two kinds of events: Service Invocation Events and Process Status Events. The former are events that allow us to know when a process or an atomic service interacts with a remote partner, or when a process starts and terminates. The latter pertain only to the process engine, since they allow us to capture a running process’ internal state at any point of its execution.

Figure 18 illustrates the four events that relate to a typical service invocation by part of the process engine. First, the Process Engine sends a SOAP request message to the Atomic Service Container. As soon the request message is issued, we create a time-stamped event and send it to the monitoring architecture. As soon as the request message is received we create a second time-stamped event. When the atomic service issues its response message, we create a third time-
stamped event. Finally, when the engine receives the response message we create a fourth time-stamped event.

![Diagram showing the four events created during a standard process-service interaction]

**Figure 18: The four events created during a standard process-service interaction**

To support such a scenario we need to provide instrumentation code that is run both on the process’ side and on the atomic service’s side.

### 4.1.1 Process Instrumentation

Process instrumentation is implemented using Aspect Oriented Programming (AOP) [67] to augment ActiveBPEL [69], a well-known open-source BPEL execution engine, with event creation capabilities. Since ActiveBPEL is implemented in Java we have chosen to adopt AspectJ [70].

The overall process instrumentation framework is illustrated in Figure 19. The infrastructure is made of four main components. The most important is the *Instrumentation Interceptor*. It represents the actual AspectJ device that intercepts the execution of a process instance. It is responsible for creating the monitoring events. The specification of what monitoring events need to be created is contained within the *Derby DB* [71], which can be accessed through the *Database Manager*. Note that the infrastructure also supports the “hot deploy” of new instrumentation specifications through the *Hot Deployer* component which
periodically checks a local folder for new XML instrumentation specifications.

Figure 19: The Process Instrumentation Infrastructure.

To understand how process-side instrumentation works we must first understand how ActiveBPEL works. ActiveBPEL works by creating a Definition Tree for each deployed process (regardless of the number of instances running concurrently), and an Implementation Tree, which extends the previous one, for each new instance. To execute a process instance, the engine traverses the Implementation Tree and calls the “execute” methods defined for each node type. Using AspectJ we intercept calls to method “execute” on BPEL activities Invoke, Receive, and Reply to know when to create Service Invocation Events, and on any BPEL activity to capture Process Status Events.

Service Invocation Events

To specify the creation of a Service Invocation Event the designer must create a Service Invocation Event Specification, that is an XML specification file that follows schema “sie.xsd”. For each event the designer must specify:

- **uniqueID**: this is the unique identifier for the event.
- **targetEPR**: this indicates where to send the event once it has been created. Events are always sent to a monitoring bus instance.
- **targetName**: this indicates what channel name to use when sending the event to the monitoring bus instance. All components registered with the monitoring bus, and “listening” on that channel, will receive the event.
- **processName**: this is the name of the process for which the designer wants to define an event.
- **activity**: this is a unique identifier of the BPEL activity that, when executed, activates the creation of a new event. Unique identifiers are given using an XPATH selection expression defined over the XML definition of the BPEL process. In the case of Service Invocation Events the expression should point to a BPEL invoke, receive, or reply activity. We also support the special keyword “processStart” to indicate that we are interested in signalling the moment in which the process starts executing, and the keyword “processStop” to indicate we want the moment in which the
process terminates.

- **direction**: this indicates whether we are interested in creating an event on an incoming or an outgoing message. The direction can be either “input” or “output”. The semantics follow the semantics used within the BPEL standard. For example, the “input” to a BPEL invoke activity is actually the variable that is sent to the partner service, while the “output” of a BPEL invoke activity is the variable received as a response. In the case of a “processStart” or “processStop” operation, the direction is ignored.

Figure 20 illustrates an example of Service Invocation Event specification for the ActiveBPEL tutorial example process called “loanProcessCompleted”.

![Figure 20: Example of a Service Invocation Event specification.](image)

The result is the creation of an event that follows the Event Instance element type defined in the “MonitoringEvent.xsd” schema.

**Process Status Events**

Process status events are used to gather information about the internal state of an executing process. To specify the creation of a Process Status Event, the designer must create a Process Status Event specification, that is an XML specification file that follows schema “pse.xsd”. For each event the designer must specify:

- **uniqueID**: this is the unique identifier for the event.

- **targetERP**: this indicates where to send the event once it has been created. Events are always sent to a monitoring bus instance.

- **targetName**: this indicates what channel name to use when sending the event to the monitoring bus instance. All components registered with the monitoring bus, and “listening” on that channel, will receive the event.

- **processName**: this is the name of the process for which the designer wants to define an event.

- **activity**: this is a unique identifier of the BPEL activity that, when executed, activates the creation of a new event. Unique identifiers are given using an XPATH selection expression defined over the XML definition of the BPEL process. In the case of Process Status Events the expression can point to any
BPEL activity.

- **variableName**: this is a unique identifier of a BPEL variable that is part of the executing process’ state. The variable name must be given in accordance with the state visibility rules defined by the BPEL standard. This means that the variable must belong to the specified operation or to a recursively enclosing scope.

- **dataExtraction**: this information is not necessarily required. If given, it indicates that the designer is not interested in the entire BPEL variable but in a part of it. What part the designer wants is specified using an XPATH expression that is applied to the chosen variable.

Figure 21 illustrates an example of Process Status Event specification for the “loanProcessCompleted” tutorial BPEL process.

```xml
<?xml version="1.0" encoding="utf-16"?>
<!DOCTYPE def SYSTEM "bpi.dtd">
<def>
  <workflowStatus>
    <uniqueID value="0001"/>
    <targetEPR value="MonitorBusEPR"/>
    <targetName value="Monitoring"/>
    <processName value="loanProcessCompleted"/>
    <activity value="/process/flow/receive
        [@name=ReceiveCustomerRequestForLoanAmount]"/>
    <variableName value="creditInformation"/>
    <dataExtraction value=""/>
  </workflowStatus>
</def>
```

**Figure 21: Example of Process Status Event specification.**

The result is the creation of an event that follows the Event Instance element type defined in the “MonitoringEvent.xsd” schema. This time, however, the event also carries the extracted state data as payload.

### 4.1.2 Service Instrumentation

The instrumentation of atomic services currently only supports the creation of Service Invocation Events that is of events that signal the arrival of a request
The atomic service container of reference in the SLA@SOI project is AXIS 1.4 [72]. Figure 22 gives an overview of how instrumentation is implemented in the AXIS container. First of all, AXIS uses handlers to manage incoming and outgoing messages. Handlers are what prepare messages for fruition by part of the service, and for delivery. There are three kinds of handlers: those that manage message transport at lower levels, those that pertain to the AXIS container’s global configuration, and those that pertain to a single AXIS service. Typical examples of handlers may be logging or authentication facilities.

To support the ever-growing world of Web standards, AXIS provides a means to extend the chain of handlers that are used by the container (see Figure 22). This is achieved through a common handler interface. Designers need to implement the interface and configure the container to consider the new handler along side default ones. This is also the standard means provided by AXIS for including crosscutting concerns.

Our instrumentation code is added as a generic handler that reads its directives from a MySQL [73] database. This level of indirectness allows us to modify the specifications in the database without having to undeploy and re-deploy the actual service. Indeed, we support hot deployment of event specifications, just as we did for process-side instrumentation.

**Service Invocation Events**

To specify the creation of a Service Invocation Event the designer must create a Service Invocation Event Specification, that is an XML specification file that follows schema “axis_sie.xsd”, a variant of schema “sie.xsd”. For each event the designer must specify:

- **uniqueID**: this is the unique identifier for the event.
- **targetEPR**: this indicates where to send the event once it has been created. Events are always sent to a monitoring bus instance.
targetName: this indicates what channel name to use when sending the event to the monitoring bus instance. All components registered with the monitoring bus, and "listening" on that channel, will receive the event.

serviceEPR: this is the unique EPR through which the atomic service is reached.

direction: this indicates what message exchange we are interested in signalling. We support keywords "incoming" to indicate a request message being received by the container and "outgoing to indicate a response message being sent out by the container.

Figure 23 shows an example of Service Invocation Event specification.

The result is the creation of an event that follows the Event Instance element type defined in the "MonitoringEvent.xsd" schema.

4.2 Service Monitoring Architecture

Within SLA@SOI, Monitoring has a twofold focus. On the one hand, we consider service monitoring, i.e. the set of techniques and tools required for monitoring dynamic service-based systems. On the other hand, at a higher level of analysis, the project considers SLA monitoring. SLA monitoring concerns (i) deriving monitoring properties from the SLA established for the provisioning of a service-based system and (ii) coordinating monitoring related information captured at different layers of service provisioning, i.e. business, software, and infrastructure services.

In this section, we discuss the issues related to service-monitoring, describing the techniques and tools required for monitoring properties of service based systems. Since service monitoring takes place within the broader picture of SLA monitoring, we first provide a brief introduction to the overarching SLA monitoring framework. Details on SLA monitoring framework are discussed in depth in Deliverable A5a.

Service monitoring within SLA@SOI is event-based. Event-based (or non-intrusive) service monitoring is made only on the basis of events, such as service operation calls and responses, which are captured within the service system execution environment. In this way, the monitoring infrastructure is completely
decoupled from the services’ execution environment. Event-based monitoring is usually opposed to intrusive monitoring of service systems, in which monitoring is achieved through the instrumentation of the service execution environment (e.g. checking for pre- and post-conditions on service operation calls within a BPEL engine). Although it has several benefits, intrusive monitoring implies tighter coupling of the monitoring features and the service execution environment.

Event-based monitoring fits naturally for the SLA@SOI ad-hoc demonstrator requirements, according to which the SLA Management Framework must be completely decoupled from the services composing the Retail solutions offered to customers (Open Reference Case). More specifically, we can not assume that the SLA Management Framework designed within SLA@SOI may have control over the execution environment of the services composing the products offered to customers.

![Figure 24 – Overall architecture of SLA@SOI SLA Monitoring Framework](image)

The Event Bus is used to collect (i) the events produced by the software service execution environment instrumentation and (ii) the monitoring result (SLA violations) produced by the service monitoring. Service monitoring is performed within the SLA monitoring framework. In particular, the Monitoring framework is developed according to a plug-in, in which several different Generic monitor engines may be plugged in as long as they comply with the interface declared in the SLA Framework. In this section we provide detail on the two monitor engines used within the SLA@SOI ad-hoc demonstrator, i.e. the monitor engine at City University (from now on CITY monitor engine) and the monitor engine developed at FBK (from now on FBK monitor engine).

The following two sections describe, for each of these monitor engines, the techniques adopted to perform service monitoring, with a specific reference to the scenario for the SLA@SOI ad-hoc demonstrator, and internal architecture of the monitoring engine.

### 4.2.1 Atomic Service Monitoring

The monitor engine proposed by CITY allows the monitoring of rules expressed in Event Calculus (EC). EC allows the expression of generic monitoring properties that can be applied for the monitoring of behavioural and non-functional properties of several kinds of dynamic software systems. In different forms and
extensions, the CITY monitor engine has been applied in the context of process-based service systems, ambient intelligence, and mobile computing [62, 63, 64].

Before going into detail in the internal architecture of the CITY monitoring engine, we first provide a brief introduction on EC and on how EC can be used to express monitoring rules on the Guarantee Term defined for the SLA@SOI ad-hoc demonstrator.

The Event Calculus (EC) [66] is a first-order temporal formal language that can be used to specify properties of dynamic systems which change over time. Such properties are specified in terms of events and fluents. An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation) and may change the state of a system. Fluents are conditions regarding the state of a system and are initiated and terminated by events. A fluent may, for example, signify that a specific system variable has a particular value at a specific instance of time.

The occurrence of an event is represented by the predicate Happens(e, t, R(t₁, t₂)). This predicate signifies that an instantaneous event e occurs at some time t within the time range R(t₁, t₂). The boundaries of R(t₁, t₂) can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The initiation of a fluent is signified by the EC predicate Initiates(e, f, t) whose meaning is that a fluent f starts to hold after the event e at time t. The termination of a fluent is signified by the EC predicate Terminates(e, f, t) whose meaning is that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicates Initially(f) and HoldsAt(f, t) to signify that a fluent f holds at the start of the operation of a system and that f holds at time t, respectively.

In order to specify monitoring rules in the context SLA-driven service systems, we extend Event Calculus with special types of events and fluents to specify service guarantee terms, and their qualifying conditions and assumptions. More specifically, the special fluent that we consider has the form

\[
\text{valueOf(} \text{fluent\_expression, value\_expression} \text{)}
\]

representing that the fluent signified by fluent_expression has the value value_expression. Furthermore, in this expression fluent_expression denotes a typed system variable or a list of such variables. fluent_expression may be (i) an internal variable that represents a variable of the composition process of a service system, or (ii) an external variable that is introduced by the creators of a service level agreement to represent the state of a service system at runtime. If fluent_expression has the same name as a variable in the service system then it denotes this variable, has the same name with it, and is treated as an internal variable. In all other cases, fluent_expression denotes an external variable and its type is determined by the type of value_expression as described below. value_expression is a term that either represents an EC variable or signifies a call to an operation that returns an object of some type. The operation called by value_expression may be an internal operation that is provided by the monitoring engine or an operation that is provided by an external web-service.

The internal operations which may be used in the specification of fluents are shown in Table 1 Note also that a fluent is valid if and only if the type of fluent_expression is more general than the type of value_expression. If fluent_expression is an external variable, the specification of its type is deduced from the type of value_expression in a fluent specification. In this case, if
fluent_expression appears in different fluents that use different value_expression terms, the above type validity condition should be satisfied by the types of all the relevant value_expression terms. On the other hand, if fluent_expression is an internal variable, its type is determined by the specification of the variable in the composition process of the SBS system that it refers to.

The calls to external and internal operations in fluents allow us to deploy complex computations. As shown in Table 1, the internal operations of EC-Assertion, for instance, can perform various arithmetic operations over numbers and compute statistics of series of numerical values (e.g., compute the average, median and standard deviation of a series of values), manage lists of primitive values and create new instances of object types. These operations are necessary for checking QoS requirements within the reasoning process of the monitoring engine. The maintenance of lists of primitive data values, for instance, is useful for recording multi-valued fluents (e.g., recording the response times of a service operation). The operation avg for instance which computes the average value of a list of real or integer number can be used to compute the average response time of a service operation. During a monitoring session, when attempting to unify formulas which include such calls, the EC variables which represent the operation parameters are unified first and then the monitor calls the relevant operation. If the operation returns successfully with a return value that is compliant with the type of fluent_expression, this value becomes the binding of the term value_expression. Otherwise, unification fails.

Table 1 - Built-in operations for specification and computation of service guarantee terms

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(n1:Real, n2:Real): Real</td>
<td>This operation returns n1+n2</td>
</tr>
<tr>
<td>sub(n1:Real, n2:Real): Real</td>
<td>This operation returns n1-n2</td>
</tr>
<tr>
<td>mul(n1:Real, n2:Real): Real</td>
<td>This operation returns n1*n2</td>
</tr>
<tr>
<td>div(n1:Real, n2:Real): Real</td>
<td>This operation returns n1/n2</td>
</tr>
<tr>
<td>append(a[], list of &lt;T&gt;, e:T): list of &lt;T&gt; where T is Real, Int or String.</td>
<td>This operation appends e to a[].</td>
</tr>
<tr>
<td>del(a[], list of &lt;T&gt;, e:T): list of &lt;T&gt; where T is Real, Int or String.</td>
<td>This operation deletes the first occurrence of e in a[].</td>
</tr>
<tr>
<td>delAll(a[], list of &lt;T&gt;, e:T): list of &lt;T&gt; where T is Real, Int or String.</td>
<td>This operation deletes all occurrences of e in a[].</td>
</tr>
<tr>
<td>size(a[]): list of &lt;T&gt;): Int where T is Real, Int or String.</td>
<td>This operation returns the number of elements in a[].</td>
</tr>
<tr>
<td>max(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real, Int or String.</td>
<td>This operation returns the maximum value in a[].</td>
</tr>
<tr>
<td>min(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real, Int or String.</td>
<td>This operation returns the minimum value in a[].</td>
</tr>
<tr>
<td>sum(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real or Int.</td>
<td>This operation returns the sum of the values in a[].</td>
</tr>
<tr>
<td>avg(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real or Int.</td>
<td>This operation returns the average of the values in a[].</td>
</tr>
<tr>
<td>median(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real, Int or String.</td>
<td>This operation returns the arithmetic median of the values in a[].</td>
</tr>
<tr>
<td>mode(a[]): list of &lt;T&gt;):&lt;T&gt; where T is Real, Int or String.</td>
<td>This operation returns the most frequent element in a[].</td>
</tr>
<tr>
<td>new(type_name: String): ObjectIdentifier</td>
<td>This operation creates a new object instance of type T and returns an atom that is a unique object identifier for this object.</td>
</tr>
</tbody>
</table>

Events in our framework represent exchanges of messages between the composition process of an service-based system system and the services coordinated by it. These messages either invoke operations or return results.
following the execution of an operation. In the scenarios considered by the SLA@SOI ad-hoc demonstrator, we consider the following signatures for events:

**Service operation invocation events** – These events signify the invocation of an operation in one of the partner services of an SBS system by its composition process and are represented by terms of the form:

\[ \text{ic}: S: O(_\text{Oid}, _P_1, \ldots, _P_n) \]

where \( O \) is the name of the invoked operation; \( S \) is the name of the service that provides \( O \), \( _\text{Oid} \) is a variable identifying the exact instance of \( O \)'s invocation within an execution of the service systems, and \( _P_1, \ldots, _P_n \) are variables indicating the values of the input parameters of \( O \) at the time of its invocation.

**Service operation reply events** – These signify the return from the execution of an operation that has been invoked in a service and are represented by terms of the form:

\[ \text{ir}: S: O(_\text{Oid}, _P_1, \ldots, _P_m) \]

where \( O, S, _\text{Oid} \) are defined for service operation invocations and \( _P_1, \ldots, _P_m \) are the eventual output parameters of the operation.

In addition to the EC predicates and events/fluents denoting terms that were discussed above, formulas that express monitorable properties can use the relational predicates \(< \text{ and } = \) to express time conditions (the predicate \( t1 < t2 \) is true if \( t1 \) is a time instance that occurred before \( t2 \), and the predicate \( t1 = t2 \) is true if \( t1 \) is a time instance that is equal to \( t2 \)) and to compare values of different variables. Also an EC formula that expresses a monitorable property must specify boundaries for the time ranges \( R(LB, UB) \) which appear in the \( \text{Happens} \) predicates. If the variable \( t \) in such predicates is existentially quantified, at least one of LB and UB must be specified. These boundaries can be specified by using: (i) constant time indicators or (ii) arithmetic expressions of time variables \( t' \) which appear in \( \text{Happens} \) predicates of the same formula provided that the latter variables are universally quantified, and that appears in their scope. If \( t \) is a universally quantified variable both LB and UB must be specified. \( \text{Happens} \) predicates with unrestricted universally quantified time variables take the form \( \text{Happens}(e,t,R(t,t)) \). These predicates express instantaneous events. Furthermore, a formula is valid in our framework if the time variables of all the predicates which include existentially quantified non-time variables, take values in time ranges with fixed boundaries. These restrictions guarantee the ability to check the satisfiability of formulas.

A generic monitorable property is then expressed by a rule and possibly one or more assumptions. A violation of the rule implies a violation of the property to which the rule refers. Assumptions are additional condition which are necessary to evaluate the rule. Both rules and assumption assume the form:

\[ \text{Body} \Rightarrow \text{Head} \]

where \( \text{body} \) and \( \text{head} \) are logical formulas expressed using the predicates, events, fluents, and internal operations previously described. In other words, a rule is violated when the \( \text{body} \) assumes the logical value TRUE and the \( \text{Head} \) assumes the value FALSE.
In the following, we provide the parametric monitoring rule expressed in EC required to evaluate the Guarantee Terms that appear in the SLAs of the SLA@SOI ad-hoc demonstrator. Before providing the actual EC rule, we remind that the archetypical Guarantee Terms, specified in WS-Agreement, assumes the following reported in Figure 23.

```
<wsag:GuaranteeTerm wsag:Name="CompletionTime_GetProductInformation_Gterm"
wsag:Obligated="ServiceProvider">
  <wsag:ServiceScope wsag:ServiceName="InventoryService" />
  <wsag:QualifyingCondition>
    <coremodel:Expression xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:coremodel="http://www.slaatsoi.org/coremodel"
xmlns:terms="http://www.slaatsoi.org/commonTerms">
      <Expression>
        <Predicate xsi:type="coremodel:LessEqual">
          <OperandA xsi:type="terms:ArrivalRate" description="Non-Functional"
context="InventoryService_getProductInformation_ServiceProperty_arrivalRate_Variable"
qualifier="" />
          <OperandB xsi:type="coremodel:Constant">
            <type xsi:type="coremodel:Frequency" uniqueID="" value="555.56" />
          </OperandB>
        </Predicate>
      </Expression>
    </wsag:QualifyingCondition>
    <wsag:ServiceLevelObjective>
      <wsag:KPITarget>
        <wsag:KPIName>InventoryService_getProductInformation_Gterm_KPI_CompletionTime</wsag:KPIName>
        <wsag:CustomServiceLevel>
          <!-- 20ms for 95% -->
          <coremodel:Expression xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:coremodel="http://www.slaatsoi.org/coremodel"
xmlns:terms="http://www.slaatsoi.org/commonTerms">
            <Expression>
              <Predicate xsi:type="coremodel:LessEqual">
                <OperandA xsi:type="terms:CompletionTime"
context="InventoryService_getProductInformation_ServiceProperty_completionTime_Variable"
qualifier="">
                  <statQ>percentile</statQ>
                  <statQValue>95.0</statQValue>
              </OperandA>
              <OperandB xsi:type="coremodel:Constant">
                <type xsi:type="coremodel:Duration" uniqueID="" unit="ms" value="20.0" />
              </OperandB>
            </Predicate>
          </Expression>
        </wsag:CustomServiceLevel>
      </wsag:KPITarget>
    </wsag:ServiceLevelObjective>
  </wsag:GuaranteeTerm>
```

Figure 25 – Example of SLA Guarantee Term for SLA@SOI Ad-Hoc demonstrator

The Guarantee Term in Figure 25 applies to a specific operation of a service in the Open Reference Case. It defines a Qualifying Condition on the customer request’s arrival rate, i.e. arrival rate less than 555.56 req/s, and a Service Level Objective on the average completion time of the operation invocation, i.e. completion time must be less than 20ms in 95% of invocations. We remind that the Guarantee Term is defined by a logic implication between the Qualifying Condition and the Service Level Objective, i.e. Qualifying Condition => Service Level Objective. In other words, the Guarantee Term is violated when the Qualifying Condition is verified (logical value TRUE) and the SLO is not met (logical value FALSE). Note that the Qualifying Conditions and SLOs within a Guarantee Term are defined in respect of the SLA@SOI Core model.
In order to monitor the Guarantee Term in Figure 23 for a generic Operation of a service in the Open Reference Case, the monitor engine requires the timestamps of events signifying the invocation of an operation and the related responses. According to the formalism previously introduced, such events, for a generic Operation exposed by service S, assume the following form:

Service operation invocations - \( \text{ic}:S:\text{Operation}(\_\text{OperationID}, P_1,\ldots,P_n) \)

Service operation responses - \( \text{ir}:S:\text{Operation}(\_\text{OperationID}, P_1,\ldots,P_m) \)

The EC rule which monitors the Guarantee Term in Figure 23 for a generic Operation is constituted by 4 assumptions and 1 rule, which are shown in the following.

**Assumption A1** – Initialising the counter of operation calls with completion time within range (counterWithin).

\[
\text{Initially} (\text{counterWithin}); \\
\]

**Assumption A2** – Initialising the counter of operation calls with completion time not within range (counterNotWithin).

\[
\text{Initially} (\text{counterNotWithin}); \\
\]

Assumption A1 and A2 initialise, using the predicate Initially, the values of the counters counterWithin and counterNotWithin for, respectively, the service operation invocations in which the completion time has complied with the SLO (completion time less than 20ms) and the ones in which the SLO has been violated (completion time exceeded 20ms).

**Assumption A3** - Updating the counter of operation calls with completion time within range

\[
\forall t_1, t_2 \\
\text{Happens} (\text{ic}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), t_1, R(t_1, t_1)) \ \text{AND} \\
\text{Happens} (\text{ir}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), t_2, R(t_1, t_1+20ms)) \ \text{AND} \\
\text{HoldsAt} (\text{valueOf} (\text{counterWithin}, _x), t_2) \\
\Rightarrow \\
\text{Initiates} (\text{ir}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), \text{valueOf} (\text{counterWithin}, \text{add}(x, 1), t_2)); \\
\]

Assumption A3 updates the value of counterWithin, in case the service operation call has succeeded in less than 20ms. The update is expressed by using the special fluent valueOf() previously introduced. Note that every time the head requires the evaluation of a fluent at a time point, the body should contain a related HoldsAt predicate that specifies the value of the fluent at the considered time point.

**Assumption A4** - Updating the counter of operation calls with completion time not within range

\[
\forall t_1, t_2 \\
\text{Happens} (\text{ic}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), t_1, R(t_1, t_1)) \ \text{AND} \\
\neg(\text{Happens} (\text{ir}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), t_2, R(t_1, t_1+20ms)) \ \text{AND} \\
\text{HoldsAt} (\text{valueOf} (\text{counterNotWithin}, _y), t_2) \\
\Rightarrow \\
\text{Initiates} (\text{ic}:\text{Operation}(\_\text{OperationID}_a, P_1,\ldots,P_m), \text{valueOf} (\text{counterNotWithin}, \text{add}(y, 1), t_2)); \\
\]
Assumption A4 updates the value of counterNotWithin, in case the service operation call has taken more than 20ms. Note that this condition is expressed through the negation of the occurrence of the service operation response event.

**Rule R1 – Monitoring the actual Guarantee Term**

\[
\forall t_1 \neg \text{Happens(ic:Operation(_OperationIDs, P1,\ldots,Pm), t_1, R(t_1,t_1)) AND} \\
\neg \text{HoldsAt(valueOf(counterWithin, _y), t_1) AND} \\
\neg \text{HoldsAt(valueOf(counterNotWithin, _y), t_1) AND} \\
\neg \text{greaterThan(add(_x,_y), 0) AND} \\
\neg \text{lessThan(add(_x,_y), 555.56) AND} \\
\neg \text{lessThan(div(_x, add(_x,_y)), 0.95)}
\]

Rule 1 monitors the actual Guarantee Term. In particular, it requires that when the arrival rate of requests for an operation is less than 555.56, than the average completion time of service operation calls should be less than 20ms in 95% of the cases.

![Architecture of Monitor Engine](image)

**Figure 26 – The architecture of monitor engine (stand-alone application)**

The CITY Monitor internal architecture is represented in Figure 26. The monitor can be used as a standalone application or as a software library. In the literature, the standalone application of the monitor engine is referred to as EVEREST (Event Calculus based Reasoning Toolkit, [65]).

Figure 26 represents the complete architecture of the monitoring engine in case it is used as a standalone architecture. The monitor engine is constituted by the following modules:

- **Event Receiver**: it receives events concerning the service execution in a standard XML format (see Appendix C for details of such event format);
- **Event Database Handler**: it provides a management interface to the database used to store events;
- **Monitor**: it is the actually reasoning engine, which checks the satisfaction of assumptions and rules on the basis of the received events;
- **Deviation Database Handler**: it provides a management interface to the database where violations of monitored properties are stored;
- **Monitor Manager**: manages the startup and termination of the monitor engine;
- **Behavioural Properties Extractor**: it is the generic components that feeds the monitor with EC-based rules and assumptions. The generation of rules and assumption is specific to the scenario in which the monitor engine is exploited;
- **Simulator**: it provides features for simulating events from the service system (for testing purposes);
- **Monitoring Console**: it is the GUI through which the violations of monitored properties can be checked.

Within the SLA monitoring framework, the monitor engine has been extended to be used as a library, which complies to the interface of the generic monitoring engine represented in Figure 26. In particular, the extensions of the original monitor engine have concerned the Monitor, Behavioural Properties Extractor, and Event Receiver internal components (see Figure 26).

- **Monitor**: It has been modified to implement the interface required by the Event Bus in the SLA Monitoring framework to push detected violations, i.e. monitoring result events, on the Event Bus.
- **Behavioural Properties Extractor**: Within the SLA Monitoring framework, this module has been substituted by the Rule Generator module, which is able to derive EC-based rules from the SLA that are submitted to SLA monitoring during service provisioning.
- **Event Receiver**: It has been modified to implement the interface required by the Event Bus to receive events pushed the Open Reference Case service.
4.2.2 Process Monitoring

The framework supports two kinds of monitors. Instance Monitors (IMs), which observe the execution of a single instance of a BPEL process; and Class Monitors (CMs), which report aggregated information on all the instances of a given BPEL process.

Within the FBK monitoring framework the monitor property must be specified in RTML, the Run-Time Monitor Speciﬁcation Language. RTML is rather expressive: it allows to specify IMs as well as CMs; moreover, it allows for specifying boolean properties related to the execution of processes, as well as statistic properties and time-related properties. Eventually the framework is able to translate automatically RTML monitor specifications into the Java code that implements the monitors, see [67] for details.

With RTML it is possible to deﬁne properties over the whole past history of collected events, using formulas such as \( f_1 \) Since \( f_2 \) (formula \( f_1 \) has been true since the last instant formula \( f_2 \) has been true) or Once \( f \) ( \( f \) has been true at least once in the past). Numeric information is obtained using built in functions such as count and time. Operator count is used to count the occurrences of an event. Since RTML closely integrates logical formulae and numerical functions, we can use count to count the occurrences of a complex behaviour, or decide to compare the values of multiple count functions using a boolean operator. time is used to compute the time span between events. Once again, this means we can use time to calculate the timespan between two complex behaviours, or compare timespans using boolean operators. Other functions are max, min, avg, and sum.
Their meanings are straight-forward. When defining a class monitor, RTML also provides additional class functions, such as \textit{Count}. These functions aggregate data over many different process instances. \textit{Count} allows to count how many times a certain event occurs in all the process instances belonging to the same process class. Similarly, function \textit{Avg} calculates the average of a numeric property evaluated across the different process instances. Class monitors predicate over events belonging to different instances.

In RTML the relevant basic events for process monitors are the creation/termination of a process instance and the input and output of messages. In some cases, it is preferable to speak of the effects of an event on the status of an interaction protocol, rather than of the event itself. The complete grammar for events $e$ is the following:

$$
e ::= \text{start} \mid \text{end} \mid \text{msg}(\text{link}.\text{input/output} = \text{msg}[\text{opt-constraints}]) \mid \text{cause}(\text{link}.\text{var} = \text{val}) \mid \text{cause}(\text{link}.\text{state} = \text{label})$$

The two kinds of monitors are reflected in the architecture of the monitoring engine depicted in figure 28. The monitor properties translation is provided by the Monitor Translator module meanwhile the monitor execution is provided by the Monitor Runtime Module (RTM). Indeed there are two distinct sets of monitor instances, namely IM instances and CM instances. In the RTM, the IMs and CMs are managed by two specific handlers, the IM Handler and the CM Handler.

A monitor is a Java class implementing both the IMonitor and the IProperty interface. More precisely, in respect to the IMonitor, IMs implement interface IProcessInstanceMonitor, while CMs implement interface IProcessClassMonitor. The IProperty interface has been specialized in the IBooleanProperty and IStatisticProperty. The time property is mapped using IStatisticProperty. The monitor class implements the proper property interface depending to the type of the monitor is going to monitor. The framework monitor interface hierarchy is depicted in figure 29.
Figure 28: Process Monitoring Engine / Framework
The IMonitor interface defines four methods which are common to all monitors: getProperty and getDescription return a short and a long description of the property that is monitored; getProcessName returns the name of the BPEL process the monitors are associated to; and status returns the current monitor internal status (e.g., RUNNING, ERROR, ...). The methods defined by interfaces
IProcessInstanceMonitor and IProcessClassMonitor manage the evolution of the monitors, and are better explained describing the life-cycle of instance and class monitors.

The IMs life-cycle is influenced by three events: the process instance creation, the input/output of messages, and the termination of the process instance. When the RTM receives the notification of the creation of a new BPEL process, it creates a set of monitor instances that are specific for that process instance. The monitor instances are initialized through the method init. When the RTM receives a message from the Mediator, it sends it to the Instance Monitor Handler which dispatches the message to all the matching monitor instances through method evolve. For each message, the Mediator provides also information on the process instance receiving/sending the message, as well as on the BPEL process specification corresponding to the instance. The information on the BPEL process is used to select the relevant set of monitors to be instantiated for that process. The process termination is captured via a termination event and dispatched to all the monitor instances associated to the process instance.

The life-cycle of a class monitor is quite different. Method init is called only once, when a single instance of the class monitor is created. The evolution of the class monitor is triggered whenever the RTM receives a message or an event is received from any instance of the BPEL process to be monitored. More precisely, after the Instance Monitor Handler has dispatched the event or message to the relevant monitor instance, and these have been updated, it signals to the Class Monitor Handler that also the class monitors have to be updated. The Class Monitor Handler invokes method update on all the different class monitors associated to the BPEL process, which can update their internal status.
Figure 30 shows the flow of interactions triggered by the reception of a message by the BPEL engine. When the message is received by the Queue Manager, a copy of it is forwarded to the Mediator. The Mediator marks the message with a time stamp and dispatches it to the RTM. The message is then passed to the Instance Monitor Handler which dispatches the message to the right IMs. The Instance Monitor Handler signals an update request to the Class Monitor Handler, which forces the update of the proper CMs. Figure 30 shows also that the CMs can interact with the associated IMs in order to update their state. Indeed, if a class monitor is responsible of collecting statistical data on all the instances of a given BPEL process, then it has to interact with the instance monitors that collect the non-aggregated values for the single process instances. This is achieved through the request/[value] flows shown in figure 30

![Figure 30: Monitor Interactions](image)

The Monitor Translator module translates automatically an instance RTML formula into the Java code implementing the monitor. The skeleton of the class in pseudo code is described in figure 31. The key element in the generation of the Java code implementing the property is map Val, which associates a value to all the sub-formulas of the property to be monitored. Function Init_RTML assigns initial (truth or numerical) values to these sub-formulas, while function Update_RTML updates these values according to a received event. The update is done compositionally on the structure of the formula, and exploits the old values of the sub-formulas that are stored in variable Val_old. In the case of operator time, value elapsed is elapsed time from last event relevant for the monitor. This value is computed using the time-stamps associated to the events sent to the monitors.

```java
package monitor.instance;
import org.astroproject.monitor.core.*;
import java.util.Map;

// Instance monitor for formula b
public class bFormulaInstanceMonitor implements IProcessInstanceMonitor, ...
{
    private Map Val; // Associates a truth value to each sub-formula
```
public void init() {
    Init_RTML(Val); // Initialize Val according to event "start"
}

public void evolve(BpelMsg message) {
    Map Val_old = Val;
    Update_RTML(Val, Val_old, message);
    // Update Val compositionally on the sub-formulas
    // according to event "message"
}

public void terminate() {
    Map Val_old = Val;
    Update_RTML(Val, Val_old, EVENT_TERMINATE);
    // Update Val compositionally on the sub-formulas
    // according to event "end"
}

public Object getValue() {
    return Val.get("b");
}

public String getProcessName() { ... }
public String getProperty() { ... }
public String getDescription() { ... }

Figure 31: Pseudo code instance monitor

Also for the class monitors we distinguish among Boolean formulas B and numeric formulas N. Most of the operators are identical to those of the instance monitor formulas. In the following the description of the operators specific of the class monitor formulas. Boolean formula And (b) checks if property b is true for all the instances of the BPEL process corresponding to the monitor. Numeric formula Count (b), instead, counts the number of instances of the BPEL process for which formula b holds. Numeric formula Sum (n) is similar, but aggregates numeric instance module formulas: it sums up the values of numeric formula n on all the instances of the BPEL process.

Also in the case of class monitors, we have implemented a translator from RTML to Java code. The key difference with respect to instance monitors is in the implementation of operators And, Count, and Sum. Indeed, these operators serve as link between class-level monitoring and instance level Monitoring.

The translation algorithm adopts the following approach:

- An instance monitor is generated for each instance property b or n that appears as argument of operators And, Count, and Sum in the class formula;
- One class monitor is generated, that aggregates the data of the instance monitors according to the formula.

The class monitor has the same structure as the instance monitor. In particular, it exploits map Val to associate a (boolean or numeric) value to each class-level sub-formula of the property to be monitored. Functions init and update are defined similarly to init and evolve in the instance monitor. The main difference is in the implementation of function Update_RTML.
Within the SLA monitoring framework, the FBK monitor engine has been extended to be used as a library. In the SLA@SOI context the monitor properties are specified within the SLA contract subscribed between the customer and the provider in the form of Guaranty Term. Once again we remind that the Guarantee Term is defined by a logic implication between the Qualifying Condition and the Service Level Objective; it means the Guarantee Term is violated when the Qualifying Condition is verified (logical value TRUE) and the SLO is not met.

```xml
<wsg:GuaranteeTerm wsag:Name="CompletionTime_PaymentService_Gterm" wsag:Obligated="ServiceProvider">  
  <wsag:ServiceScope wsag:ServiceName="PaymentService"/>  
  <wsag:QualifyingCondition>    
      <Expression>        
        <Predicate xsi:type="coremodel:LessEqual">          
          <OperandA xsi:type="terms:ArrivalRate" description="Non-Functional" context="PaymentService_request_ServiceProperty_arrivalRate_Variable" qualifier=""/>          
          <OperandB xsi:type="coremodel:Constant">            
            <type xsi:type="coremodel:Frequency" uniqueID="" value="5"/>          
          </OperandB>        
      </Predicate>    
    </coremodel:Expression>  
  </wsag:QualifyingCondition>  
  <wsag:ServiceLevelObjective>    
    <wsag:(KPITarget>      
      <wsag:KPIName>PaymentService_Gterm_KPI_CompletionTime</wsag:KPIName>      
      <wsag:Target>        
              <Expression>                
                <Predicate xsi:type="coremodel:LessEqual">                  
                  <OperandA xsi:type="terms:CompletionTime" context="PaymentService_ServiceProperty_completionTime_Variable" qualifier=""/>                  
                  <OperandB xsi:type="coremodel:Constant">                    
                    <statQ>percentile</statQ>                    
                    <statQValue>95.0</statQValue>                  
                  </OperandB>                
              </Predicate>            
          </coremodel:Expression>        
      </wsag:Target>    
  </wsag:KPITarget>  
</wsg:ServiceLevelObjective>
</wsg:GuaranteeTerm>
```
In order to enable the use of the FBK monitor engine each Guaranty Term must be translated in a RTML formula. Let us consider the property completion time defined for PaymentProcess in the SLA contract above. The property specifies that the PaymentProcess should complete in less than 70 ms in the 95% of the process instances. This property corresponds to the following RTML formula:

$$\text{Avg} \ (\text{time} \ (\text{lend Since} \ \text{start}) < 70) > 0.95$$

## 5 Service Manageability

The previous section describes the SLA monitoring solution designed for the software layer in the SLA@SOI project. This monitoring module can detect software SLA violations but does not always provide sufficient information for deciding the most appropriate response to a violation. Such decisions often require additional detailed information that explains why a violation has occurred and can, therefore, indicate the appropriate response action to it. This kind of enforcement capabilities are considered to be important in order to guarantee the SLA compliance at the software service layer.

In SLA@SOI, the adjustment activities are designed making use of autonomic management techniques, which support the robustness, selfhealing, self-managing, self-optimization, self-improvement of SOA components. In order to do this, the adjustment solution must interact with the different SOA applications composing the project framework. The Manageability Interface provides a harmonized view of the different service-oriented applications and an easy interaction between the Autonomic Management and the application specific instrumentation.

### 5.1 Service Manageability Infrastructure

Service Level Agreements (SLAs) are a common means to define the exact conditions under which services are provisioned by a service provider to a service consumer. A precondition for the specification of SLAs is that the service provider is able to predict quite accurately the quality parameters of his services. Consequently, he has to analyze and to manage his service-oriented IT landscape in order to get information about the properties of his provided services under a certain usage profile. Another requirement is to monitor the adherence to already existing SLAs. Hence, the service provider needs a framework to manage his SOA including the software and the infrastructure the software is running on. Existing SLA management frameworks typically focus on the level of singular service interfaces. They do not consider that realistic service provisioning scenarios
involve multiple stakeholders and layers of a business and IT stack as shown in the figure below [8].

**Figure 32: SLA/Service Stack**

Our project targets the provisioning of an SLA management framework that allows for consistent specification and management of SLAs in a multi-level environment. The framework allows integrating different SOA applications, which is ensured by the evaluation of the framework within various complementary industrial use cases. Each of these SOA applications possibly provides already existing manageability capabilities meaning they allow for monitoring and controlling non-functional properties of the application. Hence, the framework has to deal with the problem of heterogeneity concerning these service-oriented applications and their different management capabilities. The idea to solve that problem is to provide a unified management interface which acts as mediator between the monitored SLAs and the application specific instrumentation. Hereby, a harmonized management view on the different multi-layer service-oriented applications is created that can be easily integrated into the SLA management. A management view on an application is always related to the functional design of the application, meaning you cannot measure what you cannot describe. The Service Component Architecture (SCA) [10] is a convenient means to model the functional design of an SOA. SCA is a set of specifications which describe a model for building applications and systems using an SOA. The figure below shows the
This chapter introduces the design and implementation of unified management view through an appropriate Manageability Infrastructure. To hide heterogeneity and provide a common interface we evaluate several standards from the field of application management and adopt the most suitable.

The Manageability Infrastructure is part of a Manageability Framework that additionally comprises a Manageability Configuration Meta-Model and a methodology for designing manageable service-oriented applications. Both can be found in the A6.6 Deliverable.

### 5.1.1 Basics of Manageable Applications

This section lays the foundation for the development of the proposed solution. At first it introduces some basic aspects of management architectures from different point of views. Then an architecture approach is presented which serves as reference model for the manageability architecture designed in this chapter.

**Management Architectures**

Across all management disciplines various tools from different vendors are used. To ensure an efficient execution of the management tasks, an integrated management approach is necessary. This approach should build on unified interfaces and standards. Management architectures provide the foundation for these integrated management solutions. Because of their system and vendor independent character, management architectures enable interoperability...
between different specific components within a management infrastructure.

**Organization Model**

Aspects:
roles & responsibilities

**Function Model**

Aspects:
function domain and management functions

**Information Model**

Aspects:
structure and definition of MO in form of a MIB

**Communication Model**

Aspects:
access to MIB, management protocols

**Figure 34: Models of a Management Architecture**

The figure above shows the four (partial) models describing a management architecture [14].

**Information Model**

The information model is the basic part of management architectures since it describes the management relevant data. The managed object is the key element that represents an abstraction of an actual component. The information model must have the ability to cover all properties relevant from a management point of view. This includes the specification of how the object can be identified, what it consists of, how it behaves, how it can be manipulated, which relationships exist to other managed objects, and how it can be operated on through the management protocol. When accessing a managed object, a distinction is made between monitoring and controlling. Monitoring would constitute read access to the representation of a managed object. Controlling would be a write access to a managed object. Furthermore, the information model defines the syntax to describe a management object. The set of managed objects handled by a manager or an agent system is called management information base (MIB).

**Organizational Model**

The organizational model describes the actors, their roles, and the basic principals of their cooperation. For management systems a variety of different topological and functional arrangements exist. Central management means that one management system is responsible for all tasks. Multipoint control means that resources are combined into domains according to certain aspects, and a manager is assigned to each group. If the managers from the multipoint control approach are coordinated depending on the cooperation schema, either a multicenter control or a hierarchical management is produced. The network of managers is the most complex form since there is no clear-cut allocation of resources to management systems.

Currently there are two different forms of cooperation for management. On the one hand the asymmetrical-hierarchical manager-agent model where the
manager instructs the agent to execute a specific operation or to provide information. On the other hand, the completely symmetrical peer-to-peer approach which offers a flexible and reciprocal job relationship and an exchange of information in both directions.

**Communication Model**

The communication model defines the concepts for the exchange of management information between the actors. As a consequence, the communication model has to deal with the following aspects:

- Specification of the communicating partners.
- Specification of the communication mechanism.
- Definition of the syntax and semantics for the exchange formats.
- Embedding of management protocols into the service architecture or protocol hierarchy of the underlying communications architecture.

**Functional Model**

The functional model subdivides management into management function areas. A possible segmentation is the FCAPS model provided by the International Organization for Standardization (ISO). The FCAPS model consists of fault management, configuration management, accounting management, performance management, and security management.

**Manageable Application Systems**

This section describes the approach presented in [15], which introduces an approach to the development of manageable application systems based on management architecture-independent models. Figure 35 illustrates the resultant mappings.

![Figure 35: Mapping of Management Architecture Models to Models of Manageability Infrastructure](image)

Referred to the four submodels that describe management architectures (see previous section), the model of the Manageability Interface is aligned with the information model while the model of the Manageability Infrastructure architecture is aligned with the organizational and the communicational model of the corresponding management architecture. The functional model, which is less marked in most of the management architectures, slips in the information model.

A manageable application system comprises software components which meet the functional requirements and additional elements that enable the monitoring and controlling under certain management aspects. An additional interface of the application system provides a unified view on the distributed and in most cases heterogeneous system. In that way, the interface, called Manageability Interface,
provides meaningful management information to the management application. The Manageability Infrastructure includes the software components required to implement the Manageability Interface as well as the Manageability Interface itself. Figure 36 shows the general components in the context of application management.

Figure 36: General Architecture of a Manageable Application System

From the management infrastructure point of view, the Manageability Infrastructure is a partial component which provides application specific management information and functions. To assure purposeful management of the application system, its specifics in terms of functionality, mode of action, and structure have to be considered. Furthermore the technologies used to realize the application as well as the existing management environment impact on the development of a Manageability Infrastructure.

When designing a Manageability Infrastructure the following model artifacts have to be specified:

**Manageability Interface**
The description of the interface in the form of an application specific Manageability Model includes the specification of the managed objects as well as their corresponding information and functions, their behavior, and their relations.

**Manageability Infrastructure Architecture**
The design of the internal structure in the form of an architecture model comprises the splitting of the information and functionality defined in the Manageability Model to the elements of the Manageability Infrastructure. For each element its interfaces, its internal structure, and its dependencies have to be defined. Basically, two elements make up the architecture of a Manageability Infrastructure: the Management Agents and the Instrumentation. The Management Agents are the concrete implementation of the Manageability Interface. Each agent implements a specific part of the Manageability Interface. The management relevant information is stored in a local management information base (MIB) which is accessible to the management application by the respective agent. Access to the application components takes place by a specific interface which is implemented by the Instrumentation in the form of additional logic within the components.

**Instrumentation**
The Instrumentation is a solution to the question how the managed objects can be accessed in order to monitor and control them. Instrumentation is defined as devices or instructions installed or inserted into hardware or software to monitor
the operation of a system or component. We focus on the monitoring of services in service-oriented applications. In that case, the component that has to be monitored is the service or the service composition respectively. Figure 37 depicts the Instrumentation of a software component.

![Figure 37: Generalized Approach to Instrumenting Software Components](image)

The sensors are responsible for the monitoring of the service components while the effectors can control the service components. Both, sensors and effectors are accessible through a common instrumentation interface. This interface is used by the Management Agent responsible for this component. The current state of the component can thereby be saved in a local MIB. The Management Agent in turn provides a management interface that can be used by any management application.

Figure 38 illustrates an example of the three components of a Manageability Infrastructure introduced above.

![Figure 38: Refined Architecture of a Manageable Application System](image)

Since the design of the Manageability Infrastructure is intended to follow the model-based approach, first of all both models will be designed platform-independent (meaning independent of a certain management standard or architecture). In a second step the models are aligned with suitable management standards. These standards are evaluated in the following section.
5.1.2 Requirements

The proposed solution should provide a means that supports the service provider conducting his SLA-driven IT service management. The A1 deliverable defines a six-phase service-lifecycle: Service Design and Development, Service Offering, Service Negotiation, Service Provisioning, Service Operations and Service Decommissioning. In the following, the requirements for a Manageable Service Engineering Methodology and Manageable Services are identified and - where feasible - aligned with the phases defined in the service lifecycle.

1. In the Service Design and Development phase the solution has to support the service provider by providing the ability to configure the need of management information in the specificity that is necessary to estimate the required capacities e.g. hardware, and to allow the specification and the monitoring of SLA conditions. Since the functional design of the services is conducted by means of SCA and the monitoring specification is in most cases closely related to the functional design, the service provider has to be enabled to create a manageability design for the service components which is based on the meta-model of SCA.

R1: Ability to configure required management information

2. During the Service Design the service provider needs to extend his SCA-based functional application design of the services with the management configuration information defined. Therefore, the solution has to define rules, how the functional design has to be extended to supply the Manageability Infrastructure with the required management information.

R2: Guidelines for creating the Instrumentation

3. For service provisioning a concept is required for starting new instances of the manageability infrastructure in conjunction with the new instances of the application itself.

R3: Provisioning mechanism for new manageability infrastructure instances

4. Within the Service Operations phase the solution has to gather the runtime information supplied by the different components at the different levels of the application. Since the different components deliver the information in arbitrary formats, the solution has to provide a unified management view on this information in order to offer a simple and meaningful interface to the SLA management component within the framework or any other management application.

R4: Unified management view

5. By gathering historical and statistical information about the quality parameters of an executed service, the service provider is able to calculate the guaranteed service quality. This information can also be used to derive the quality requirements each component of the service stack has to fulfil in order to provide a specific service quality of a composite service.
R5: Durable access to management information

6. In order to enable SLA adjustment the Manageability Infrastructure has to provide a possibility to detect problematic components within the service stack. Thus, in case of SLA violations, the service provider is able to locate the cause of failure and the responsible stakeholder can be identified and a faster bug fixing can be guaranteed.

R6: Ability to track executed service transactions throughout component/technology stack

7. Since the provided management information is used by multiple management applications from different vendors, the Manageability Infrastructure has to deal with the requirement for a common interface. From this it follows that the Manageability Infrastructure should be based on appropriate standards in order to be easily integrated in existing management applications.

R7: Standard-based manageability interface

R1 is particularly addressed by the manageability configuration metamodel presented within scope of the WP A6 deliverable. Everything else has to be supported by the manageability infrastructure we present in the following.

5.1.3 State of the Art

Management Standards

To provide a standard-based Manageability Infrastructure and a harmonized SOA Manageability Interface we have to deal with various existing management standards. The objective is to spot the standards that have a potential field of application within the service-oriented Manageability Infrastructure including the Manageability Data Model. The value of standards at the development of manageability aspects of an application is shown in [17]. The conclusion of this paper is that different applications have benefits depending on the nature of the application they are managing and the environment in which they are set. Another result is that many applications require multiple technologies or standards to fully satisfy their operational management needs. Therefore, this section introduces some existing and emerging management standards, and rates the value of the particular standards for the Manageability Infrastructure.

Web-Based Enterprise Management (WBEM)

Web-Based Enterprise Management (WBEM) represents a set of management and Internet standard technologies developed by the Distributed Management Task Force (DTMF). The goal of these standards is to unify the management of distributed computing environments [18]. WBEM comprises the following components [18][19]:

- Common Information Model (CIM): CIM provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. CIM’s common definitions enable vendors to exchange semantically rich management information between systems throughout the network.
The CIM schemas provide the actual model descriptions. Schemas are sets of CIM classes that represent a particular management domain.

- The CIM Specification defines the details for integration with other management models.

- **Managed Object Format (MOF):** The MOF is a formal description of the classes and associations used in CIM schemas.

- **CIM-XML:** A protocol that uses XML over HTTP to exchange Common Information Model (CIM) information.

- **CIM Query Language (CQL):** A query language that is used to select sets of properties from CIM object instances.

- **CIM Object Manager (CIMOM):** The database where the instances of the CIM classes are stored. A CIMOM is the central point for accessing management resources.

- **WS-Management:** Web Services for Management (WS-Management) [20] provides a common way for systems to access and exchange management information across the entire IT infrastructure. By using Web services to manage IT systems, deployments that support WS-Management will enable IT managers to remotely access devices on their networks. WS-Management allows the management information in the CIM to be exposed in a Web services environment.

The WBEM standard supports the modelling of management information since it contains the CIM, which is described on basis of the MOF. Because MOF is similar to UML it is easy to understand and easy to use. CIM provides an information model for the management of applications and thus could be used as information model for the manageability architecture. Furthermore there are several implementations available e.g. Open WBEM, MS-WMI, and Solaris WBEM, which support the development of the manageability infrastructure.

**Web Services Distributed Management (WSDM)**

The WSDM standard was submitted to the Organization for the Advancement of Structured Information Standards (OASIS), with version 1.0 ratified in March 2005, and version 1.1 ratified in August 2006 [21]. It seeks to unify management infrastructures by providing a vendor, platform, network, and protocol neutral framework for enabling management technologies to access and receive notifications of management-enabled resources [22]. The WSDM standard specifies how the manageability of a resource is made available to manageability consumers via Web services. It consists of the following sub standards [22]:

- The Management Using Web Services (MUWS) standard deals with the basic mechanisms and Message Exchange Patterns (MEPs) for managing any manageable resource using web services as the platform for exchanging messages.

- The Management of Web Services (MOWS) standard addresses the management of a Web service itself, i.e. a web service is the manageable resource. MOWS may be viewed both as application of the WSDM MUWS standard and as an extension of the WSDM MUWS standard.

WSDM is built upon W3C and other OASIS standards like XML, SOAP, WSDL, WS-Addressing, WS-Resource Framework and the Message Exchange Patterns (MEP).
WSDM-MUWS expresses how to provide access to manageable resources via web services. Thus, it supports the interoperability between manageable resources and manageability consumers. WSDM-MOWS allows the modeling of atomic web services as managed resources, but there are no details on the composition of a manageable resource. So WSDM does not describe a comprehensive information or functional model and also does not provide an instrumentation technology. With Apache Muse a Java-based implementation of the standard specifications exists, that could be used when implementing the Manageability Infrastructure. Nevertheless, an additional standard e.g. WBEM-CIM is necessary to describe the manageable resources.

**Java Management Extensions (JMX)**
The Java Management Extensions (JMX) technology was developed by Sun Microsystems and has been part of the Java Platform Standard Edition since the Java 2 Platform, Standard Edition (J2SE) 5.0 Release. JMX is used to manage resources such as applications, devices and services with the Java Programming Language. Because of its dynamic structure JMX can be used to monitor and manage resources as they are created, installed and implemented [23].

The architecture of the JMX Technology consists of three levels:

- The instrumentation of the managed resources is implemented by Java objects known as Managed Beans (MBeans).
- A JMX agent which manages the MBeans and thus directly controls the resources and makes them available to remote management applications.
- The management application outside the agent's Java Virtual Machine is the third level of the JMX architecture.

Implementations of the JMX specification can be found in nearly every application server based on Java technology (e.g. JBoss, BEA Weblogic, IBM Websphere and Apache Tomcat).

Since JMX is a platform-dependent standard, the managed resources have to support Java technology. Nevertheless, JMX is independent of any operating system. It does not provide an information model but for example a WBEM-CIM model could also be implemented in JMX. The existing implementations support the realization and the management of the agents.

**GLUE**
The GLUE specification is provided by the GLUE working group and is currently available in version 2.0. The specification includes a conceptual information model for Grid entities described using natural language and enriched with a graphical representation using UML Class Diagrams [24]. The group also provides renderings for concrete data models such as XML Schema, LDAP Schema and SQL [25].

Based on abstract entities like domain, service, endpoint manager, resource, and others, the conceptual model for a computing service to share computational capacity in a Grid environment and a storage service to share storage capacity in a Grid environment are introduced by the group's GLUE information model.

With GLUEMan [26] there is a framework available to manage information providers for GLUE 2.0. It also provides a client enabling to query GLUE 2 information and render them according to the available mappings GLUEMan is based on Open Pegasus [27] an open-source implementation of the DMTF CIM and WBEM standards in C++. 
The GLUE information model covers the functional aspects of a Grid environment. However, the manageability aspects are not covered by the model. So GLUE could be used as an extension to CIM in Grid environments.

**Service Modeling Language (SML)**

The Service Modeling Language (SML) is the successor of Microsoft's System Definition Model (SDM). The SML specification was developed by an Industry Group and is currently being standardized in a W3C working group established to generate W3C Recommendations.

According to [28], SML provides a set of constructs for creating and validating models of complex services and systems. These models might contain information like policy, deployment, configuration, service level agreements, etc. and are described in a technically uniform XML format.

In SML a model is realized as a set of interrelated XML documents. Thereby a distinction is made between the definition documents and the instance documents. The definition documents describe the abstract structure of the model as well as most of the information a model validator needs to decide whether the model, as a whole, is valid. The instance documents support the description of the individual resources represented by the model. To interchange SML models between different systems the Service Modelling Language Interchange Format standard was introduced at the same time. The purpose of SML-IF is to package the set of documents that form an SML model into a standard format so that they can be exchanged in a standard way [29].

The Eclipse Foundation provides an open source reference implementation of the SML specification called COSMOS (Community-driven Systems Management in Open Source) [30]. However, the implementation is in a very early phase yet.

The SML specifications are situated in an early development stage (last call version). Furthermore, SML is a domain-neutral language and does not define any management specific feature. Thus, it can be seen as a layer above the information model (e.g. CIM), describing the desired state, interconnected relationships and management policies of the distributed system.

**Common Model Library (CML)**

The Common Model Library (CML) is specified by the CML working group, a consortium of eleven leading technology companies. The specification is based on the W3C Service Modeling Language (SML) and defines common expressions of, and the semantics for, concepts which enable information exchange between both management tools and managed resources [31]. Currently, the consortium is working on an initial draft of the specification, i.e. there is no concrete specification available yet, but CML will support a multitude of management disciplines including service desk, configuration management, performance management, systems management, SLA management and more. It will also facilitate the sharing of information across a typical set of lifecycle phases which include planning, development, deployment, and operation phases [32].

CML looks very promising and probably will provide a suitable information model for a Manageability Infrastructure. However, there is neither a specification nor an implementation available so far.

**Application Response Measurement (ARM)**
ARM was standardized by the Open Group and is currently available in version 4.0V2. The standard describes a common method for integrating single-system and distributed applications as manageable entities. ARM provides an API that enables comprehensive end-to-end management including measuring application availability, application performance, application usage, and end-to-end transaction response time [33]. Applications using ARM define important transactions within the application. Transactions initiated by a user and transactions with servers are typical examples. When transactions start and/or stop, applications on clients and/or servers call the ARM API. The ARM management agent collects the measurements associated with the transactions and communicates with management applications that provide analysis and reporting of the data [34]. ARM is related to the CIM standard presented above since ARM and CIM’s Metrics Model are based on the same standards. Thus, the CIM Metrics Model can be used to represent data measured with ARM [35].

ARM describes a simple and thus widespread procedure for the monitoring of transactions in application systems. However, as it is restricted to the monitoring of transactions it is not sufficient to support a comprehensive Manageability Infrastructure. From this it follows that is has to be extended by other approaches. Furthermore, the information provided by ARM is also covered by the Metrics Model in WBEM-CIM. Nevertheless, it describes useful approaches for the design of the Instrumentations.

**SLA-Driven Manageability / Management Frameworks**

The proposed solution is designed for being used within an SLA management infrastructure. Two major frameworks for a SLA-based management of service-oriented applications have been presented in literature. In [36] a solution for SLA-driven management based on Web Service Level Agreements (WSLA) is introduced. However, this solution mainly focuses on the management of atomic Web services and does not adequately support Web service compositions. [37] extend the approach by a WBEM-based monitoring infrastructure. In the paper, SLAs are mapped to an extended CIM metrics model. In summary, both approaches do not provide a common model for the mapping of managed resources and do not describe how a unified view on heterogeneous applications is rendered possible. Thus, the aforementioned approaches could integrate the contributions of this task in order to provide a more comprehensive framework.

The solution presented in [38] is competing to our solution, since it explicitly supports the SLA-based monitoring of services and service compositions. The introduced manageability infrastructure includes an information model based on an ITU-T standard. However, the solution relies on the Hewlett-Packard (HP) Process Manager and the manageability interface is not built on standards. This complicates the integration with other management applications. Another competing solution is presented in [39]. This paper addresses an integrated, tool-based management system to support the tasks of service level management. A manageability architecture including an information model on the basis of the Shared Information/Data Model (SID) is presented. However, the correlation of instrumentation and manageability infrastructure is not considered in this work. Moreover, there is no standard-based interface.

Another approach to the development of a management infrastructure is presented in [40]. However, the scope of this infrastructure is limited to the monitoring of atomic services. In [41] a manageability infrastructure is provided
that builds on a proprietary workflow engine. However, the solution does not meet the requirements of having an interoperable and standard-based instrumentation and manageability interface. [42] presents an architecture that allows monitoring and recovery of BPEL processes. However, the solution is limited to BPEL processes and does not allow the integration of the whole SOA stack. The approach described in [43] uses SCA to integrate manageability modules in an SOA. The architecture allows the integration of multiple management tools and technologies. Our solution benefits from the ideas presented in this paper when developing the architecture of the manageability infrastructure.

In [44] an approach is presented that partially relates to this project. The solution presents extensions to WBEM/CIM for an active monitoring of service composites, including a rule-based monitoring. However, the design of the manageability infrastructure is not the major concern of the paper and therefore it is kept at a minimum. For example, the information model is limited to the CIM_UnitOfWork class and does not allow the monitoring of internal actions or incoming and outgoing messages. [44] describes an approach from which this project can benefit when creating the instrumentation models and the configuration meta-model. The paper describes how the CIM Metrics Model can be used in order to correlate the performance measurements within a distributed application.

### 5.1.4 Standard-independent Design of the Manageability Infrastructure

According to the approach presented in section 5.1.1, the design of a Manageability Infrastructure comprises three major components:

- Instrumentation
- Management Agent
- Manageability Interface

This section introduces in detail the design of Manageability Infrastructure. As shown in figure 39, the business components in an SCA composite have to provide an additional management service interface. This interface is implemented by the Instrumentation (I) and is used by the Management Agent Composite, which implements the Manageability Interface (M) of this Managed SCA Composite. The Manageability Interface of the SCA composite can then be used by a management application or another manager component that aggregates multiple Manageability Interfaces.

![Figure 39: Managed SCA Composite](image)

Since, the Management Agent and the Business Component are designed in terms of an SCA composite, they can be made up by multiple components. Figures 39 and 40 illustrate two sample architectures of a Manageable SCA Application.
In figure 40, one Management Agent is responsible for all Instrumentations (I) of the service-oriented application. Hence, the management application can directly use the Manageability Interface (M) of that Agent to receive the management information captured by the Instrumented Functional Composites.

The example in figure 41 uses three Management Agents, one agent for each Instrumented Functional Composite and one agent that aggregates the Manageability Interfaces (M) of the other Management Agents in order to provide a single interface to the management applications. The meta-model in figure 42 describes the general structure and the relationships between the involved
A ManagementApplication uses one or more ManageabilityInterfaces to monitor the (functional) components. These Manageability Interfaces in turn are implemented through one or more ManagementAgents. Thereby, a ManagementAgent can be subdivided into multiple other ManagementAgents, where each agent is responsible for a certain part of the ManageabilityInterface. In order to make the management information permanently available, each ManagementAgent contains a DataRepository. To gather the management information, provided through the ManageabilityInterface, the ManagementAgent communicates with one or more CompositeInstrumentation interfaces. According to the concept of SCA, the instrumentation service interface of an SCA Composite can aggregate the instrumentation service interface of one or more SCA Components or other SCA Composites.

In the following, the design of the three elements of a Manageability Infrastructure according to this general architecture is introduced. After a short description of each component, the design is presented. Where applicable, we used existing design patterns [45].

**Instrumentation Design**

The Instrumentation allows the Manageability Infrastructure to monitor and control the functional service components.
According to the Engineering Methodology for Manageable Service Components introduced in the A6 deliverable, the development of the actual Instrumentation has to be taken over by an Instrumentation Developer subject to a certain Functional Design (see figure 43). However, this section introduces guidelines and rules for the development of an actual Instrumented Functional Design.

The Instrumentation Developer gets the information on how and where he has to extend the Functional Design from the corresponding Manageability Model created by a Manageability Designer. This allows capturing the specified management information during service execution by the Running Instrumented Service Component. This information contains runtime information about each single instance, e.g. start/end time, as well as the information about the relation between the instances, e.g. service component instance X operation 1 calls service component instance Y operation 2. The Running Instrumented Service Components is the sensor that delivers the management information to the Running Manageability Infrastructure where it is processed, and persisted in the Manageability Data Model. Since we focus on monitoring aspects in year 1, there is no need for effectors so far. The Instrumentation Guidelines presented in this section concern the communication between the Management Agents within the Manageability Infrastructure Design and the Instrumented Function Design.

Design Decisions
The Instrumentation enables the management agents to access the functional components of an application system. Because of that, the logic which processes the information can either be located in the Instrumentation or in the management agent. This concerns especially the question to what extent the data, supplied by the Instrumentation, conforms to the standard format defined in the information model. More precisely, either the Instrumentation converts the data into the format required by the information model, or the Instrumentation sends raw management data and the agent has to take care of the transformation (e.g. if the Instrumentation already exists).

A further question concerns the communication between management agent and Instrumentation. Either the Instrumentation passively provides an interface and the management agent actively reads the management information from this interface in a certain time interval, or the Instrumentation actively notifies the passive management agent when new management information is available. Thereby the question occurs whether the Instrumentation should send the new data within the notification event (Push Model [46]) or the event does not contain any information and the management agent has to read the new or changed data from the instrumentation interface (Pull Model [46]).

A final issue is related to the communication between different instrumented service components. The question is, whether the different Instrumentations have to communicate with each other in order to provide the information management agents require. For example, this is the case when the Instrumentation has to keep track of the call hierarchy of multiple operation calls in a transaction.

Design
Following the active instrumentation pattern presented in [15], our instrumentation (see section 4.1) is designed as an active component within the Manageability Infrastructure, i.e. on the one hand the Instrumentation notifies the management agent when new management information is available, and on the other hand the Instrumentation decides how much information is delivered.
This approach is used because there is no system-burdening polling through the management agent and the management agent is always up-to-date regarding the status of the service components.

**Communication between Instrumentation and management agent**

The communication between Instrumentation and management agent is handled by a message bus [46] as described in section 4.2. The management data is delivered within a message. In the following, we introduce a conceptual event data schema, which specifies the information required by the Management Agent to fill the Manageability Data Model. Based on this event format we created a concrete extension of the event specification used by the Monitoring Bus and Instrumentation presented in chapter 4.1. Figure 44 depicts the basic structure of the conceptual event schema.

![Figure 44: Basic Structure of Events](image)

The `eventId` is a unique identifier of the event. The `eventSourceDefinitionId` is the identifier of the managed element which caused the event, while the `eventSourceInstanceId` is the identifier of the concrete instance of that managed element. The `eventSourceParentDefinitionId` and the `eventSourceParentInstanceId` identify the parent managed element of the managed element that caused the event. The provisioning of this parent information is discussed later in this section. The `eventSourceUser` attribute is a string which specifies the user who started the actual transaction. The `eventType` is an abstract type which is extended by the following concrete event types.

![Figure 45: ServiceOperationCall Events](image)

*ServiceOperationCallStartEvent* indicates the start of a service operation call. It consists of one start time and one or more incoming messages. *ServiceOperationCallEndEvent* indicates that a service operation call is finished. It consists of the end time and optionally the outgoing message and the completion state.
Figure 46: ReferenceOperationCall Events

The reference operation call event types have the same structure as the service operation call event types described above.

Figure 47: Loop Events

LoopStartEvent and LoopEndEvent indicate the start or the end of a loop within the application logic of the service component. Besides the start or end time, it is possible to transfer the current status of the loop’s condition.

Figure 48: Conditional Flow and Variable Events

The event types ConditionalFlowDataEvent and VariableDataEvent announce that there is new or updated data available concerning the status of a condition of a conditional flow or the content of a variable respectively.

Logic within the Instrumentation

From this it follows that the Instrumentation has to include as much logic as necessary to provide the data in the format described above. However, most of this information is raw data that can be captured directly. The only exception is the information about the parent instance, when a sequence of managed element instances is executed. This information is required by the management agent in order to provide the possibility to track the actual executed instances of managed elements within a transaction. The agent needs the instance information because the static information regarding the relationship between several components is not sufficient, since there are multiple paths of managed elements, and the actual
path of managed elements (e.g. operation calls) is determined during runtime. Since a service-oriented application can consist of multiple systems using different technologies, multiple instrumentation components might exist. These Instrumentations have to communicate among each other in order to provide the parent instance information.

**Communication between different Instrumentations**

Since a service-oriented application consists of multiple components on different levels, distributed over multiple systems, the particular sensors have to keep track of the parent/child relationship between the instances in order to deliver this information to the Manageability Infrastructure.

![Communication between different Sensors](image)

In doing so, the Manageability Infrastructure is capable of retracing the service transactions and thus provides the required analyzing mechanism. Concerning the interaction mechanism between the different instrumentation components, the Instrumentation Developer can decide whether the information exchange is done by putting the information in a meta-element (e.g. the SOAP header) or by extending the operations with a variable that contains the information. Moreover, it is possible to use both approaches within one service-oriented application. The use of a meta-element has the advantage that the actual operations of the services remain unchanged. The disadvantage of this approach is that it is often not possible, or coupled with high effort, to put the information in a meta-element. The advantage of extending the operations with a meta-variable is in the simpler realization, while the drawback of this approach is that the services are then specific to a certain management configuration, which contradicts the loose-coupling given by an SOA. However, this decision has to be taken by the Instrumentation Developer in dependency of the given scenario.

**Example**

This section provides a sample instrumentation design. Figure 50 illustrates an appropriate instrumented functional design for an abstract service-oriented
The service components are extended with the ability to send messages in the form of events. The number and type of events is specified with regard to the requirements defined in a corresponding Manageability Model (see WP A6 Deliverable). According to this, $E_1$ and $E_{11}$ indicate the start and the end of service operation call $SOC_{Op-A}$. The corresponding events are depicted in figure 51.

![Sample Instrumentation Design](image)

**Figure 50: Sample Instrumentation Design**

```xml
<Event>
  <eventId>E0001</eventId>
  <eventSourceDefinitionId>SomeServiceComponent_SOC_Op-A_Def</eventSourceDefinitionId>
  <eventSourceInstanceId>Inst_0001</eventSourceInstanceId>
  <eventSourceUser>Jack Beauregard</eventSourceUser>
  <eventType xsi:type="ServiceOperationCallStartEvent">
    <startTime>125408709</startTime>
    <incomingMessage>x param: Who's there?"</incomingMessage>
  </eventType>
</Event>

<Event>
  <eventId>E0123</eventId>
  <eventSourceDefinitionId>SomeServiceComponent_SOC_Op-A_Def</eventSourceDefinitionId>
  <eventSourceInstanceId>Inst_0001</eventSourceInstanceId>
  <eventType xsi:type="ServiceOperationCallStartEvent">
    <endTime>123456789</endTime>
    <outgoingMessage>"Nobody"</outgoingMessage>
  </eventType>
</Event>
```

**Figure 51: Sample OperationCall Events**

The Instrumentation delivers the management information, which is defined in the Manageability Model, to the Manageability Infrastructure. This always includes
the unique identifier of the event \((eventId)\) and the unique identifier of the managed element instance \((eventSourceDefinitionId)\) together with \((eventSourceInstanceId)\). In the example above, the name of the user who started the BPEL operation is also provided. Moreover, the \(startTime\) and \(endTime\) as well as the incoming and outgoing message, for the service operation call instance of \(op-A\) of the service component \(SomeServiceComponent\), are delivered to the Manageability Infrastructure. A further example is illustrated in figure 52. It shows the event which is sent when the value of variable \(SomeVariable\) is changed \((E10)\).

![Figure 52: Sample VariableData Events](image)

In that case, the Manageability Designer defined that, in addition to the unique identifiers of the current managed element instance (which is an instance of \(SomeVariable\)),

- the content assigned to the instance,
- and the parent-child relationship between the instances of managed element \(SOC_Op-A\) and the instances of managed element \(SomeVariable\),

are relevant for SLA management and have to be delivered to the Manageability Infrastructure.

Furthermore, the needed communication between the Instrumentation of \(SomeServiceComponent\) and \(AnotherServiceComponent\) is realized through an extension of operation \(op-B\). A parameter called \(instrMetaData\) of type \(Object\) is added by the Instrumentation Developer. This is because the Manageability Designer defined in the Manageability Model (figure 43) that the Instrumentation has to deliver information about the parent-child relationship between reference operation \(ROC_Op-B\) of \(SomeServiceComponent\) and service operation \(SOC_Op-B\) of \(AnotherServiceComponent\). Within this parameter the unique identifier of the current instance of \(ROC_Op-B\) can be transferred to \(SOC_Op-B\), so that the parent information can be used when creating event \(E5\).

**Manageability Interface**

The Manageability Interface enables management applications to access the management information of a service-oriented application. In this context, a Manageability Data Model, which includes the specification of the managed elements as well as their corresponding information and functions, their behaviour, describes the management information and their relations. The Manageability Interface provides operations for creating, reading, updating,
deleting, and searching instances of the Manageability Data Model.

The Manageability Interface is part of the Manageability Infrastructure Design as well as the Manageability Data Model. The Manageability Data Model can be derived from the Manageability Configuration Meta-Model presented in the A6 Deliverable. It is placed on a lower level of abstraction and describes an executable database schema, which allows storing the information specified in the instances of the Manageability Configuration Meta-Model. The Manageability Data Model is influenced by the Common Information Model (CIM) standard (see section 5.1.3), which offers a common representation of a management information model. Moreover, it includes some well-defined and well-known concepts that can be reused when developing the Manageability Data Model. Nevertheless, in this chapter, the Manageability Data Model is designed in a standard-independent way. This basic Manageability Interface provides the full capability of the Manageability Infrastructure. However, the design allows creating multiple standard adapters that use the basic Manageability Interface to provide standard-conform versions of the Manageability Interface.

**Design Decisions**

The Manageability Data Model has to cover all aspects that can be defined by the Manageability Configuration Meta-Model. This, however, includes no guidance regarding the required granularity of the Manageability Data Model. The granularity can either be completely generic, or aligned to certain specific service-oriented applications, or somewhere between the two aforementioned grades. A more generic design has the advantage that the data model is simpler and clearer (in particular concerning the semantics), whereas a fine-grained granularity is more meaningful and thus easier to use.

The associations between different managed elements have to be persisted in the Manageability Data Model and exposed through the Manageability Interface. This leads to the question how the data model should map these associations. They can either be embedded in the entities that are part of the association, or modeled in a separate association class. Whether a separate association class should be employed or not, depends on the usage and the significance of associations in a scenario.

Besides defining which management information must be captured during runtime by the Instrumentation, the Manageability Configuration Meta-Model allows to specify static management meta-information. This static management meta-information is equal for all instances of a managed element. A simple approach to map this information to the data model would be the integration of both, the static information and the runtime information of a managed element, into one class. However, this approach has the drawback that the static
information is stored redundantly in each instance of the managed element. Another approach would be the modelling of two classes, one for the static information and one for the runtime information. This approach increases the number of classes in the model, but avoids the redundant storage of the static information.

As for the Manageability Interface, the amount and type of operations decide on the granularity (figure 54).

<table>
<thead>
<tr>
<th>fine-grained operations</th>
<th>coarse-grained operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>getobjectA()</td>
<td>getObject(String name)</td>
</tr>
<tr>
<td>getobjectB()</td>
<td></td>
</tr>
<tr>
<td>getobjectC()</td>
<td></td>
</tr>
</tbody>
</table>

Figure 54: Granularity of Operations

Many fine-grained operations increase the usability, but require more design and implementation effort. Less coarse-grained operations shift the effort, e.g. for creating queries, to the user of the Manageability Interface. In that case, the user needs more knowledge about the data model, but this fact reduces the usability.

Design

Granularity of objects in the Manageability Data Model
The information model provided by CIM uses relatively coarse-grained objects since all the management information is mapped to the unit of work elements. However, the Manageability Data Model should provide a more meaningful view on the managed elements, while not being limited to certain service-oriented applications. Hence, the Manageability Data Model benefits from the structure of CIM’s unit of work concept, but splits the unit of work class in more meaningful classes with additional attributes. This eases the mapping from the Manageability Configuration Meta-Model to the Manageability Data Model. Since we focus on monitoring the performance of actions within or across SCA service components, all managed elements can be modelled as service component actions. The separation of a service component action into more specific actions as introduced in the Manageability Configuration Metamodel (see A6 Deliverable), for instance internal action and operation call, is not carried over to the Manageability Data Model in order to keep the data model clear and avoid complex database queries.
Figure 55 shows the design of the Manageability Data Model.

**Separation of static and dynamic management information**

The concept of storing static and dynamic information in separate classes (execution and definition classes) is adopted by the CIM Metrics Model described in section 5.1.3. The `ServiceComponentActionDefinition` and `ServiceComponentAction` are the central classes of the Manageability Data Model depicted in figure 55. The `ServiceComponentActionDefinition` class specifies the semantics and usage of a service component action (its meta-data) while the `ServiceComponentAction` class contains data values captured for a particular instance of the definition class. An instance of a `ServiceComponentAction` should exist only when a definition of its characteristics is present. The `ServiceComponent` class is the logical element which contains the actions. The Manageability Data Model is designed in a way, so that dynamically (i.e. at runtime) associating both `ServiceComponentActions` and their definitions with a `ServiceComponent` is possible. At this point, describing the several attributes of the classes is renounced, since the meaning is the same as in the Manageability Configuration Meta-Model described in the A6.6 Deliverable.

**Associations**

The model uses association classes to describe associations. This eases the mapping of the data model to the CIM standard, since in CIM the only way to describe associations are association classes. Moreover, it simplifies querying associations in the database. The `SubServiceComponentActionDefinitions` class and the `SubServiceComponentActions` class express a one-to-many relationship between `ServiceComponentActionDefinitions` or `ServiceComponentActions` respectively. These associations describe the relationships between different instances of the respective class. For example, the associations between different `ServiceComponentAction` instances describe the execution path of these actions within a mutual context (e.g. a service operation call executed by a customer).

The `ExecutedServiceComponentActions` class describes a one-to-many relationship between `ServiceComponentActionDefinition` and the corresponding running or completed execution units (instances of `ServiceComponentAction`).

The `IncludedActions` association class relates the action elements of a service component (`ServiceComponentActionDefinition`) to the respective `ServiceComponent` instance.
Granularity of operations provided by Manageability Interface

The Manageability Infrastructure exposes the gathered management data, stored in the Manageability Data Model, through a Manageability Interface. The interface follows the ‘Façade’ pattern [45] and provides methods to create, read, update and delete managed object instances of the data model. Thereby, it follows the structure of the CIM standard, but also provides additional operations not covered by CIM. In CIM the components responsible for the provisioning of these operations are called provider. A provider is defined as an executable that can return or set information about a given managed element [48]. The components that realize the provider functionality within the Manageability Infrastructure are depicted in figure 56. The interfaces of these providers form the Manageability Interface.

**Figure 56: Manageability Interface Design**

The *ManagedObjectReader* component provides an interface where single objects as well as a list of all existing managed objects can be read. Furthermore, some predefined queries, like reading all service component actions that share a mutual context ID, are offered. Moreover, there is a subscribe operation where management applications can subscribe for a given event. This approach follows the ‘Publish/Subscribe’ pattern presented in [47]. The *AssociationReader* component provides operations to get the associations associated with a given managed object, for example, reading all executed service component action instances of a given service component definition instance. *AssociationReader* and *ManagedObjectReader* have in common, that they are both able to return the concrete object instances or only the unique identifier of the respective instances. The *DefinitionUpdater* component enables write access to the definition objects. It provides operations to add and update definition object instances. The *Eraser* component offers the possibility to remove particular managed objects. Moreover, it provides operations to remove all execution instances and to reset the whole management repository. Since the interfaces illustrated above can be implemented by multiple Management Agents the Manageability Interface acts as a facade.

**Example**
This section provides an excerpt of an instantiated Manageability Data Model. The following tables include sample instances of the classes `ServiceComponentActionDefinition` and `ServiceComponentAction`.

### ServiceComponentActionDefinition

<table>
<thead>
<tr>
<th>definitionId</th>
<th>context</th>
<th>name</th>
<th>description</th>
<th>binding</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SomeServiceComponent_SOC_Op-A_Def</td>
<td>someContext</td>
<td>-</td>
<td>Some description.</td>
<td>-</td>
<td>ServiceOperationCall</td>
</tr>
<tr>
<td>SomeServiceComponent_SomeCondFlow_Def</td>
<td>-</td>
<td>-</td>
<td>Another Description.</td>
<td>-</td>
<td>ConditionalFlow</td>
</tr>
<tr>
<td>SomeServiceComponent_SomeVariable_Def</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Variable</td>
</tr>
<tr>
<td>SomeServiceComponent_ROC_Op-B_Def</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ReferenceOperationCall</td>
</tr>
<tr>
<td>AnotherServiceComponent_SOC_Op-A_Def</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ServiceOperationCall</td>
</tr>
<tr>
<td>AnotherServiceComponent_SomeLoop_Def</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Loop</td>
</tr>
</tbody>
</table>

### ServiceComponentAction

<table>
<thead>
<tr>
<th>instanceId</th>
<th>definitionId</th>
<th>status</th>
<th>user</th>
<th>mutualContextId</th>
<th>startTime</th>
<th>endTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst_0001</td>
<td>SomeServiceComponent_SOC_Op-A_Def</td>
<td>Completed</td>
<td>Jack Beauregard</td>
<td>mutCtx_0001</td>
<td>T23456789</td>
<td>T23456800</td>
</tr>
<tr>
<td>Inst_0001</td>
<td>SomeServiceComponent_SomeVariable_Def</td>
<td>-</td>
<td>-</td>
<td>mutCtx_0002</td>
<td>T23456900</td>
<td>T23456900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elapsedTime</th>
<th>incoming Messages</th>
<th>outgoing Message</th>
<th>variable Content</th>
<th>loopCount</th>
<th>condStatus</th>
<th>lastLoop ElapsedTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>param::Who’s there?</td>
<td>Nobody</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>I don’t know</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The definition object instances in the table `ServiceComponentActionDefinition` are in accordance with the sample instance of the Manageability Model presented in the A6.6 Deliverable. The values of the `ServiceComponentAction` table are derived from the sample instrumentation events shown before. Some of the values in table `ServiceComponentAction` are not included in the events provided by the Instrumentation (e.g. mutualContextId or elapsedTime), but have to be created by the Management Agent responsible for the corresponding Instrumentation.

### Management Agent

The Management Agents are the concrete implementation of the Manageability Interface described above. They represent the core of the Manageability
Infrastructure (see section 5.1.4).

![Diagram of Manageability Infrastructure](image)

**Figure 57: Role of Management Agents**

The Management Agents, as the main part of the Manageability Infrastructure Design, have to gather and process the information they are responsible for. This information is provided by the Manageability Infrastructure Administrator and the Running Instrumented Service Components. After processing this information, the Management Agents store the information in the local data repository.

**Design Decisions**

The separation of the Manageability Data Model in different aspects and thus, the distribution to different Management Agents is one decision that has to be made during design time. This determines the number of agents in the Manageability Infrastructure, the scope each agent is responsible for, and the Instrumentations or other agents a Management Agent has to communicate with. The number of Management Agents can also be influenced by the expected workload. If the components are intensively used, one Management Agent could not be sufficient to process the arising amount of data in an appropriate time.

Another important design decision is the physical location of the Management Agents. This decision impacts the communication between the Instrumentation and the Management Agent. On the one hand, the communication is more efficient if Instrumentation and Management Agent are located on the same system. On the other hand, in case of a system crash, the Management Agent is also no longer available if Instrumentation and Management Agent are both located on that system. If Instrumentation and Management Agent run on different systems, it is easier to integrate further Instrumentations at a later date.

The design decisions regarding the communication between Management Agent and Instrumentation have already been discussed above. Moreover, an internal structure for the Management Agents has to be developed that supports the aforementioned design decisions.

**Design**

**Separation of responsibility**
The scope of the Manageability Infrastructure is limited to the monitoring capabilities required for monitoring performance related parameters. Furthermore, we focus on providing basic data that do not need to be aggregated. From this it follows that the Manageability Data Model described above is already limited to a certain aspect and does not have to be separated in further aspects. As for the number of Management Agents, it follows that it is sufficient to design one Management Agent that includes the whole Management Data Model.

**Physical location of the Management Agents**
Since the scenario, in which the Management Agent is employed, is characterized by a high grade of distribution, the Management Agent has to run in its own process. It is also recommended to deploy the Management Agent on an extra system, even though this is not essential. By using the ‘Decoupled Agent’ pattern described in [45], it is possible to develop the Management Agent independent of the actual functional design of a service-oriented application. This allows the use of an arbitrary technology for the implementation of the Management Agent. Moreover, the availability of the Management Infrastructure is independent of the availability of the service-oriented application. This is an important feature when it comes to the failure analysis for the application.

**Communication between Management Agent and Instrumentations**
The communication between Management Agent and Instrumentation is handled by a message bus as already described in section 4.2. The Management Agent has a passive role and waits for messages from the Instrumentations. However, the Management Agent needs a component that is able to process the messages from different Instrumentations. Thereby, the agent has to be designed in a way that it is simply extendable for the integration of new Instrumentations. At this point, the ‘Message Dispatcher’ pattern introduced in [46] is employed. In dependency of the message content, the message routing component calls the appropriate implementation. This allows adding new instrumentation data processing components without changing any other component than the message router.

**Internal structure of the Management Agent**
The issues described above lead to the following internal structure of the Management Agent (figure 58).

The InstrumentationSensor component is responsible for the communication between the Management Agent and the Instrumentations. The component waits for incoming messages from the Instrumentations. By the time a message arrives, the InstrumentationMessageHandler (which corresponds to the MessageDispatcher in the ‘Message Dispatcher’ pattern) parses the message content and calls the appropriate operation of the sensor responsible for this type of message. If Instrumentations that do not use the given message format have to be included, the InstrumentationMessageHandler is responsible for the mapping to the required format. In the scope of the first year, there is only the ServiceComponentActionSensor available since the design is limited to the monitoring of service component actions. In order to support additional instrumentation message formats, it is sufficient to extend the InstrumentationMessageHandler. The ServiceComponentActionSensor (which is the counterpart of a Performer in the ‘Message Dispatcher’ pattern) informs the registered management applications about finished transactions at the IndicationPort. Management applications can register for this information at the Manageability Interface. Moreover, the ServiceComponentActionSensor processes the given data and writes the given and calculated information to the according entities of the ManagementRepository. The ManagementRepository includes the entities of the Manageability Data Model as well as an interface that provides access to these entities. The entities in the ManagementRepository are persisted in the ManagementDatabase of the Management Agent. The elements of the ManagementFacade enable the management applications to access the data saved in the ManagementRepository. However, the functionality of the ManagementFacade is already described in section Manageability Interface.

Figure 58: Internal Design of the Management Agent
Example

This section illustrates the behavior of the Management Agent at the arrival of the sample instrumentation events.

![Diagram](image)

**Figure 59: Sample Interaction between Instrumentation and Agent**

When the Instrumentation sends the event that indicates the start of the service operation call of service component `SomeServiceComponent`, the `InstrumentationMessageHandler` calls the `onServiceOperationCallStart()` method of the `ServiceComponentActionSensor`. Since this service operation call is the first in the stack of the managed elements defined in the Manageability Model of the example (see figure 67), the `ServiceComponentActionSensor` creates a new mutual context ID. This mutual context ID will be added to all the following managed element instances on this transaction. Next, the `ServiceComponentActionSensor` creates a new instance of the `ServiceComponentAction` entity to save the received and computed management information. Furthermore, an instance of the `ExecutedServiceComponentActions` association entity is created to link the `ServiceComponentAction` instance with its definition instance.

By the time the Instrumentation sends the event that indicates the end of the service operation call of the service component `SomeServiceComponent`, the `InstrumentationMessageHandler` calls the `onServiceOperationCallEnd()` method of the `ServiceComponentActionSensor`. The `ServiceComponentActionSensor` reads the corresponding instance, created when the operation call was started, of the `ServiceComponentAction` entity from the `ManagementRepository`, calculates the elapsed time, and updates the instances in the `ManagementRepository`. Since the transaction is finished with the end of this service operation call, the `ServiceComponentActionSensor` sends a message to the registered management applications which indicates the end of the transaction. The management...
applications can then use the Manageability Interface to check whether the execution of the service has caused an SLA violation.

**Summary**

This section introduced the design of a Manageability Infrastructure for the monitoring of performance aspects of service-oriented applications. Thereby, the design of the three elements of a Manageability Infrastructure was presented in the form of design decisions, design patterns, the actual design and examples. The models presented in section provide guidance for the actual design of an Instrumentation and therefore meet the requirement R2 (Guidelines for creating the instrumentation).

The Manageability Data Model and the Manageability Interface (which exposes the Manageability Data Model) provide a unified view on the information delivered by the different Instrumentations. Furthermore, they allow the tracking of the executed instances in order to identify the cause of an SLA violation. From this it follows that the two requirements R4 (Unified management view) and R6 (Ability to track executed service transactions throughout component/technology stack).

Finally, the developed Management Agent fulfills the requirement R5 for durable access on the Management Information by processing and storing the information delivered by the Instrumentations.

### 5.1.5 Adding Support for Management Standards

Today, a variety of management solutions exist, that allow conducting the management of applications. Each of these management applications might have its pros and cons in a specific scenario. The proposed solution should be able to be integrated into as many of these applications as possible, so as not to limit the provider to a single vendor or application. Thus, the Manageability Infrastructure has to deal with the requirement of providing a common interface. Consequently, the Manageability Infrastructure should be based on appropriate standards. Hereby, a harmonized management view on the different multi-layer service-oriented applications is created that can be easily integrated into the SLA management of different management applications. In the area of application management multiple standards exist, and new standards are developed over and over. Because of this, the following mapping approach (figure 60) is designed to provide variability concerning the type of standard. It allows using multiple standards, and is simply extendable if new standards should be supported.
The design of the Management Agent Composite including the basic Manageability Interface ($M_{\text{basic}}$) was presented in section 5.1.4. In this chapter it is explained how the Manageability Infrastructure has to be extended by Standard Adapters in order to provide additional standard-based versions of the Manageability Interface ($M_{\text{standard}}$).

This chapter first presents an evaluation of several management standards, which results in a composition of standards we agreed on to use within the SLA@SOI project.

**Evaluation of Management Standards**

In this chapter the management standards presented in section 5.1.3 are evaluated against the general architecture we introduced in section 5.1.4. From this we specify an appropriate combination of standards for the implementation of the manageability interface.

**Management Information Model**

First, we regard the requirements on a management information model, which should cover the considered quality of service aspects. Thus, this management information model should provide a management view on the different data and elements of the service-oriented application that has to be monitored. WBEM provides with CIM a common management information model that meets most of the requirements of SLA@SOI and allows vendor specific extensions. CML will also provide an information model which sounds very promising but the specification is not available yet. NGOSS seems to provide an information model too. However, the model is only available for members of the TMF. The information model provided by the GLUE Specification is specialized to Grid environments and therefore does not cover manageability aspects of service-oriented applications. Hence, we came to the conclusion that CIM would be the best choice for the information model used in SLA@SOI.

**Manageability Interface**

According to the reference architecture presented in section another standard is necessary for implementing the standardized and harmonized manageability interface. Thereto the standard has to integrate the chosen information model and must support the introduced Manager-Agent-Pattern. In this field, WS-Management (from WBEM), WSDM and JMX are possible alternatives. JMX would indeed be suitable for the ORC, but due to its platform-dependency it is not the
perfect choice for the entire SLA@SOI framework. WSDM and WS-Management would be better choices as they rely on interoperable WS standards. Thereby, the two standards are very similar and there are already attempts to conflate the two specifications into one single standard for management of IT resources using web services [IBM07]. Since we decided on CIM as the standard to use for defining the information model, it is advisable to use WS-Management (as another part of the WBEM framework) for implementing the unified manageability interface.

Manageability Infrastructure
Following the reference architecture, the manageability interface is implemented through the manageability infrastructure. This in turn is comprised of management agents that communicate with a specific instrumentation. In case of the instrumentation, there is no common standard available. The Application Response Measurement framework might be used for supporting performance monitoring. The evaluation of this standard however is part of our future work. For implementing the management agents either the WBEM framework (CIM Object Manager + CIM providers) or the JMX framework could be used. As we decided on CIM as in information model, we recommend the employment of the WBEM framework as far as possible. In this way, particularly the discovery of available management information is already supported by the CIMOM. However, in some cases it might be necessary to also use JMX, in particular if providers are required that actively poll information. Nevertheless, we do not expect mixing the two standards to be a problem.

Standard Composition for SLA@SOI
The goal of this section was the identification of an appropriate combination of standards according to the requirements of SLA@SOI. Several existing management standards were introduced and evaluated. Figure 61 shows the affected elements and the determined combination of standards.

![Figure 61: Selected Combination of Standards](image)

In the SLA@SOI project SCA serves as the main modeling approach for SOA software components. Within the ORC these components are implemented using the Java SE Platform, ActiveBPEL and Tomcat Application Server. To monitor these service-oriented applications a manageability infrastructure will be designed which is based on DMTFs WBEM infrastructure. WBEMs CIM describes the management view on the SOA components modelled with SCA. DMTFs WS-Management is the standard specification, which allows the provisioning of a web service based manageability interface and thus provides a harmonized view on the managed SOA.

Available Standard Implementations
There is a set of open source implementations of both integrated and components of the WBEM infrastructure available [50]. Since the ORC is based on Java the chosen implementation should also be based on Java. WBEM Services [51] is an open source Java implementation of WBEM which consists of APIs, server and client applications and tools. An implementation of the WS-Management specification for Java is provided by the Wiseman project [52].

Adding Support for WBEM/CIM

This chapter maps the standard-independent design and models we presented in the previous chapter to the set of standards defined in DMTF’s Web-Based Enterprise Management (WBEM) [53, 54]. After a short motivation for the need of a standard-based interface, the mapping between the Manageability Data Model and WBEM’s Common Information Model (CIM) standard, in particular the CIM core model [55] along with the CIM Metrics Model extension [56], is explained. Afterwards, we present the required extensions to the Manageability Infrastructure Design, in order to support the provisioning of a WBEM-based interface.

Mapping of Manageability Data Model

Since the Manageability Data Model is designed on the basis of CIM, the mapping to a CIM model is straightforward. The classes of the Manageability Data Model are transformed into MOF entities. These MOF entities are then used to extend CIM’s Metrics Model [56] by inheriting from the Metrics Model classes.

![Figure 62: CIM Extension for Manageability Data Model](image)

In the figure above the CIM classes are identified by the prefix `CIM_` and the classes derived from the Manageability Data Model are identified by the prefix `SLASOI`. Within the developed Manageability Infrastructure a unit of work is defined as an action of an SCA service component. Thus, the `SLASOI_ServiceComponentAction` class, which compiles with the `ServiceComponentAction` class of the Manageability Data Model, inherits from
CIM_UnitOfWork. In addition to the attributes that are already defined in CIM_UnitOfWork, SLASOI_ServiceComponentAction specifies the following attributes:

- **IncomingMessage**, and **OutgoingMessage** in order to allow a more detailed analysis of service and reference operation calls.
- **CondStatus** to store the state of a condition of a loop or a conditional flow.
- **LoopCount** and **LastLoopElapsedTime** to provide more information on the execution of a loop within a service implementation.
- **VariableContent** in order to allow the analysis of values that are assigned to certain variables.

The definitional aspects of a SLASOI_ServiceComponentAction are defined in the class SLASOI_ServiceComponentActionDefinition which inherits from CIM_UnitOfWorkDefinition. The SLASOI_ServiceComponentActionDefinition class extends CIM_UnitOfWorkDefinition with attributes for the specification of a BindingType and a more concrete definition of the ActionType.

The actual managed element is represented by the class SLASOI_ServiceComponent. SLASOI_ServiceComponent inherits from CIM_LogicalElement and defines the additional attribute Implementation.

The association classes defined in CIM's Metrics Model directly correspond to the association classes in the Manageability Data Model. Thus, no additional attributes have to be defined in the sub-classes. SLASOI_ExecutedServiceComponentActions inherits from CIM_StartedUoW, SLASOI_SubServiceComponentActions inherits from CIM_SubUoW, and SLASOI_SubServiceComponentActionDefinitions inherits from CIM_SubUoWDef. The meaning of the mentioned associations is explained in chapter XXX. The association between the units of work and the actual managed element they belong to, is realized by the class SLASOI_ServiceComponentServiceComponentActionDefinition which inherits from CIM_LogicalElementUnitOfWorkDef. The class associates an SLASOI_ServiceComponentActionDefinition with an SLASOI_ServiceComponent.

Now that we mapped the Manageability Data Model to CIM, the Manageability Infrastructure Design can be extended in order to provide a WBEM-based Manageability Interface. This process is explained in the next section.

**Extended Manageability Infrastructure Design**

This section aligns the components of the Manageability Infrastructure Design described in chapter 5.1.3 with the components of the WBEM architecture illustrated in this chapter. Within the WBEM architecture, the CIMOM represents the interface to the management clients. Thus, it is the counterpart of the Manageability Interface which is the central access point to the Manageability Infrastructure. The CIMOM acts like a facade that hides the actual information provider from the management client. In WBEM, a provider is defined as an infrastructure that is capable of processing the CIM-formatted requests from the CIMOM interface. The counterpart of such a provider in the Manageability Infrastructure is the Management Agent, since it is responsible for the implementation of the Manageability Interface. The last part that has to be mapped is the resource. In the WBEM architecture a resource is an instrumented managed element. Hence, the counterpart of a resource is a service component together with its corresponding Instrumentations.
Figure 63 gives an overview on the entire extended Manageability Infrastructure.

The Manageability Infrastructure now includes two Manageability Interfaces: The standard-independent Manageability Interface and the CIM-based Manageability Interface. The basic Manageability Interface is directly exposed by the Management Agents. The CIM-based Manageability Interface is exposed by a WBEM Adapter that uses the basic Manageability Interface.

Since the Management Agent is already able to communicate with different resource-specific Instrumentations, the only requirement to the extension of the Manageability Infrastructure Design is that the Management Agent is additionally capable of processing CIM formatted requests and translating responses from Manageability Data Model format into CIM formatted responses. Therefore, a
further component is added to the architecture of the Management Agent.

The **CIM Provider** component implements the interfaces required by the CIMOM. The CIMOM and the **CIM-Provider** component form the standard adapter depicted in figure 64. The CIMOM uses the **CIM-Provider** component of the Management Agent to get instance information on the CIM classes and holds the schema of the extended CIM Metrics Model in its CIM Repository. The three provider classes (**AssociationProvider**, **DefinitionObjectProvider**, and **ExecutionObjectProvider**) perform the communication between the CIMOM and the Management Agent. In case of a request from the CIMOM the appropriate **CIM Provider** class is called. Then, the class translates the request and calls the corresponding operation at the Manageability Interface of the Management Agent. The information responded by the Manageability Interface is then retranslated by the **CIM Provider** component into the format of the extended CIM Metrics Model. The messages at the **Indication Port** are not translated into CIM format and thus cannot be used by the CIMOM. However, since the **Indication Port** only indicates that a service transaction has finished and does not deliver any model specific information the management clients can use the port directly and can then request the needed information at the CIMOM interface. The prototypical realization of this architecture is explained in the following chapter.

**Summary**

In this chapter the basic Manageability Infrastructure has been extended in order to provide a standard-based Manageability Interface (R7). Therefore, the Manageability Data Model was mapped to a CIM-conform data model.
Furthermore, the Management Agent was extended with an additional component that is capable of translating the CIM model into the basic Manageability Data Model and back. Finally, the extended Manageability Infrastructure architecture was introduced.

### 5.1.6 Implementation

This chapter describes the prototypical implementation of the Manageability Infrastructure. The implementation is based on Sun’s Java Standard Edition (SE) Technology. An overview on the implementation of the particular components is followed by a description of the deployment procedure.

#### Solution Overview

**Communication between Management Agent and Instrumentation**

Implementation details on this part are described in section 5.1.3

**Configuration of Static Management Meta-Information**

The communication between Management Agent and Instrumentation described above concerns the dynamic management information that is captured during runtime of the service components. However, the Management Agent must also be provided with the static management meta-information as specified in the Manageability Data Model. The configuration of this definitional data can be accomplished by using the Manageability Interface. However, for the prototypical implementation there is also a tool available that reads the data from XML files (5).

![Figure 65: Definition Loader Implementation](image)

The DefinitionLoader reads the definition instances from one or many XML files into the repository of the Management Agent using the Manageability Interface. The required format of the files is specified in terms of an XML-Schema definition. An advantage of using this tool is that the Manageability Designer can easily create the XML file according to the given schema and need not be aware of the operations at the Manageability Interface. Furthermore, the tool can be used for the automatic configuration of the Manageability Infrastructure by means of transformations.

**Management Agent Repository**

The management repository (or management information base) is the implementation of the Manageability Data Model described in chapter XXX. It is realized using the Java Persistence API (JPA). JPA provides an object/relational mapping facility for managing relational data in Java applications. The objects of the Manageability Data Model are implemented in terms of persistence entities. An entity is an annotated java object that represents a table in a relational database. Thus, each entity instance corresponds to a row in that table. The object/relational mapping annotations map the entities and entity relationships to the relational data in the underlying data store. Apache Open JPA is the open source implementation of the Java Persistence API employed within this prototype. The database used by the implementation is Apache Derby.
Manageability Interface

The operations provided at the Manageability Interface are exposed by the use of Java Remote Method Invocation (RMI) (included with Java SE). RMI enables Java technology-based clients to invoke the methods of the Manageability Interface from other Java virtual machines, possibly on different hosts. The WBEM version of the Manageability Interface is realized by means of WBEM Services. WBEM Services is an open-source Java implementation of WBEM. It consists of APIs, server and client applications, and tools. The provider classes used by the CIMOM are implemented in terms of adapters that call the RMI methods provided at the basic Manageability Interface.

For more detailed information on the implementation, see the Javadoc and the source files located in the appendix.

Deployment

This section describes how the Manageability Infrastructure has to be deployed.

Prerequisites

- Java Standard Edition 5 or higher
- Apache Ant 1.6 or higher

Deployment Procedure

(1) Install Manageability Infrastructure

- Unzip the installation package `ManageabilityInfrastructure.zip` in your installation folder (any folder without blanks in the path) or get latest version from SLA@SOI’s SVN repository.

(2) Open the properties file `...\manageabilityinfrastructure\manageability.properties` and change properties such as hostnames and ports according to your needs. However, you can also run the Manageability Infrastructure using the basic settings in the properties file.

(2) Start Management Agent

- Open the folder `manageabilityinfrastructure` in the command line.
- Run command `ant runBroker` to start the ActiveMQ message broker.
- Open another command line and go to the `manageabilityinfrastructure\folder`.
- Run command `ant runInstrumentationSensor` to start the sensor that receives the messages from the Instrumentation of the monitored application system.

(3) Start RMI-based Manageability Interface

- Open a further command line and go to the `manageabilityinfrastructure folder`
- Run command `ant runFacade` to start the RMI-based Manageability Interface

(4) Start WBEM-based Manageability Interface

- Open folder `...\manageabilityinfrastructure\wbemservices\cimom\bin` and run `start_cimom.bat` on Windows system or `start_cimom.sh` on Unix system to start the CIMOM.
(5) Load definition instances

- In order to be able to store the runtime management information, the corresponding definition instances must be loaded into the Manageability Infrastructure repository. This can be accomplished using the DefinitionLoader tool or the Manageability Interface.
- To run the definition loading tool, open the properties file (see step 2) and set the path to the definition file(s). This can either be a folder or a concrete filename.
- The XML schema file is located in the following directory:
  \manageabilityinfrastructure\src\main\resources\manageability\definitionProvisioning
- Open directory \manageabilityinfrastructure in command line and run command ant runDefinitionLoader.

5.1.7 Using the Manageability Interface within ORC / Ad Hoc Demonstrator

This chapter shows the application of the manageability interface within the ad hoc demonstrator. First, we introduce a sample scenario based on the ORC and afterwards we demonstrate how the solution can be used to detect SLA violations as part of the service operation phase and the SLA adaptation.

Sample ORC Scenario

The SCA model in figure 66 shows the functional design of the payment service.

![Figure 66: Payment Service](image)

The Payment Service is implemented in BPEL by the software provider. Thereby, the composition depends on the atomic services Card Validation Service and Payment Debit Service also provided by the software provider. The SLAs between the service provider and the software provider have to be derived from the SLA the service provider negotiated with the customer. Further SLAs have to be negotiated between the service provider and the infrastructure provider since all service run on top of the infrastructure offered by the infrastructure provider. However, we focus on the management of service compositions and atomic services, the infrastructure is not considered at this point. Figure 67 shows
sample SLAs as they could be negotiated in the scenario explained above.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Usage Profile [requests/second]</th>
<th>Percentage (Completion Time)</th>
<th>Completion Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle Payment()</td>
<td>0.69</td>
<td>90</td>
<td>200</td>
</tr>
</tbody>
</table>

**SLA Payment Service
*********************/

**Conditions**

**SLA Card Validation Service
***************************/

**Conditions**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Usage Profile [requests/second]</th>
<th>Percentage (Completion Time)</th>
<th>Completion Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate Card()</td>
<td>0.69</td>
<td>90</td>
<td>70</td>
</tr>
</tbody>
</table>

**SLA Payment Debit Service
***************************/

**Conditions**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Usage Profile [requests/second]</th>
<th>Percentage (Completion Time)</th>
<th>Completion Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>debit Card()</td>
<td>0.65</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

**Figure 67: Sample SLA for PaymentService**

The customer declares that it requires the *Payment Service* in a maximum frequency of 0.69 requests per second. The service provider guarantees a completion time of less than 200ms in 90 percent of the service calls if the customer adheres to the obligated usage profile. In order to offer this service level objective (SLO), the service provider translates the objective into sub-objectives for the atomic services the *Payment Service* depends on. Thus, it negotiates appropriate SLAs with the software provider (and the infrastructure provider).

Once the SLAs are negotiated, and the services are running and in use, the service provider has to take care that it achieves the SLO offered to the customer during service operation phase. Therefore, it requires a comprehensive SLA management that monitors the metrics defined in the SLA. In order to calculate these aggregated metrics of an SLO the SLA management needs structured basic management information, which is provided by the Manageability Infrastructure.

**Using the Manageability Interface within Service Operations Phase**

Within the service operations phase, the Manageability Infrastructure monitors the service-oriented application. Using the definitional meta-information (A6 Deliverable) and the dynamic management information regarding the executed instances of these definitional elements, the Management Agent offers a unified management view on the heterogeneous service-oriented retail solution. Thus, the SLA management application is capable of calculating the aggregated metrics and statistical data specified in the SLOs. However, detecting an SLA violation is only one issue the SLA management has to deal with. In fact, the service provider must also be able to identify the cause of an SLA violation in order to start corresponding countermeasures. Since the Manageability Interface also allows the specification of structural information, and information concerning the status of critical internal actions of an operation (e.g. the state of a condition or the number of loop iterations), the Manageability Interface eases the analysis of those SLA violations.
An SLA violation can be caused by any component within the SOA stack. Some examples are listed beneath:

- The customer does not meet his obligations since the actual request arrival rate exceeds contracted average.
- The user made a mistaken input that caused a component to run extraordinarily long.
- The implementation of a service component (composition or atomic) contains an error that was not found in the test phase of the component.
- Middleware, Network or Hardware is overloaded or no longer available.
- There are too many database entries so that a request takes more and more time.

In the following some possible indications are outlined that allow identifying the cause of a violation subject to the component of the stack.

- **Customer Fault:**
  - # service operation calls in timeframe [min] / timeframe [min]
  - individual analysis of the incoming messages of a service operation call

- **Composite or Engine:**
  - response time of provided service operation call in relation to the response time of the integrated services

- **Database:**
  - # database transactions
  - # database entries
  - response time of database request in relation to the response time of the calling service component

- **Network:**
  - delay between start time of reference operation call in the calling service component and start time of service operation call in the called service component

In the following we describe how to use of the Manageability Interface for detecting the cause of SLA violations, which is crucial to support for the support of SLA adaptation (see WP A5 deliverable for more information). With the Manageability Interface at hand the SLA Management is able to request the data required to calculate the metrics outlined above and identify the cause of an SLA violation and start appropriate countermeasures. Examples of such countermeasures are listed below:

- changing the configuration of the middleware (e.g. BPEL engine)
- scaling the hardware resources like network bandwidth, number of processors, main memory
- improving the quality of code within the service component implementation
- switching the binding between service components
- temporary forwarding of service operation calls to a service implementation with a higher service level
- redeployment of the service instance on another machine
- notifying the customer about usage profile violation

In the following three usage scenarios are illustrated to demonstrate the capability of the Manageability Interface.

**Scenario 1**
Cause of Failure: The BPEL engine that runs the PaymentService composition is overloaded.

1. The SLA Management calls the `getExecSCActions()` operation with the parameter:
   a. definition ID of the service operation call of operation `handlePayment()`
   The Manageability Infrastructure returns all executed service operation call instances of the `handlePayment()` operation.
2. The SLA Management requests for each of these instances the corresponding child instances. In case of the PaymentService the Manageability Interface returns the reference operation call instances of the operations `validateCard()`, and `debitCard()`.

Now, the SLA Management can compare the elapsed time of the service operation call instances of operation `handlePayment()` with the elapsed time of the corresponding reference operation call instances of the operations `validateCard()`, `debitCard()`, and `createReceipt()`. This results in the following table:

<table>
<thead>
<tr>
<th>Instance</th>
<th>SOC: handlePayment()</th>
<th>ROC: validateCard</th>
<th>ROC: debitCard()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>280 ms 😐</td>
<td>65 ms 😐</td>
<td>88 ms 😐</td>
</tr>
<tr>
<td>2</td>
<td>290 ms 😐</td>
<td>63 ms 😐</td>
<td>81 ms 😐</td>
</tr>
<tr>
<td>3</td>
<td>286 ms 😐</td>
<td>66 ms 😐</td>
<td>83 ms 😐</td>
</tr>
<tr>
<td>4</td>
<td>305 ms 😐</td>
<td>66 ms 😐</td>
<td>88 ms 😐</td>
</tr>
</tbody>
</table>

**Figure 69: Results**

The table above shows that the response times of the integrated basic services are all beyond the values defined in the corresponding SLAs. Thus, one can easily conclude that the failure must either be caused by the engine that runs the service composition or by the underlying hardware.

**Scenario 2**
Cause of Failure: There is a higher arrival rate than contracted in the SLA of the PaymentService => Customer Fault.

Figure 70: Scenario 2

1. The SLA Management calls the `getExecSCActions()` operation with the following parameters:
   a. definition ID of the service operation call of operation `handlePayment()`
   b. the timeframe in which the SLA violation occured

The Manageability Interface returns all executed service operation call instances of the `handlePayment()` operation within the given timeframe.

With the number of returned service operation call instances, the SLA Management can easily calculate the arrival rate and detects that the violation was caused by the customer.

Scenario 3
Cause of Failure: Bad network performance between the service components PaymentService and PaymentDebitService.

Figure 71: Scenario 3

1. The SLA Management calls the `getExecSCActions()` operation with the parameter:
   a. definition ID of the service operation call of operation `handlePayment()`

The Manageability Infrastructure returns all executed service operation call instances of the `handlePayment()` operation.

2. The SLA Management requests for each of these instances the corresponding child instances. In case of the PaymentService the Manageability Interface returns the reference operation call instances of the operations `validateCard()` and `debitCard()`.
3. Additionally, the SLA Management requests the child instances of the `debitCard()` reference operation call instances. This request entails that the Manageability Infrastructure returns the service operation call instances of the operation `debitCard()` which belongs to the service component `PaymentDebitService`.

When executing the requests described above, the SLA Management gets the information depicted within the table below.

<table>
<thead>
<tr>
<th>Instance</th>
<th>SOC: handlePayment()</th>
<th>ROC: ValidateCard</th>
<th>ROC: debitCard()</th>
<th>SOC: debitCard()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>270 ms</td>
<td>65 ms</td>
<td>98 ms</td>
<td>75 ms</td>
</tr>
<tr>
<td>2</td>
<td>281 ms</td>
<td>63 ms</td>
<td>101 ms</td>
<td>78 ms</td>
</tr>
<tr>
<td>3</td>
<td>276 ms</td>
<td>66 ms</td>
<td>103 ms</td>
<td>77 ms</td>
</tr>
<tr>
<td>4</td>
<td>295 ms</td>
<td>66 ms</td>
<td>108 ms</td>
<td>81 ms</td>
</tr>
</tbody>
</table>

*Figure 72: Results*

When analyzing this information it becomes clear that the cause of failure must be located in the communication between the service component `PaymentService`, which conducts the reference operation call `debitCard()`, and the service component `PaymentDebitService` which implements the service operation call `debitCard()`.

5.2 Autonomic Service Management

In previous sections, the monitoring solution in SLA@SOI at the software level has been described. This monitoring module can detect SLA violations but does not always provide sufficient information for deciding the most appropriate response to a violation. 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 Module described in this section, thus helping the service providers to guarantee their SLA commitments not only by detecting SLA violations once occurred but further by anticipating these violations to trigger the appropriate preventing actions.

5.2.1 Principles of Autonomic Systems

The increasing complexity of the system and communication infrastructure of any company makes the IT systems management very difficult and resource-consuming, just threatening to undermine the income of the company and therefore their ability to develop their business. This was first pointed out as the real challenge facing the IT industry by Paul Horn, Vice President of the IBM Corporation and Director of Innovation, in a speech at the Academy of Engineering at Harvard University [2].

P. Horn suggested as a solution 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 [3] 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 [4] 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 defend 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 [5] (Monitor-Analyze-Plan-Execute-Knowledge). This architecture, shown in figure 73, is based on an Autonomic Element (AE) implementing an intelligent control loop.

![Figure 73: MAPE-K Architecture](image)

The Autonomic Element is composed of four parts that share a knowledge layer:

- The monitor function provides the mechanisms to collect, aggregate, filter and report details (such as metrics and topologies) collected from a managed resource.
- The analyze function provides the mechanisms to 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.

The MAPE-K architecture describes an approach for creating self-managing resources and composing them to form self-managing systems, with the ultimate goal of dramatically reducing the cost and perceived complexity of the IT systems. A self-managing resource governs its behavior according to established policies and exposes manageability interfaces so that it can participate in system-wide autonomic control.

### 5.2.2 Autonomic-based Adjustment

The Adjustment Module 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.

SLA Enforcement ensures continual identification, monitoring and reviewing of the optimally agreed levels of services as required by the business. Most targets set in a Service Level Agreement are subject to direct financial penalties or indirect financial repercussions if not met. It is therefore critical for this management process to flag when service levels are projected to be violated in order for the service provider to take actions to address the issue. To this extent, the adjustment module must detect any deviation from the expected standard operation of a service that causes, or may cause an interruption to the normal operation, or a reduction in the quality of the service. The objective is to provide continuity by restoring the service in the quickest way possible.

The main goals of the Adjustment module described above can be summarized as follows:

1. The inference of the appropriate corrective action for a given SLA violation.
2. The order of the execution of the corrective action.
3. The monitoring of the performance of the corrective action.
4. The determination, based on post-corrective action monitoring results, whether corrective action goals have been attained, or are likely to be attained.
5. To propose additional or alternative corrective action measures in case the executed corrective action is unable to attain its goals.

Corrective actions allow the specification of automated responses to SLA violations. The Adjustment module will ensure that routine responses to alerts are automatically executed, therefore saving time and costs, and even ensuring problems are dealt with before they noticeably impact users.
In order to achieve that, the adjustment module must form a closed loop with its environment, in which the system is able to monitor itself and its context, analyses these data to ensure that goals and objectives are being met, alters its behaviour in case these goals and objectives be threatened, and observes the result to ensure that closed-loop operation is maintained.

The change in conditions that appear at runtime cannot be planned and requires a real-time approach. Classical control theory, represented in figure 74, has been proven to work in achieving this purpose.

![Figure 74: Block diagram for a classical closed control loop](image)

The reference input is the desired value of the system’s measured output. The sensor monitors the output and feeds the data to a computer which continuously adjusts the control input as necessary to keep the control error to a minimum. Feedback on how the system is actually performing allows the controller to dynamically compensate for disturbances to the system. An ideal feedback control system cancels out all errors, effectively mitigating the effects of any forces that may or may not arise during operation and producing a response in the system that perfectly matches the user’s wishes.

In the particular case of the SLA enforcement, the Reference is specified by the parameters or Service Level Objectives signed in the SLA, which, when compared with the actual value of the parameters (measured output), could produce a SLAViolation. This event is the input (measured error) to the Adjustment Module, playing the role of the Controller block. The Adjustment can potentially order some corrective action (system input) to the SLA@SOI system. After the configuration, the system continues running and emitting raw data as output, that can be measured by the monitoring module (Sensor).

Although a design based on the classical control theory describes well the adjustment functionality, as shown in the paragraph above, the adjustment architecture can be further evolved implementing an intelligent control loop. That means that 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 all these functions can be automated, an intelligent control loop is formed, and the system has self-managing capabilities.
Figure 75 shows a schematic representation of the adjustment control loop.

![Adjustment Control Loop Diagram](image)

**Figure 75: Adjustment Control Loop**

Inputs to the control loop consist of messages sent by the monitoring module. These messages are, either signals indicating that a software-level SLA has been violated, or warnings indicating that a problem is about to occur. These are then analyzed to ensure that the system is providing the appropriate services and resources at this moment in time. If it is not, then the system is reconfigured and re-analyzed. Outputs of the control loop are commands to the affected component to adjust its operation, or messages to the business level. The manageability interface described in section 5.1.4 is the layer that provides access to the managed resources at the software level. Therefore, it can be seen as the effector allowing the reconfiguration of the software components.

The similarities between the adjustment control loop and the MAPE-K architecture in 74 and 75 are rather evident. This leads to the conclusion that autonomic techniques can be used in SLA@SOI to implement an advanced adjustment of dynamic behaviours of services and processes. In order to do that, a knowledge layer must be added to the basic adjustment control loop. This knowledge component will be based on a domain semantic description, and will enriched the system with properties as robustness, self-healing, self-managing, self-optimization and self-improvement.

This method, based on the real-time system reconfiguration, exploits the system monitoring information, along with quality parameters from the SLAs and experience from previous executions (e.g. previous SLA violations). The proposed set of actions is based on the application of specific policies, which are closely coupled with particular SLAs in order to provide, in real time, the required QoS level for the application.
5.2.3 Initial Architecture of Adjustment module

As described in the previous section, SLA@SOI will use autonomic techniques to implement the adjustment module that ensures the correct dynamic behaviours of services and processes. This process must support the robustness, self-healing, self-managing, self-optimization and self-improvement of the system components.

The initial architecture for the adjustment module will not provide all the characteristics defining an autonomic system. As shown in figure 76, the Autonomic Element accepts data from the software context in form of SLAViolations through the Event Bus, and, using various analysis techniques, determines the impact, or potential impact, of this information (e.g., a software failure means that a group of customers is not being delivered the QoS level indicated in their SLA for the contracted service). Then, a set of reasoning processes are used to determine whether the actual state of the system corresponds to the desired state. If it does not, the Autonomic Element determines corrective action (or set of actions) to take, and orders the execution of the reconfiguration operations.

Of central importance to the adjustment system behavior is the ability for high-level, broadly-scoped directives to be translated into specific actions to be taken by resources. This is achieved through the use of policies to determine the types of decisions and actions that autonomic capabilities perform.

![Figure 76: Simplified Adjustment Architecture](image)
The Execution Manager component is responsible for ordering the execution of the planned action on the system. It can trigger the reconfiguration of the software component that presents a wrong behaviour, using the manageability interface described in section 5.1.4. When the SLA violation involves a business action (e.g. application of penalties or rewards), the adjustment module generates a notification to the business layer.

This architecture will be refined in following versions, closing the control loop and providing an autonomic-based adjustment module.

**Conclusions**

**6.1 Summary**

This deliverable document provided insights into the activities of WP A3 under the umbrella topic of SLA-aware Service Management. The main objective of the document was to describe the achievement made in the first year with a special focus towards the adhoc demonstrator. SLA@SOI aims at delivering an SLA management framework capable of holistic SLA management for multi-layer SOA stacks. WP A3 specifically addresses the software services layer which sits between the business service layer and the infrastructure services layer. Within WP A3, work was carried out in a number of different yet connected and complementary themes, all of which work in a coordinated fashion to deliver SLA-aware service management capabilities to the SLA management framework.

Modelling activities delivered basis for work both within A3 as well as activities in other work packages in SLA@SOI. In this regard, focus was on two main aspects: SOA modelling and landscape modelling. SOA modelling attempted to model the service components of the SOA based applications. Special focus was on modelling service components behaviour and non-functional properties which are used for design time prediction activities. Service Component Architecture (SCA) was extended to provide for these modelling aspects. Landscape modelling was the other aspect of modelling covered in A3. In this regard, we designed a meta-model which can be used to capture software service landscape artefacts like services, software elements (application software and execution software) as well as the packaging and deployment information.

Dynamic service composition and binding was also an important part of the A3. The activity focused on providing capabilities to dynamically compose services and bind the services during run time based on the quality of service properties of the composition. This capability is a vital requirement in dynamic environments which is the main premise of SLA@SOI objectives.

Monitoring & Management activities focused on the runtime aspects of service management. A3 adopted and event based non-intrusive monitoring approach. Service instrumentation investigated the instrumentation of services and composition engines to capture and disseminate events to be analyzed and correlated for SLA violation detection. The landscape elements which were monitored in this context were atomic services, composite processes as well as the process execution engines. In terms of manageability infrastructure, work was conducted on designing a unified manageability infrastructure for various elements of the software service landscape. An important element of this manageability infrastructure is Unified Manageability Interface which is enables
the management applications to get monitoring information as well as controlling the landscape elements. Additional emphasis was on equipping the system with autonomic management capabilities.

### 6.2 Outlook on Future Work

There was a substantial amount of work done in WP A3 in Y1; however, the current focus was on the SLA@SOI adhoc demonstrator. In the future, focus will be shifted to take into account the requirements from the various industrial use cases SLA@SOI will be targeting. Therefore, in the following years, we will build on our work done so far and extend the artefacts to make sure that these are ready to be used in the industrial used cases which represent real world scenarios.

In the context of modelling activities, SOA modelling will build on the existing work and identify the refinements required to model the complex service components behaviour from the industrial use cases. Whereas landscape modelling will focus on how the landscape models can be linked to the infrastructure models so that the administrators can be get a holistic view of the complete landscape. Additionally, landscape meta-model will be revised to captures the information from running software landscapes.

For dynamic composition and binding activities, we are going to realize an implementation of the conceptual architecture sketched in Figure 15 based on an extension of the open source product activeBPEL.

For monitoring, there are several ways in which the work presented within SLA@SOI in the first year could be extended. A first extension of our approach concerns the dynamic assignment of monitoring tasks. Given a provisioned SLA, the service monitoring framework should be able to assign its monitoring to the best available monitoring engine. In this case, the suitability of a monitoring engine for the monitoring of a given property can be assessed in terms of both theoretical and performance-related aspects. A monitoring engine, for example, may not be able to monitor a property because (i) it has not be designed to monitor certain kinds of service properties or (ii) it is currently overloaded and may not be able to receive further service properties to be monitored.

At the level of the service monitoring framework, we plan to develop a more tight integration between service monitoring and SLA negotiation within the SLA management framework. First, during SLA negotiation, service monitoring should ensure that the service properties under negotiation may actually be monitored at runtime during service execution. Service monitorability could be evaluated in terms of the intersection of (i) the events produced by the service execution environment and (ii) the events required by a monitoring engine to actually check the satisfaction of a given monitoring property. Second, data on SLA violations and satisfactions over time could be used to fine tune the negotiation strategies of service providers and customers in future negotiations.

The Adjustment Module, as it has been implemented in this initial phase of the project, does not present all the characteristics that define an autonomic system. In particular, the closure of the loop, i.e. the supervision of the results of the control actions that have been raised, must be designed and will be included in following versions. Additionally, the activity on multi-domain / multi-provider service management will commence in Y2.
Overall, considering our progress and achievement in Y1, we are confident that we will be able to build on our existing work and deliver service management capabilities which will be ready to be used for the real world industrial use cases in the following years.

7 References


Appendix A: Glossary

The glossary from Deliverable D.A1a was used throughout this document.

Appendix B: Abbreviations

AOP Aspect Oriented Programming
ARM Application Response Measurement
BPEL Business Process Execution Language
CIM Common Information Model
CML Common Model Library
CQL CIM Query Language
DMTF Distributed Management Task Force
DOW Description of Work
EC Event Calculus
GLUE Grid Laboratory Unified Environment
HTTP Hypertext Transfer Protocol
ISO International Organization for Standardization
JMX Java Management Extensions
MIB Management Information Base
MOF Managed Object Format
MOWS Management of Web Services
MUWS Management using Web Services
NFP Non-functional Property
OASIS Organization for the Advancement of Structured Information Standards
ORC Open Reference Case
RTML Runtime Monitor Specification Language
SID Shared Information / Data Model
SDM System Definition Model
SLA Service Level Agreement
SLAT SLA Template
SLO Service Level Objective
SML Service Modelling Language
SCA Service Component Architecture
SOA Service Oriented Architecture
UML Unified Modelling Language
WBEM Web Based Enterprise Management
WSDL Web Service Description Language
WSDM Web Services based Distributed Management
XML Extensible Markup Language

Appendix C: Event Schema
Figure 80 reports the XML Schema adopted for expressing events within the Service and SLA monitoring. The schema can be used for expressing both Interaction Events and Monitoring Result Events.

The schema can be used to represent two different types of events, i.e. Interaction Event and Monitoring Result Event types.

- **InteractionEventType**: events of this type represent service operation calls and responses captured by the instrumentation of the Open Reference Case services. More in general, they refer to the basic information required to feed a generic monitoring engine to check the satisfaction of a monitored property derived from an SLA;

- **MonitoringResultEventType**: events of this type represent violations of GuaranteeTerm. An event of this type is therefore defined by the IDs of the GuaranteeTerm that has been violated and of the SLA to which the Guarantee Term belongs.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<x:schema
xmlns="http://slasoi.org/monitoring/xml/eventformat"
xmlns:x="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="EventInstance">
    <xs:sequence>
      <xs:element name="EventID" type="EventIdType" />
      <xs:element name="EventContext" type="EventContextType" />
      <xs:element name="EventPayload" type="EventPayloadType" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="ipAddress">
    <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="opStatus">
    <xs:restriction base="xs:string">
      <xs:pattern value="REQ-B|REQ-A|RES-B|RES-A" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="EventIdType">
    <xs:sequence>
      <xs:element name="ID" type="xs:long" />
      <xs:element name="EventTypeID" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="StructArgument">
    <xs:sequence>
      <xs:element name="Argument" type="SimpleArgument" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="SimpleArgument">
    <xs:sequence>
      <xs:element name="ArgName" type="xs:string" />
      <xs:element name="ArgType" type="xs:string" />
      <xs:element name="Direction" type="xs:string" />
      <xs:element name="Value" type="xs:anySimpleType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ArgumentType">
    <xs:choice>
      <xs:element name="Struct" type="StructArgument" />
      <xs:element name="Simple" type="SimpleArgument" />
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="ArgumentList">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Argument" type="ArgumentType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="InteractionEventType">
    <xs:sequence>
      <xs:element name="Struct" type="StructArgument" />
      <xs:element name="Simple" type="SimpleArgument" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```
In the following, we present the list of concrete events required for monitoring the SLAs defined for the ORC services within the SLA@SOI ad-hoc demonstrator.

List of Available Services in the ORC (from Scenario Document)

- Composed Service: PaymentService
- Atomic Services
  - PaymentDebitService (partner of the PaymentService Business Process)
  - CardValidationService (partner of the PaymentService Business Process)
  - InventoryService

List of Interaction Events

Note: these events are generated by the instrumentation of the ORC and consumed by the Monitoring module to detect SLA violations at the Software Service Layer

From the BPEL Engine instrumentation:

Start and termination of a business process instance:
- PaymentService: started(processInstance_id)
- PaymentService: terminated(processInstance_id)

Invocation of operations exposed by a partner service (BPEL engine calling its partner services) – from to
- PaymentService \( \rightarrow \) PaymentDebitService: debitCard()
- PaymentService \( \rightarrow \) CardValidationService: validateCard()

Note: partner services of the BPEL Process PaymentService expose only one operation.

Service responses (BPEL engine receiving responses from its partner services) – to left
- PaymentService \( \leftarrow \) PaymentDebitService: debitCard()
- PaymentService \( \leftarrow \) CardValidationService: validateCard()

From the AXIS container instrumentation:

- The AdHoc Demonstrator is the client application of the atomic services

Invocation of operations exposed by a partner service (AdHoc calling an atomic service) – from to
- AdHoc \( \rightarrow \) PaymentService: request()
- AdHoc \( \rightarrow \) InventoryService: operationX()
Note: the PaymentService exposes only one operation to its client, whereas the InventoryService exposes several operations.

Responses received from invoked operations (AdHoc demonstrator receiving a response from an atomic service) – to ↔ from
• AdHoc ↔ PaymentService: request()
• AdHoc ↔ InventoryService: operationX()

Example: Invocation by the AdHoc Demonstrator of InventoryService:getProductDetails()
Response received from invoked operation InventoryService:getProductDetails()

Monitoring Result Events
Note: these events are generated by the Monitoring module and consumed by the AdHoc demonstrator and Adjustment modules

The generic Monitoring Result is identified by the id of the SLA and of the violated Guarantee Term
Example
Violated SLA: “SLA_AdhocDemonstrator_InventoryService$Revision:XXXX”
Guarantee Term: “CompletionTime_BookSale_GTerm”