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

Deliverable D.A5.a

Keywords:
Service Level Agreement, Foundations, SLA Management

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: UDO
Revision: V.0.1 (26th April 2009)

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

| Dissemination level | PU | Public | Yes |
### Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>Constantinos Kotsokalis (UDO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Christof Momm (FZI)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Juan Lambea Rueda (TID)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Joe Butler (INTEL)</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td></td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td></td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td></td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td></td>
</tr>
<tr>
<td>PMT Approval</td>
<td></td>
</tr>
</tbody>
</table>

### Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDO</td>
<td>Constantinos Kotsokalis (editor), Miguel Angel Rojas Gonzalez</td>
</tr>
<tr>
<td>ENG</td>
<td>Keven Kearney, Francesco Torelli</td>
</tr>
<tr>
<td>CITY</td>
<td>Marco Comuzzi</td>
</tr>
<tr>
<td>SAP</td>
<td>Ulrich Winkler, Tariq Ellahi</td>
</tr>
<tr>
<td>FBK</td>
<td>Annapaola Marconi</td>
</tr>
<tr>
<td>TID</td>
<td>Beatriz Fuentes, Alfonso Castro Escudero</td>
</tr>
<tr>
<td>PMI</td>
<td>Liliana Pasquale</td>
</tr>
<tr>
<td>eTel</td>
<td>Mark Evenson</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.

### Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>26/04/2009</td>
<td>WP A5</td>
<td>First draft for internal review, including all submissions from A5 partners and edited.</td>
</tr>
<tr>
<td>0.2</td>
<td>15/05/2009</td>
<td>WP A5</td>
<td>Second draft, after 1st round of internal review</td>
</tr>
<tr>
<td>1.0</td>
<td>29/05/2009</td>
<td>WP A5</td>
<td>Final version</td>
</tr>
</tbody>
</table>
Executive Summary

This document, SLA@SOI deliverable D.A5a, is the first in a series of three annual deliverables which document the progress in the context of Work Package A5 ("SLA Foundations and Management"). We herein focus on the core results and major achievements.

The document is organised in the following structure:

• First, an introduction is discussing briefly the topic in its most abstract form: What is a Service Level Agreement, what are the major conceptual gaps and the major challenges.

• Then, a chapter follows on "essential concepts". We discuss the most important parts of a complete, detailed SLA model developed within the first year; we formally define the technical terms that will be included in our electronic contracts (i.e. the SLAs); we detail the lifecycle of a Service Level Agreement; and we elaborate on the very concept of SLA hierarchies, the various views on them, their meaning and candidate approaches.

• The next chapter discusses the architectural details that pertain to WP A5 – a focused subset of the overall architecture elaborated in deliverable D.A1a. We start the analysis from template management, and the main use cases related to this topic. Then, we provide insight on negotiation, i.e. the way to establish SLAs. It should be noted, that for Year 1 we only dealt with customer-initiated, 1-1, single-round negotiations – the simplest form there is. In upcoming work, we will extend negotiation to include different strategies and cardinality of parties, multiple rounds with counter-offers, etc. The presentation of template management is followed by a discussion on provisioning architecture – more specifically, scheduling the SLAs and starting the services when in due time, also taking into account configuration and software-inherent start-up delays. After that, the document introduces the concepts used for Monitoring the established SLAs. This particularly includes information on the event-based monitoring architecture. The chapter closes with insight on the architectural foundations for reacting to SLA violations, i.e. when the SLA terms are not honoured anymore.

• The fourth chapter is concerned with the implementation of these concepts. We briefly describe the main parts of our implementation work, towards implementing the architectural design from chapter 3. This provides a chance to discuss inherent difficulties in generic and complete SLA template discovery; the adaptation of an existing WS-Agreement framework to match our current architecture; a simple mechanism for storing SLAs and retrieving their hierarchy recursively, starting from any point in this graph; the underpinnings for implementing the monitoring facilities; and finally the footing for the necessary reactive mechanisms when monitoring detects a violation.

• The closing chapter is about conclusions, but also about future outlook. This important discussion is related to the industrial use case analysis, completed at the end of this first year. This analysis has already provided invaluable input, and a multitude of requirements, some of them completely new in relation to the current design and implementation choices. It is therefore anticipated that an architectural update is needed to address these new requirements; this is the starting point for Year 2 and the rest of the project.
# Table of Contents

1 Introduction .................................................................................................................. 8  
  1.1 High-level design choices ..................................................................................... 8  
    1.1.1 SLA hierarchy establishment and activation process .................................. 9  
2 Essential Concepts ..................................................................................................... 11  
  2.1 Conceptual SLA Model ....................................................................................... 11  
    2.1.1 Service & Agreement Concepts .................................................................. 11  
    2.1.2 Service Level Agreements (SLA) ............................................................... 12  
    2.1.3 SLA Templates ......................................................................................... 17  
    2.1.4 Support for Annotations .......................................................................... 18  
  2.2 Standard QoS Terms ......................................................................................... 18  
  2.3 On The Use Of Semantics ................................................................................... 24  
    2.3.1 Uses of Semantics in SLA@SOI ............................................................... 24  
    2.3.2 Guarantee Terms ...................................................................................... 25  
  2.4 SLA Lifecycle ....................................................................................................... 26  
    2.4.1 Description of SLA States ......................................................................... 27  
    2.4.2 Guarantee Term States ............................................................................. 29  
  2.5 SLA Hierarchies ................................................................................................... 30  
    2.5.1 The Association Relationship .................................................................. 31  
    2.5.2 The SLA Hierarchy Graph ....................................................................... 31  
3 Architectural Elements ............................................................................................... 35  
  3.1 Management of Templates .................................................................................. 35  
    3.1.1 Use Case Actors ....................................................................................... 36  
    3.1.2 Use Case : SLA Template Design & Management .................................. 36  
    3.1.3 Use Case : Template-based Service Discovery ....................................... 38  
  3.2 Establishing SLAs ............................................................................................... 39  
    3.2.1 Negotiation strategies ............................................................................. 39  
    3.2.2 Architectural details For Year 1 ............................................................... 41  
3.3 Provisioning ........................................................................................................... 46  
    3.3.1 Provisioning Module Architecture ............................................................. 47  
    Service Activation Engine ...................................................................................... 47  
    3.3.2 Public Interfaces ....................................................................................... 50  
  3.4 Monitoring .............................................................................................................. 53  
    The Overall Architecture of the Monitoring module ........................................... 54  
  3.5 Reacting ................................................................................................................. 59  
    3.5.1 Adjustment in the Overall Architecture ................................................... 60  
    3.5.2 Component and Interface specifications ................................................. 61  
    3.5.3 Behavioural view ..................................................................................... 63  
    3.5.4 Internal Module Architecture ................................................................. 66  
4 Implementation ............................................................................................................. 70  
  4.1 Template Registry ............................................................................................... 70  
    4.1.1 Approach .................................................................................................. 70  
    4.1.2 Query Resolution ..................................................................................... 72  
    4.1.3 Data-Model ............................................................................................. 73  
    4.1.4 SLA Template Registry Interface ............................................................. 79  
  4.2 WSAG4J Adaptation ............................................................................................ 79  
    4.2.1 WSAG4J as a maven library ..................................................................... 79  
    4.2.2 WSAG4J Extension ................................................................................. 80  
    4.2.3 How to use the extension library ............................................................... 81  
    4.2.4 SLA Registry ........................................................................................... 83  
  4.3 Monitoring Framework ....................................................................................... 84  
    The Monitoring module ......................................................................................... 84  
    The Event Bus ...................................................................................................... 85  
  4.4 SLA Enforcement Mechanisms ........................................................................... 87  
    4.4.1 Autonomic Adjustment ............................................................................ 87  

4.4.2 Manageability Interface................................................................. 88
5 Conclusions.......................................................................................... 90
  5.1 Summary ......................................................................................... 90
  5.2 Outlook on Future Work ................................................................. 90
    5.2.1 Major requirements for the SLA@SOI framework.................... 90
6 References .......................................................................................... 93
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Service &amp; Agreement Concepts</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Service Level Agreement (SLA)</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>SLA Template</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Support for Annotations</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>SLA state diagram</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Agreement states and re-negotiation</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Guarantee term states</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Graphical representation of the association relationship</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>AdHocDemonstrator Hierarchy</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>SLA Template Registry Use Case Actors</td>
<td>36</td>
</tr>
<tr>
<td>11</td>
<td>SLA Template Design &amp; Management</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>Publishing an SLA Template</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>Revoking an SLA Template</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>Editing an SLA Template</td>
<td>38</td>
</tr>
<tr>
<td>15</td>
<td>Template-based Service Discovery Use Cases</td>
<td>38</td>
</tr>
<tr>
<td>16</td>
<td>SLA Template (Service) Discovery</td>
<td>38</td>
</tr>
<tr>
<td>17</td>
<td>SLA Template (Service) Discovery (2)</td>
<td>39</td>
</tr>
<tr>
<td>18</td>
<td>Negotiation module internal architecture</td>
<td>44</td>
</tr>
<tr>
<td>19</td>
<td>Recursive nature of negotiator invocations</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>Sequence diagram for negotiation process</td>
<td>46</td>
</tr>
<tr>
<td>21</td>
<td>Provisioning module architecture</td>
<td>47</td>
</tr>
<tr>
<td>22</td>
<td>UML sequence diagram for the createSchedule operation</td>
<td>52</td>
</tr>
<tr>
<td>23</td>
<td>UML sequence diagram for the storeSLA operation</td>
<td>53</td>
</tr>
<tr>
<td>24</td>
<td>Black-box SLA Monitoring</td>
<td>55</td>
</tr>
<tr>
<td>25</td>
<td>Grey-box architecture of the SLA Monitoring framework</td>
<td>57</td>
</tr>
<tr>
<td>26</td>
<td>Sequence diagram for SLA monitoring</td>
<td>59</td>
</tr>
<tr>
<td>27</td>
<td>Adjustment in the overall architecture</td>
<td>61</td>
</tr>
<tr>
<td>28</td>
<td>Customer fault sequence diagram</td>
<td>63</td>
</tr>
<tr>
<td>29</td>
<td>Software-triggered hardware adaptation sequence diagram</td>
<td>64</td>
</tr>
<tr>
<td>30</td>
<td>Software-triggered hardware adaptation with business notification sequence diagram</td>
<td>65</td>
</tr>
<tr>
<td>31</td>
<td>Software adaptation sequence diagram</td>
<td>66</td>
</tr>
<tr>
<td>32</td>
<td>Generalized Approach to Instrumenting Software Components</td>
<td>67</td>
</tr>
<tr>
<td>33</td>
<td>Autonomic Adjustment Architecture</td>
<td>68</td>
</tr>
<tr>
<td>34</td>
<td>Software reconfiguration scenario. Role played by the manageability interface</td>
<td>68</td>
</tr>
<tr>
<td>35</td>
<td>Relations between SLAs, SLATemplates &amp; ServiceDescription</td>
<td>71</td>
</tr>
<tr>
<td>36</td>
<td>SLA Template Registry as an Service (Description) Registry</td>
<td>72</td>
</tr>
<tr>
<td>37</td>
<td>Hierarchy of Service Descriptions (data model)</td>
<td>73</td>
</tr>
<tr>
<td>38</td>
<td>ServiceDescription Interfaces</td>
<td>74</td>
</tr>
<tr>
<td>39</td>
<td>SLA Interfaces</td>
<td>75</td>
</tr>
<tr>
<td>40</td>
<td>ServiceConstraint Interfaces</td>
<td>76</td>
</tr>
<tr>
<td>41</td>
<td>BusinessValue Interfaces</td>
<td>77</td>
</tr>
<tr>
<td>42</td>
<td>SLA Template Interface</td>
<td>78</td>
</tr>
<tr>
<td>43</td>
<td>SLA Template Registry Interface</td>
<td>79</td>
</tr>
<tr>
<td>44</td>
<td>WSAGWrapper class diagram</td>
<td>80</td>
</tr>
<tr>
<td>45</td>
<td>Exceptions class diagram</td>
<td>80</td>
</tr>
<tr>
<td>46</td>
<td>Utility classes</td>
<td>81</td>
</tr>
<tr>
<td>47</td>
<td>POM file definitions</td>
<td>81</td>
</tr>
<tr>
<td>48</td>
<td>Template validation</td>
<td>82</td>
</tr>
<tr>
<td>49</td>
<td>SLASOIEExplorer class</td>
<td>82</td>
</tr>
<tr>
<td>50</td>
<td>SLASOIEExplorer usage</td>
<td>83</td>
</tr>
<tr>
<td>51</td>
<td>Agreement objects (de)marshaling</td>
<td>83</td>
</tr>
</tbody>
</table>
Figure 52 – Monitoring module implementation: class diagram ..................... 85
Figure 53 – SLA@SOI wrapping of the Smack-Openfire API .......................... 86
Figure 54 – Event Bus implementation: class diagram ................................. 87
Figure 55: Class diagram of the Autonomic Adjustment Module ................. 88
Figure 56: Manageability Data Model .......................................................... 89
1 Introduction

A Service Level Agreement (SLA) is a representation of all features a user should expect to receive by a service. By features we not only refer to the functionality delivered by the service, but also to the quality that the user experiences. As a matter of fact, SLAs are typically associated with Quality of Service (QoS), but in a formal representation it is reasonably expected to find the service description as well.

Given the definition above, SLAs may be represented in various ways. SLA@SOI considers only their electronic representation for purposes of automated management by means of machine-readable documents. In a direct analogy with contracts as known from traditional business, we commonly refer to SLAs in our context as electronic contracts. As a consequence of this analogy, a number of important questions, such as the following, come up:

- What is the language that an electronic contract is written?
- How can the terms of such electronic contracts be expressed in uniform, commonly understood ways?
- How is outsourcing modelled in this electronic counterpart of formal contracts?
- How are electronic contracts negotiated?
- How is compliance monitored and what actions should be taken when an electronic contract is violated?

This document tries to reply to these questions and outlines the respective achievements during the 1st year of the SLA@SOI project. More specifically, it reports important current results and ongoing work within the context of Work Package A5 – “SLA Foundations and Management”. It should be noted that, as probably expected, this is work in progress which constantly evolves. The effort committed within Year 1 of the project was to build an “ad-hoc framework” which satisfies the requirements of the “Open Reference Case – ORC” and the “Ad-hoc demonstrator”, as described in deliverable D.B2a. Naturally, this document cannot discuss all details about the produced results; we try to provide a complete picture in reasonable text length.

Deliverable D.A1a describes the current overall architecture and the design motivation for it. In what follows, we outline some important design choices that affected the current design, based on the ORC. Finally, in section 5.2 of this document, we provide requirements that came as the result of the industrial use case analysis at the end of Year 1 (see deliverable D.B1a for more details). Taking into these requirements, in Year 2 we will extend and adjust the current design as needed, as further discussed in section 5.2.

Following this introduction chapter, we continue with a chapter on an A5-specific architectural discussion.

1.1 High-level design choices

In this chapter, we elaborate on a number of requirements which drive the architectural design regarding SLA management. To this end, we start from a set of ORC requirements (documented in more detail in deliverable D.B2a), and based on them devise an abstract description of the process for negotiating and establishing a complete SLA hierarchy, which consists of a top-level business SLA relying on lower-level software and infrastructure SLAs. This SLA establishment
process (as it results from the ORC requirements) has been the starting point for building the first year’s architecture. An additional source of requirements was the WS-Agreement standard. Although we aim for the final implementation to be transport-agnostic, it is reasonable to assume that web services will be the prevalent platform. Therefore, with standards-compliance being a major objective, we looked thoroughly into the assumptions of WS-Agreement.

We are assuming that service discovery is performed via SLA template discovery (which also came as a requirement from the industrial use cases). In other words, we are not simply looking for services offering specific functionality, but rather, we are looking for services which also provisionally advertise the commitments they are willing to negotiate regarding their offering as such. Concerning the negotiation design, for Year 1 only we are working under the assumption that we have only bilateral negotiations with a single customer and a single provider. Also, we do not have multi-round negotiation with contra-offers, but rather only single-offer/single-response negotiation (i.e. a “take it or leave it“ model). This was due to the fact that at the time of design, the kind of multi-round negotiation that will be supported by WS-Agreement in the upcoming release was not finalised. Additionally, multi-party negotiation is not foreseen by WS-Agreement at all. It should be noted that the GRAAP Working Group of the Open Grid Forum made final decisions on the amendment for multi-round negotiation, and the decided process will be reflected on the architectural update of the project’s Year 2.

1.1.1 SLA hierarchy establishment and activation process

Below we are providing an abstract process for establishing and activating a SLA hierarchy, with the ORC requirements and assumptions as a starting point. We describe the process in text form as a series of steps, including involved actors and SLA management artefacts in italics. We thereby assume that templates are issued by providers (not only by customers as, for instance, it happens in tenders/RFIs). This process is adapted to the terminology used by the current architecture, as documented in deliverable D.A1a.

1. The end-customer is receiving the SLA templates for a business-level service, from a respective template registry.
2. The end-customer creates an SLA offer starting from these templates, and submits the offer to the selected business service.
3. The business service is forwarding the agreement terms of the offer, to a business translation component which is responsible for converting them to software-level agreement terms for consecutive offers.
4. The translation component applies business policies to the agreement terms and therefore performs the required transformations. We are assuming here that the business-level agreement terms are not necessarily technical terms related to software and fabric, but rather arbitrarily chosen values for such technical terms based on business criteria and models.
5. The business service looks for software service templates, and chooses the appropriate ones.
6. The business service negotiation component is producing consecutive SLA offers, which are submitted to software services for negotiation. For Year 1 and the adhoc demonstrator we assume (without loss of generality) that business services do not rely on infrastructure directly. In upcoming work we will research the direct relationship that might exist between business and infrastructure services.
7. The software service negotiation component(s) are submitting *usage profiles and service level objectives* (SLOs) to the *prediction component* which provides estimates for follow-up offers. These are fed into the *translation component*, which transforms them structurally in syntactically correct agreement offer documents. In a recursive fashion, based on the results, composite software services are going through the same cycle that the business service went, to find appropriate software services for their composition, then make agreement offers to them in the same way.

8. The (non-composite) software service use prediction results, and apply them to *infrastructure service* templates retrieved from respective registry/-ies. The offers thereby produced are submitted for negotiation with the selected infrastructure service(s).

9. Based on the replies of all services down this stack, the SLAs are accepted or rejected leading to the acceptance or rejection of the top-level, end-user agreement offer.

10. SLAs that are established indeed are stored in appropriate *SLA registry/-ies*. A *monitoring component* is receiving them and produces the necessary rules for monitoring them as soon as they are activated. A *planning and activation component* is responsible for building an activation schedule, taking into account the lead times required to start new service instances.

11. In due time, the new service instances are deployed on the infrastructure and activated, taking into account the schedule previously produced.
2 Essential Concepts

2.1 Conceptual SLA Model

This section presents a conceptual model of SLAs and SLA-Templates. The purpose is to standardise the concepts we will be using, so to communicate without ambiguity. It is not implemented directly, but will drive implementation (especially in Years 2 & 3). The model is derived in large part from the WS-Agreement Specification, with the following key differences:

i) WS-Agreement is concerned specifically with Web-Services, while in SLA@SOI we are supposed to also deal with human-based services (as involved in B-line Use Cases). Hence the present conception of “service” is generic, and not bound by specific requirements for instantiation.

ii) WS-Agreement is a specification of an XML syntax for representing SLAs, and has only informal semantics – which, moreover, is in part defined in terms of the XML structure (notably; a <Location> element which refers to a sub-branch in the XML document). The present SLA Model, instead, is intended to be independent of syntactic constraints, and is entirely conceptual. No particular concrete syntax is defined or prescribed.

The SLA Model is presented as a series of UML class diagrams, as described in the following sub-sections.

2.1.1 Service & Agreement Concepts

The various high-level relations between Services, SLAs and SLA-Templates are depicted in Figure 1, and described below. We also include in this figure certain related abstract concepts (that of an agreement, a operation and a service offer), as well as involved actors (consumer, provider, customer).

![Diagram](image)

Figure 1: Service & Agreement Concepts

The class Service encapsulates a generic, or abstract, notion of “service” – loosely; a kind of action that may be performed by one entity (the service Provider) for the benefit of another (the service Consumer). The class
ServiceDescription, in contrast, denotes any concrete representational artefact which describes a Service (in more or less formal terms). A WSDL document, for example, is an instance of a ServiceDescription. The same Service may be described in many different ways (e.g. in different languages), but each ServiceDescription always describes exactly one Service.

A Service is (in part) composed of one or more (abstract) Operations – each of which encapsulates an interaction (of arbitrary complexity) between the Service Provider and Service Consumer (or more generally; agents acting on behalf of the Provider and/or Consumer). Typically, this interaction entails the Consumer requesting (or invoking) a Service Operation, which is then performed (or executed) by the Provider. But other scenarios are also possible – for example; a hotel ‘wake-up call’ Service, in which an Operation ‘call consumer’, say, would be triggered by a timer event. At the other extreme, a single Operation may be conceived as a sequence of interactions – e.g. a conversation.

Just as Services are composed of Operations, ServiceDescriptions are (minimally) composed of interaction Protocols, each of which describes (more or less formally) one or more Service Operations. Protocols may range from simple message signatures (e.g. WSDL <operation> elements) to complex workflows (e.g. as expressed in BPEL).

Any given Service may be provided by zero or more different Providers - which formulation allows us to model:

i) Services which are not, in fact, provided by any Provider – e.g. possible “future” Services such as; ‘holidays in space’.

ii) Service Queries – i.e. requests, by potential Customers, for Services which may or may not in fact be provided.

iii) competition between Providers for the provision of the same service.

The association class ServiceOffer denotes the offer of a particular Service by a particular Provider. ServiceOffers will typically be advertised in the form of publicly accessible, Provider-specific ServiceDescriptions. In particular ServiceOffers may be advertised in the form of SLATemplates (see Section 2.1.3 below) published through an SLA Template Registry.

Finally, the association class Agreement denotes an agreement between a particular Provider and a particular Customer concerning the provision of a particular Service to one or more Service Consumers. The Customer acts as signatory to the agreement on behalf of the individual Consumers, and must therefore be invested with appropriate authority to represent those Consumers. The term “agreement”, as used here, denotes both potential & actual agreements. A potential agreement (a mere proposal or offer) is transformed into an actual agreement (with legal effect), by a formal public act of agreement (such as a handshake, or witnessed signing). A Service Level Agreement (SLA; see Section 2.1.2) is a concrete representational artefact (a sub-species of ServiceDescription) which describes, or codifies, an Agreement.

### 2.1.2 Service Level Agreements (SLA)

As stated above, an SLA is a concrete representation, or codification, of an Agreement. For present purposes, we treat an Agreement as essentially defining a set of Obligations on the part of Service Provider & Customer (the parties to the Agreement). In general, an Obligation will entail:

i) a performative aspect – i.e. that which the obligated party is obligated to do (or ensure is the case), together with
ii) some form of negative utility (e.g. a punishment - typically; a pecuniary cost) suffered by the obligated party in the case that the obligation is not met.

Obligations may also obtain a priority ranking according to their relative severity. Within an SLA, Obligations are codified through GuaranteeTerms – as depicted in Figure 2, and described below.

**Figure 2 : Service Level Agreement (SLA)**

An SLA comprises one or more GuaranteeTerms, where each GuaranteeTerm codifies an Obligation. The performative aspects of the Obligation are encapsulated by the ServiceConstraint class, while utility is captured by the BusinessValue\(^1\) class. In order to ensure that Obligations are met – i.e. that the Service as delivered in fact corresponds to the Service as agreed – the Service delivery must be monitored. To this end the SLA further comprises a set of monitoring policies – encapsulated by the class MonitoringPolicy. The classes ServiceConstraint, BusinessValue & MonitoringPolicy are described in the following sections.

**Service Constraints**

A ServiceConstraint is a (more or less formal) expression which asserts some constraint over one or more Observable properties of the Service provision and execution. A property is Observable just if it can be observed – i.e. if it can be measured. Such properties include, among others:

i) **performance aspects** – e.g:
   a. response-time of Operations,
   b. queueing delay,
   c. jitter,
   etc.

ii) **context** – e.g:
   a. time of invocation of an Operation,
   b. ID of the Consumer invoking the Operation,
   c. geographical location,
   d. temporal ordering of invocations,
   etc.

iii) **statistical measures** – e.g:

\(^1\)The term “business value” is taken from WS-Agreement
a. average-response-time,
b. frequency of invocations,
c. reliability,
d. availability,

etc.

It should be emphasised that the notion of ServiceConstraint, as employed here, is generic. In particular, it includes common quality-of-service (QoS) notions such as Service Level Objective (SLO) or KPI-Targets – both of which may be captured as sub-species of ServiceConstraint (see Comment 1). A normative list of QoS terms is presented in Section 2.2.

**Comment 1 : Function versus Quality**

It should be noted that the SLA Model presented here makes no explicit distinction between function & quality as applied to Services – which requires some justification. In short, although we can intuitively draw such a distinction, it is difficult to pin down in formal, general terms precisely what such a distinction might entail. Instead, we take the view that the distinction is essentially context dependent. Specifically; the notion of what constitutes quality, in respect of a particular Service, depends just on what function that Service provides.

To illustrate, consider the notion of response-time, which is often considered a QoS property. Now if the function of a Service can be defined in purely computational terms (i.e. as a sequence of time-independent state-transitions), then treating time as a qualitative measure is justifiable (since the time taken to move between states is computationally irrelevant). If, however, the notion of response-time is fundamental to the function of the Service – e.g. consider a clock, or timer – then response-time must per force be treated as a functional property. In short, the same property can be qualitative in one context and functional in another – and there seems to be no generic, context-independent means to distinguish the two.

As such, we conceive only of a generic ServiceConstraint operating over generic Observable properties (this abstraction is sufficient, for example, to support the notion of monitoring).

In principle, ServiceConstraint expressions may assert anything over any number of Observables, and may be arbitrarily complex - e.g. comprising any logical combination of (nested) sub-expressions. Following are just a few illustrative examples of possible ServiceConstraints (assuming X & Y are numerically quantifiable Observables, and Q & P are spatio-temporal events). We treat a spatio-temporal event as a particular kind of Observable, corresponding to a fact about a change in the world.

- X < 5
- 0 < X < 5
- X < 5 AND Y > 10
- for all values of X measured between times T1 and T2, X = 4
- Q occurs before P
- if Q (is the case) then X > 8, else X < 2

Each GuaranteeTerm is associated with a single ServiceConstraint specifying the performative aspects of the Obligation codified by that GuaranteeTerm. In addition, each GuaranteeTerm may also be associated with a precondition (or, to use the terminology of WS-Agreement; a ‘qualifying condition’) – specifying what must be the case in order for the Obligation to obtain. These preconditions are
also captured by the ServiceConstraint class. So, for example, a GuaranteeTerm might guarantee that: *the completion time of a particular Service Operation, X, will be less than 5ms* (the guaranteed ServiceConstraint), *but only if the number of invocations of X per hour does not exceed 100* (the precondition).

**Business Values**

A single GuaranteeTerm may be associated with one or more BusinessValues – which serve to capture the utility, or value, aspects of the Obligation. As noted earlier, this term’s name has been chosen according to WS-Agreement terminology and is not directly related to WP A2, which looks into business-related issues of the SLAs. Nevertheless, Business Value provides a placeholder that MAY be used for describing any particular kind of:

i) **Cost**: which comes in two flavours:
   a. *penalty*: denoting a punitive measure on the obligated party in the case the obligation is not fulfilled, and
   b. *reward*: denoting (potential) benefits for the obligated party in the case the Obligation is fulfilled.

ii) **Priority**: denoting a “preference” given to each Obligation in case there is a conflict between multiple Obligations which can not be simultaneously fulfilled. Note that priorities may be implicitly defined by cost assertions (e.g. preference might be given to obligations which carry higher penalties).

BusinessValues may be applied to *either* Provider or Customer – such that the specification of a BusinessValue must make explicit the intended ObligatedParty.

Examples of Provider penalties include:
- a reduction in the Service price to a Customer (e.g. applied directly in the billing process),
- an upgrade in ‘service level’ (e.g. moving the Consumer from a low quality ‘bronze’ band to high quality ‘gold’ band),
- the issue of ‘credits’ to the Customer – e.g. discounts on the future use of Services offered by that Provider.

Examples of penalties applied to the Customer, instead, include:
- an increase in Service price,
- a degradation of Service conditions – e.g. reducing the quality of Service, or making the Service temporarily unavailable.

More specific examples are provided in Comment 2.

**Comment 2 : Examples of BusinessValue**

**Obligation on the Provider:**

A Service Provider is obligated to provide a 99% level of availability. If availability is < 99% then a discount in the Customer bill is applied in the next billing period equal to the percentage of unavailability (i.e. if availability is only 90% then the bill is reduced 10%), or credits with a discount equivalent to 10% of the Customer’s current bill are awarded to the Customer, who can apply them to any future billing period. Another option may be to upgrade the Customer to a higher level of service at the same price (with faster response time for instance or a 99,9% of availability).
Obligation on the Customer:

A Customer has a cap on the amount of data that can be downloaded/uploaded from/to a file-sharing service (e.g. 10 GB / Month). Once it is surpassed then each GB extra is charged 10% of the total bill without limit, or until a maximum of 150% of the bill is reached. It also may be the case that once the limit is surpassed then the bandwidth is reduced so that DL / UL speeds are set to 10% of the original agreed speed (with no impact on the bill or limitation on the amount of data downloaded).

Illustration of Priority:

A Service Provider realises that, due to unforeseen circumstances, they cannot simultaneously fulfill obligations in respect of:

i) response-time, and
ii) the number of requests per minute.

In this case the SLA has defined (per Consumer request/invocation) that response time is more important than number of requests per minute – with the result that; the Provider adjusts the provisioning and/or execution of the Service so that some requests are rejected in order to ensure that those which are accepted receive a response within the predetermined time.

In another scenario, no explicit Priority has been defined. The Provider realises, however, that rejecting requests has an associated penalty which is much more onerous than the one associated with delayed response time. As such, the Provider makes adjustments so that all Consumer requests are accepted and queued to be processed later, even if that requires a longer response time than agreed.

Monitoring Policies

The class MonitoringPolicy encapsulates any information required to configure & effect the monitoring of Service delivery.

Typically, a MonitoringPolicy defines one or more MonitoringActions – where each MonitoringAction is either:

i) an observation: i.e. the taking of a measurement of the value of an Observable,
ii) a verification: checking that measured values of Observables meet guaranteed ServiceConstraints – i.e. that the performative aspects of Obligations are fulfilled.

As example, consider a GuaranteeTerm which guarantees a certain minimum average response time on a particular Service Operation. Observation in this case could be implemented as the collection of timestamps of the individual invocations & responses effecting the Operation. Verification would then entail computing the average response time on the basis of these timestamps, and then checking this calculated value against the guaranteed ServiceConstraint. Either MonitoringAction or Observable may contain a specification of calculating observable values.

In addition to ‘pure’ MonitoringActions, a MonitoringPolicy might specify:

- the particular actors responsible for carrying out the monitoring,
- temporal validity: the application of a given MonitoringPolicy may be limited to a specific time-window (e.g. to ‘peak hours’ only), and an SLA might specify that different policies apply at different times,
• **SLA violations reporting**: violations may be reported as soon as they occur, or at periodic (preset) intervals. Additionally, in the case of *pre-emptive* monitoring, reports may be triggered when the probability of violating an Obligation exceeds some predefined threshold.

• **ControlActions**: a MonitoringPolicy may also specify various actions that must be taken in order to recover from a violation. Such ControlActions may be either *pre-emptive* (preventive) or *corrective* (*after-the-fact*).

In principle, a MonitoringPolicy may specify anything considered relevant to ensuring the Obligations defined in the SLA are fulfilled.

### 2.1.3 SLA Templates

In practical terms, an SLATemplate can be thought of as a *customisable* SLA — i.e. an SLA in which certain terms or values are left unspecified, and treated as *free-variables* (which may, or may not, be constrained to take certain values). These free-variables may occur in any of the constituent parts of an SLA — i.e. in GuaranteeTerms, ServiceConstraints, BusinessValues or in MonitoringPolicies.

At a conceptual level we can abstract this practical notion by conceiving a SLATemplate as essentially specifying, or representing, a *set of possible SLAs* — namely; those SLAs which can be created (i.e. generated) just by assigning particular values to the free-variables\(^2\). This high-level conception is difficult to depict in any simple way in UML, but Figure 3 provides a rough approximation (described below).

![Figure 3: SLA Template](image)

**Figure 3 : SLA Template**

The (greyed) specialisation arrow shown in Figure 3 should not be taken *literally* as asserting that an SLATemplate is *conceptually a* kind of SLA. Instead, the intention is just that the *representational artefacts* constituting SLATemplates form a sub-class of the *representational artefacts* constituting SLAs. Even though these representations have comparable forms, however, the concepts they represent are very different. Conceptually, a SLATemplate *represents a set of SLAs*, while an individual SLA *codifies an Agreement*.

The key differences between SLATemplates and SLAs, as *representations*, are that SLATemplates may include:

\(^2\) As an example, a variable could be the numerical value for the Completion Time of the service’s operation; then, this could be constrained accordingly, to accept sensible values.
i) one or more (free-)Variables (as constituents of GuaranteeTerms, ServiceConstraints, BusinessValues or in MonitoringPolicies), and
ii) one or more CreationConstraints - each of which encapsulates a constraint over the values that embedded Variables might take.

### 2.1.4 Support for Annotations

The SLA Model described above is concerned only with generic aspects of SLAs & SLATemplates – i.e. with the essential properties & relations which we consider to apply across all application domains. Application specific extensions are supported through annotations, which are in turn supported by introducing the ground notion of a *uniquely identifiable named entity* – as encapsulated by the *root* class Nominal depicted in Figure 4. The class Annotation denotes any application-specific annotation (or metadata) serving to describe a Nominal.

![Figure 4: Support for Annotations](image)

### 2.2 Standard QoS Terms

One of the requirements of a SLA management framework is to have a well-defined, extensible set of agreement terms. The core meta-model from WP A1 and the SLA model of section 2.1 address the “extensible” part of this requirement. However, for the “well-defined” part, we have to provide a set of definitions that are either explicitly requested by the use cases (e.g. availability, completion time, etc) or they are commonly referred to in literature. We have implemented such formal definitions and present them in the following tables. It should be stressed, at this point, that it is not in the scope of Year 1 to implement all these QoS terms as guarantees for the ORC and the ad-hoc demonstrator; rather, a very restricted set of them is applied. It is the purpose of the project to implement all of them in the following two years. One thing to note before listing the terms, is that it is not straightforward to isolate them completely from the context of a SLA. For instance, throughput may refer to steady or burst rate; therefore, a SLA about it would include constructs (e.g. elasticity) that define the term in complete.

---

3 the notion of ‘creation constraint’ is taken from WS-Agreement.
<table>
<thead>
<tr>
<th>Term name</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>The probability that the service is up and running.</td>
</tr>
</tbody>
</table>
| Term definition | Assuming  
  • service S;  
  • time $T_1$ as the beginning of monitoring time;  
  • time $T_2$ as the time of evaluating availability;  
  • monitoring duration $T = T_2 - T_1$;  
  • $b_i$ as a time when S could not be invoked any more, by all of its (established or potential) customers, due to reasons other than network connectivity, where $T_1 \leq b_i \leq T_2$;  
  • $e_i$ as the moment when S became usable again following $b_i$, where $T_1 \leq e_i \leq T_2$;  
  • $d_i = e_i - b_i$;  
  • $d = \sum d_i$  
We then define availability for service S as $A = \frac{T - d}{T}$. |
| Negotiable values | $0 \leq A \leq 1$, $A \in \mathbb{R}$ |
| Dependencies | |

<table>
<thead>
<tr>
<th>Term name</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>The probability that the service is accessible. By &quot;accessible&quot; we refer to the ability to complete an invocation request, leading either to a state change or to a return message (independent of its kind, i.e. success, failure/error, information, warning, etc).</td>
</tr>
</tbody>
</table>
| Term definition | Assuming  
  • operation $O$ of service S;  
  • time $T_1$ as the beginning of monitoring time;  
  • time $T_2$ as the time of evaluating accessibility;  
  • monitoring duration $T = T_2 - T_1$;  
  • $R_a$ as the number of all invocations to $O$ during time $T$;  
  • $R_d$ as the number of invocations that were not served (i.e. were dropped) during time $T$;  
We then define accessibility for operation $O$ as $C_O = \frac{R_a - R_d}{R_a}$. |
| Negotiable values | $0 \leq C_O \leq 1$, $C_O \in \mathbb{R}$ |
| Dependencies | |

<table>
<thead>
<tr>
<th>Term name</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>The economic aspect of invoking a service in general, or a specific operation of it.</td>
</tr>
</tbody>
</table>
| Term definition | Assuming  
  • a value metric $M$, commonly understood and explicitly or implicitly agreed upon by both parties involved;  
  • operation $O$ of service $S$  
  • a well-defined and unambiguous assignment of the value $V$ (measured in $M$), either per invocation of $O$, or per time unit of using service $S$;  
  • the conditions $K$ under which the value assignment is true,  
then cost may be defined in one of the following ways:  
  • per $N$ invocations of $O$: $C_O = N \cdot V \cdot M$ iff $K$. |

---

Term name: **Availability**

**Term description**: The probability that the service is up and running.

**Term definition**: Assuming:
- service $S$;
- time $T_1$ as the beginning of monitoring time;
- time $T_2$ as the time of evaluating availability;
- monitoring duration $T = T_2 - T_1$;
- $b_i$ as a time when $S$ could not be invoked any more, by all of its (established or potential) customers, due to reasons other than network connectivity, where $T_1 \leq b_i \leq T_2$;
- $e_i$ as the moment when $S$ became usable again following $b_i$, where $T_1 \leq e_i \leq T_2$;
- $d_i = e_i - b_i$;
- $d = \sum d_i$

We then define availability for service $S$ as $A = \frac{T - d}{T}$.

**Negotiable values**: $0 \leq A \leq 1$, $A \in \mathbb{R}$

**Dependencies**: 

---

Term name: **Accessibility**

**Term description**: The probability that the service is accessible. By "accessible" we refer to the ability to complete an invocation request, leading either to a state change or to a return message (independent of its kind, i.e. success, failure/error, information, warning, etc).

**Term definition**: Assuming:
- operation $O$ of service $S$;
- time $T_1$ as the beginning of monitoring time;
- time $T_2$ as the time of evaluating accessibility;
- monitoring duration $T = T_2 - T_1$;
- $R_a$ as the number of all invocations to $O$ during time $T$;
- $R_d$ as the number of invocations that were not served (i.e. were dropped) during time $T$;

We then define accessibility for operation $O$ as $C_O = \frac{R_a - R_d}{R_a}$.

**Negotiable values**: $0 \leq C_O \leq 1$, $C_O \in \mathbb{R}$

**Dependencies**: 

---

Term name: **Cost**

**Term description**: The economic aspect of invoking a service in general, or a specific operation of it.

**Term definition**: Assuming:
- a value metric $M$, commonly understood and explicitly or implicitly agreed upon by both parties involved;
- operation $O$ of service $S$;
- a well-defined and unambiguous assignment of the value $V$ (measured in $M$), either per invocation of $O$, or per time unit of using service $S$;
- the conditions $K$ under which the value assignment is true,

then cost may be defined in one of the following ways:
- per $N$ invocations of $O$: $C_O = N \cdot V \cdot M$ iff $K$. 

---
- flat-rate for contract duration: $C_S = c t$ iff $K$
- dependent on time: $C_S = N \cdot V \cdot M$ iff $K$, where $N = [(\text{contract duration}) / (\text{usage time unit})]$

**Negotiable values**

$C \in \mathbb{R}$

**Dependencies**

A well-defined, commonly understood dictionary of conditions $K$.

<table>
<thead>
<tr>
<th>Term name</th>
<th>Term definition</th>
</tr>
</thead>
</table>
| **Precision**    | **Assuming**
|                  | • operation $O$ of service $S$;  
|                  | • $O$ returns result $V$  
|                  | • $V \in \mathbb{C}$, $V = a + bi$  
|                  | • $\{x\} = x - \lfloor x \rfloor$  
|                  | • $\{a\} \in \mathbb{R} \land \{a\} \cdot 10^n \in \mathbb{Z}$, $n \in \mathbb{N}$  
|                  | • $\{b\} \in \mathbb{R} \land \{b\} \cdot 10^m \in \mathbb{Z}$, $m \in \mathbb{N}$  
|                  | we then define precision for operation $O$ as $U = \max(n, m)$

| **Performance / Throughput** | **Term description** | Maximum invocation rate for a specific service operation.  
|-------------------------------|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| **Term definition**           | **Assuming**
|                                | • operation $O$ of service $S$  
|                                | • time unit $t$  
|                                | • request arrival rate $R = N/t$, $N=$number of requests per time unit $t$, $N \in \mathbb{N}$  
|                                | • accessibility $C_O=1$ for $R = R_1$  
|                                | • accessibility $C_O<1$ for a $R = R_2$, $R_2 > R_1$  
|                                | we then define throughput for operation $O$ as $H_O = R_1 / t$

| **Performance / Completion time** | **Term description** | The time between receiving an invocation message on the service end, and completing the construction of a response; measured in time units.  
|-----------------------------------|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| **Term definition**               | **Assuming**
|                                  | • operation $O$ of service $S$  
|                                  | • request message $M_Q$ of a client to the service $S$ for the invocation of operation $O$  
|                                  | • response message $M_R$  
|                                  | • $M_Q$ received in full on the service end at time $t_i$  
|                                  | • $M_R$ put on the wire in full at time $t_0$  
| We then define completion time $T_{CO} = t_0 - t_i$  
| Assuming a series of Completion Time measurements, $T_{CO1}$, $T_{CO2}$, ... $T_{CON}$, we define Average Completion Time as $T_{AO} = (\Sigma T_{CO}) / n$. |

| **Configuration / Supported standards** | **Term description** | Used to confirm interoperability capabilities; consists of a list of standards / versions.  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Term definition</td>
<td>Assuming</td>
<td>Negotiable values</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>-------------------</td>
</tr>
<tr>
<td></td>
<td>• Set L of standards specifications, ( L = (L_1, \ldots, L_n) )</td>
<td>( Q = (Q_1, \ldots, Q_m) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ( L_i ) a fully qualifying name, e.g. “ISO8859-7”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Subset Q of set L, ( Q = (Q_1, \ldots, Q_m) ), ( m \leq n )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ( Q_i ) fully implemented by S ( \forall i \in (1, \ldots, m) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>We then define Q to be the list of standards supported by S.</td>
<td></td>
</tr>
</tbody>
</table>

Term name | Configuration / Regulatory
---|---
Term description | Indicates compliance with national/international regulatory law.

Term definition | Assuming | Negotiable values | Dependencies |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Set L of regulatory specifications, ( L = (L_1, \ldots, L_n) )</td>
<td>( V = (V_1, \ldots, V_m) )</td>
<td>A commonly understood dictionary of regulatory specifications that may apply in the domain at hand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ( L_i ) a fully qualifying name, e.g. “31994Y0702(01)” or “OJ C 181, 2.7.1994, p. 1–2”</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Subset V of set L, ( V = (V_1, \ldots, V_m), m \leq n )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ( V_i ) fully respected by S ( \forall i \in (1, \ldots, m) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>We then define V to be the list of regulatory specifications supported by S.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Term name | Integrity
---|---
Term description | Represents ability of service to preserve data integrity.

Term definition | Assuming | Negotiable values | Dependencies |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Given a set of service transaction protocols, integrity is synonymous to Q (supported standards) including one or more of those.</td>
<td>( Q = (Q_1, \ldots, Q_n), Q_i \in (&quot;WS-C/WS-Tx&quot;, &quot;BTP&quot;, ...) )</td>
<td>A commonly understood dictionary of transaction protocols available, including acceptable combinations of subsets of their functionality.</td>
</tr>
</tbody>
</table>

Term name | Reliability / Mean Time To Repair (MTTR)
---|---
Term description | A time value indicating how much time it takes, on average, from the moment the service becomes unavailable to the moment it is available again.

Term definition | Assuming | Negotiable values | Dependencies |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Service S</td>
<td>( MTTR \geq 0, MTTR \in \mathbb{R} )</td>
<td>MTBF = MTTR + MTTF</td>
</tr>
<tr>
<td></td>
<td>A moment in time, ( t_b ), that the service becomes unavailable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The respective moment in time, ( t_e ), that it becomes available again</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The period (duration) of unavailability, ( t = t_e - t_b )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A series of such periods, ( T = (t_1, t_2, \ldots, t_n) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The total unavailability time ( u = \Sigma t_i )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>We then define ( MTTR = \frac{u}{n} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term name</td>
<td>Reliability / Mean Time To Failure (MTTF)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term description</td>
<td>A time value indicating, on average, how much time it takes from one service failure to the next.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term definition</td>
<td>Assumptions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A restoration after failure for this service, taking place at time $t_b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The consecutive failure of the service, starting at time $t_e$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The respective period of availability $t = t_e - t_b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A series of such periods, $T = (t_1, t_2, ..., t_n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The total duration of service availability, $u = \sum t_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>We then define $MTTF = \frac{u}{n}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negotiable values</td>
<td>MTTF $\geq 0$, MTTF $\in \mathbb{R}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>MTBF = MTTR + MTTF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term name</th>
<th>Security / Auditableity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>Indication of auditable logs being maintained.</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assumptions</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• Set I of auditable pieces of information (e.g. originating IP address, time of request), $I = (I_1, ..., I_n)$</td>
</tr>
<tr>
<td></td>
<td>• Set K of preconditions that confirm the validity of auditable information (e.g. NTP synchronisation, read-only user access to auditable data, etc)</td>
</tr>
<tr>
<td></td>
<td>• Subset $D$ of set $I$, $D = (D_1, ..., D_m)$, $m \leq n$</td>
</tr>
<tr>
<td></td>
<td>• $D_i$ logged by $S$ $\forall i \in (1, ..., m)$</td>
</tr>
<tr>
<td></td>
<td>We then define $(D, K)$ as the auditability of $S$.</td>
</tr>
<tr>
<td>Negotiable values</td>
<td>$((D_1, ..., D_m), (K_1, ..., K_m))$</td>
</tr>
<tr>
<td>Dependencies</td>
<td>A commonly understood dictionary of auditable pieces of information $I$;</td>
</tr>
<tr>
<td></td>
<td>A commonly understood dictionary of preconditions $K$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term name</th>
<th>Security / Authentication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>Indication of proper authentication method(s) being used.</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assumptions</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• Customer C</td>
</tr>
<tr>
<td></td>
<td>• Uniquely identifying information $ID_i$ of type $T_i$ for this customer, either for the specific service, or for a broader domain of services (possibly all available services through unique world-wide identification)</td>
</tr>
<tr>
<td></td>
<td>• A well-defined methodology $M_i$, which accepts input of type $T_i$ and certifies it as correct (acceptable credentials) or incorrect (unacceptable/false credentials)</td>
</tr>
<tr>
<td></td>
<td>• A set of all possible methodology/input type pairs $P = ((M_1, T_1), ..., (M_n, T_n))$</td>
</tr>
<tr>
<td></td>
<td>• A subset $N$ of $P$, $N = ((M_1, T_1), ..., (M_k, T_k))$, $k \leq n$</td>
</tr>
<tr>
<td></td>
<td>We then define $N$ as the authentication indicator of $S$, iff $S$ offers all methods $M_i$ of $N$ for the identification of users.</td>
</tr>
<tr>
<td>Negotiable values</td>
<td>$N = ((M_1, T_1), ..., (M_k, T_k))$</td>
</tr>
<tr>
<td>Dependencies</td>
<td>A commonly understood dictionary of (authentication method / input data type) pairs.</td>
</tr>
<tr>
<td>Term name</td>
<td>Security / Authorisation</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Term description</td>
<td>Indication of proper authorisation framework &amp; rules being used.</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assuming</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• Customers C = (C₁, ..., Cₙ)</td>
</tr>
<tr>
<td></td>
<td>• Well-defined set R of m resources of service S, R = (R₁, ..., Rₘ) exposed and accessible by customers (Cᵢ), 1 ≤ i ≤ n, but not by customers (Cⱼ), 1 ≤ j ≤ n</td>
</tr>
<tr>
<td></td>
<td>• C = Cᵢ ∨ Cⱼ</td>
</tr>
<tr>
<td></td>
<td>• Cᵢ, Cⱼ MAY be empty sets</td>
</tr>
<tr>
<td></td>
<td>• A well-defined set of rules (possibly undisclosed to customers) which define unambiguously which customers are granted access (or are not granted access) to each resource</td>
</tr>
<tr>
<td></td>
<td>• A set B = (B₁, ..., Bₘ), Bᵢ ∈ (true, false), 1 ≤ i ≤ m</td>
</tr>
<tr>
<td></td>
<td>We then define Z, the authorisation indicator of the service, to be the mapping function Z(C, R) → B.</td>
</tr>
</tbody>
</table>

**Negotiable values**

Domain-specific

**Dependencies**

<table>
<thead>
<tr>
<th>Term name</th>
<th>Security / Data encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>Indicates encryption capabilities for data management (transfer, storage, or other)</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assuming</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• A well-defined methodology Mᵢ which accepts input of type Tᵢ and produces encrypted output of type Tⱼ that can be then decrypted using well-defined methodology Mⱼ</td>
</tr>
<tr>
<td></td>
<td>• A set of all possible methodology/input type pairs P = ((M₁, T₁), ..., (Mₙ, Tₙ))</td>
</tr>
<tr>
<td></td>
<td>• A subset N of P, N = ((M₁, T₁), ..., (Mₖ, Tₖ)), k ≤ n</td>
</tr>
<tr>
<td></td>
<td>• The set of m resources of service S (e.g. network stack, storage) that support each such methodology, R = (R₁, ..., Rₘ)</td>
</tr>
<tr>
<td></td>
<td>We then define E = N × R as the data encryption capabilities of service S.</td>
</tr>
</tbody>
</table>

**Negotiable values**

N = ((Mᵢ, Tᵢ, Rⱼ)), 1 ≤ i ≤ k, 1 ≤ j ≤ m

**Dependencies**

A commonly understood dictionary of (encryption method / input data type) pairs, and a common naming scheme for service resources.

<table>
<thead>
<tr>
<th>Term name</th>
<th>Security / Non-repudiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>Indication whether it is possible for a service user or a service provider to deny use of the service by the user.</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assuming</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• M, a well-defined, commonly understood methodology which certifies the authenticity of a signature and/or the integrity of data signed</td>
</tr>
<tr>
<td></td>
<td>• M = (Mᵢ, ..., Mₙ) the set of all such methodologies known to both parties (customers/providers),</td>
</tr>
<tr>
<td></td>
<td>We then define R = (Mⱼ, ..., Mₖ), 1 ≤ j,k ≤ n, as the non-repudiation indicator of service S.</td>
</tr>
</tbody>
</table>

**Negotiable values**

R = (Mⱼ, ..., Mₖ), 1 ≤ j,k ≤ n
### Dependencies

A commonly understood dictionary of methodologies to certify the authenticity of a signature and/or the integrity of data signed.

<table>
<thead>
<tr>
<th>Term name</th>
<th>Security / Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term description</td>
<td>Indication whether it is possible to have ProviderAgents executed on different physical (or virtual) machines.</td>
</tr>
<tr>
<td>Term definition</td>
<td>Assuming</td>
</tr>
<tr>
<td></td>
<td>• Service S</td>
</tr>
<tr>
<td></td>
<td>• Customer set C = (C₁, ..., Cₙ)</td>
</tr>
<tr>
<td></td>
<td>• Provided to the customers through separate processes, P = (P₁, ..., Pₙ), for each customer</td>
</tr>
<tr>
<td></td>
<td>• These processes are executed on s independent systems (e.g. virtual-machines), M = (M₁, ..., Mₙ)</td>
</tr>
<tr>
<td></td>
<td>• The pair (Cᵢ, Mⱼ), 1 ≤ i ≤ n, 1 ≤ j ≤ s, represents the fact that the processes of customer Cᵢ are executed on system Mⱼ. We extend this notation to accept sets of customers and systems, to denote that the processes of all customers in the set are executed on the systems of the respective set.</td>
</tr>
<tr>
<td>Negotiable values</td>
<td>True, False</td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
</tr>
</tbody>
</table>

#### 2.3 On The Use Of Semantics

Service level agreements must allow the computer-assisted specification of meaning in order for the involved parties to agree as to the success or failure in their implementation. The natural question arises as to whether logic programming techniques ("semantics") could be of assistance in the SLA@SOI framework in the mediation of such meaning. This question has formed a constant concern in the unfolding of our research, so we now briefly describe the results of this question. A loose characterization of the consensus reached by our consortium is that computer-assisted semantics should best be viewed not as an architectural foundation as some of the more zealous proponents of the Semantic Web proclaim, but rather one technique among many that may or may not be appropriate to the task and should thus be evaluated as to its suitability on a case by case basis.

We now provide some insight into the treatment of the question of semantics by the consortium by first noting where, after the completion of the first year, computer-assisted semantics have been used by the consortium. We then conclude with a short synopsis on the argument as to why semantics were not adopted comprehensively in the specific area of the modelling and implementation of SLA guarantee terms.

#### 1.1.1 Uses of Semantics in SLA@SOI

At the end of the first year, the consortium has employed the use of computer-assisted semantics in the following areas:

1. The prototype of the SLA template registry uses an embedded Prolog system hosted on the JVM for its implementation. The API provided to the rest of the SLA@SOI framework does not expose the semantic based facet of the implementation, as it was not identified as a requirement in the
component design process. In the future this prototype may be replaced by a system not so primarily associated with computer-assisted semantics as Prolog, but which provides equivalent functionality. This provides an example of the use of semantics as the best technique for the given task with the given resources.

2. In the closest match to the technology stack promoted under the Semantic Web banner, the Business SLA Ontology—initially modelled in UML—was converted to an OWL-based ontology representation to make it amendable to use in business rules as specified in SWRL (Semantic Web Rules Language). This provides an example where we identified a specific standards-based technology (semantic rules engines) we wished to use, and were able to transform the consortium-wide standard for modelling (UML), to the necessary artefacts for implementation.

3. In the monitoring engine, the rule generator module developed by CITY utilizes a language based on the event calculus to reason upon the temporal ordering of events. The event calculus provides the example of a programming logic technique not directly related to any of the technologies standardized by the Semantic Web community, but nevertheless applied by the SLA@SOI consortium as the best technique for the job.

1.1.2 Guarantee Terms

In this section we provide some insight into our considerations of semantics with SLA@SOI by examining the more restricted question of whether to apply semantics to the SLA guarantee terms. Within the SLA@SOI model, our use of “guarantee terms” are more or less synonymous with those that occur in WS-Agreement, so we consider here the slightly broader question of whether to adopt an ontologization of WS-Agreement. We now present a short commentary on our survey of the relevant research literature.

A translation of the XML syntactic basis of WS-Agreement into OWL-S is presented in [1]. The result forms an ontologized version of the subsumption hierarchy (i.e statements of the A subClassOf B, etc.) implicit in the WS-Agreement specification, but fails to add any rules (i.e. axioms) that would be of help in enforcing the mediation of semantics between ontologizations arising from differing organizations. A more complete implementation is described in [2], but still the necessary axioms (rules) necessary to provide help in mediation, need to be created in addition. Insight as to why only a non-axiomized subsumption hierarchy can be derived from WS-Agreement is provided in [3] which in part provides arguments as to the justification of the lack of formal semantics in WS-Agreement. The RBSLA (“rules-based SLA”) system described in [4] seems to be the most advanced out of our survey and provided an open source implementation which we could audit, but we were overwhelmed by the sheer complexity of its implementation as this system uses no less than seven distinct logical formalisms in its implementation.

To summarize, we found that a useful ontologization of WS-Agreement will always require additional work to specify the rules of interaction as the formal semantics of WS-Agreement are not provided in its specification. From our survey of the relevant literature, we judged that using semantics to specify these rules required much more effort than was justified in a project whose focus was on the SLA lifecycle as opposed to the use of semantics.

We would like to note in closing that contemporaneous with SLA@SOI, the research from the 6FP BREIN project [5] targets the adaption of grids to business technologies. BREIN is already developing a semantically annotated, WS-
Agreement based notion of SLAs. In the future, we will continue monitoring developments in this front, and evaluate the possibility of integrating this project’s results.

2.4 SLA Lifecycle

The state life-cycle for SLAs is directly linked to the SLA model of section 2.1, and adopted more or less intact from the WS-Agreement specification (specifically; WS-Agreement Specification, March 14 2007). There are four key differences:

1) we have interpreted the WS-Agreement term “PendingAndTerminated” to denote a (provisional) withdrawal, by the initator, from the agreement. Subsequently we use the term ‘withdrawn’ in favour of the WS-Agreement term.

2) likewise, we have interpreted the WS-Agreement term “ObservedAndTerminated” to denote a (provisional) annulment, by either initator or responder, from the agreement. Subsequently we use the term ‘annulled’ in favour of the WS-Agreement term.

3) the most significant difference is that we have extended the WS-Agreement state ‘observed’ – largely to take account the possibility of renegotiation (explained further below),

4) following from ‘iii’, we also introduce the new state ‘deprecated’ to signify an SLA which has been renegotiated and is no longer in effect.

The complete SLA State Diagram is shown in Figure 5. The WS-Agreement specification should be treated as primary reference – but for convenience, we provide brief summaries later in this document.

One thing that should be noted, is related to renegotiation and how it affects SLA states. The state-diagram allows that a “renegotiated” SLA may still be monitored. E.g. consider an effective SLA whose end-time is, say, January 1st 2010. Prior to this, say January 1st 2008, the SLA is renegotiated – with the new SLA coming into effect on (i.e. start-time equal to) January 1st 2009. The question then, is what is the status of the ‘renegotiated’ (previous) version during the period January 1st 2008 (date new SLA was agreed) to January 1st 2009 (date new SLA comes into effect). In the present model we have assumed that the previous version continues to be effective (monitored) until the new version comes into effect – but this is not the only possibility. It is likely that alternative scenarios will have to be considered & explicitly modelled.
2.4.1 Description of SLA States

OfferReceived

The initial state `Agreement Proposed` corresponds to the WS-Agreement ‘dummy’ state “OfferReceived” – intended simply to “..clarify that exposed initial states can be ‘Pending’, ‘Observed’ or ‘Rejected’ ..” (we use the term ‘agreement proposed’ just to avoid confusing an ‘agreement offer’ with a ‘service offer’). The event leading to this state is the proposition of an agreement (“agreement offer”) from the initiator to the responder, making use of the template which the responder made available to the world.

Pending, Rejected & Withdrawn

The agreement moves to a `Pending` state if/when the responder decides to defer the acceptance or rejection of the proposed agreement to a later time.

Following the `Pending` state or perhaps directly after receipt of the proposed agreement, a rejection by the responder changes the SLA state to `Rejected`, which leads to SLA destruction/cleanup immediately.

The `Withdrawn` state indicates that, while in a `Pending` state, the initiator has opted to withdraw the proposed agreement. For consistency with WS-Agreement the `Responder` is free to either accept the initiator’s withdrawal (& terminate the agreement), or deny it (returning the agreement to the `Pending` state).
**Observed, Agreed & Provisioned**

The (WS-Agreement defined) state *Observed* is treated here as comprising the sub-states *Agreed* (initial sub-state), *Provisioned* & *Monitored*. While in any of these substates:

- the agreement may be unilaterally terminated by either party, or
- either party may propose to annul the agreement – subject to acceptance by other party.

The *Agreed* state denotes just that a *Responder* has agreed to a proposed agreement, issuing the respective acceptance message towards the *Initiator*. Once *Agreed*, the *Service Provider* must ensure that the service can in act be delivered as agreed – i.e. that all the resources required for service delivery are appropriately provisioned. The details of Service provisioning naturally depend on the nature of the Service to be delivered – and may be arbitrarily complex. In the case of atomic software services this would typically involve the provisioning of the necessary infrastructure and the deployment of the necessary software. For composed software services, in contrast, provisioning could entail the negotiation of second-level SLAs with 3rd party Service Providers. Once everything is in place, the ‘readiness’ for service delivery is signalled by switching the agreement to the *Provisioned* state – the deadline for which is the *start-time* of the agreement.

**Monitored (Effective & Renegotiated), Completed & Deprecated**

At the *start-time* specified in a *Provisioned* agreement, the agreement state switches to *Monitored* – indicating that the agreement is now in full effect and that service delivery is operative and requires monitoring. This state comes to a natural end, moving to the *Completed* state, at the specified *end-time* of the agreement. A signalling mechanism that keeps track of time, and is aware of the *start*- and *end-time* of the SLA, is responsible for producing the trigger events for switching from *Provisioned* to *Monitored* and then to *Completed* states.

The Monitored state comprises two sub-states: *Effective* (initial sub-state) & *Renegotiated*. A *Monitored* agreement is an *Effective* one – that is; it is an agreement which is in effect. If an agreement is renegotiated, then, at the point at which the new version is agreed, the old version switches from *Effective* to *Renegotiated*. At the start-time of the new version (i.e. the point at which the new version becomes *Effective*), the old version moves to the *Deprecated* state (and is preserved for reference only).

**Figure 6: Agreement states and re-negotiation**

If there is no renegotiation, then the agreement continues in the *Effective* state until it is either terminated, annulled or completed.
Annulled

The *Annulled* state indicates that, while in an *Observed* state, one of the parties has opted to annul the agreement (using appropriate messages as defined by the interaction protocol at hand). The other party can then either accept the proposed annulment (& terminate the agreement), or deny it (returning the agreement to the *Observed* state).

Terminated

The *Terminated* state indicates that an agreement has been terminated (come to end before its agreed end-time) – either by:

i) unilateral decision : in the case that one or more guarantees have been violated by one of the parties, the non-violating party may consider the agreement broken and subsequently terminated (included cases where re-negotiation of violated terms has been attempted but failed),

ii) a bilateral decision – i.e. annulment (while Observed), or withdrawal (while Pending) – in which one party proposes to terminate the agreement, and the other accepts this proposal.

2.4.2 Guarantee Term States

As with SLA states, the definition of *GuaranteeTerm* states is adopted, again more or less intact, from the WS-Agreement (WS-Agreement) specification (specifically; WS-Agreement Specification, March 14 2007). WS-Agreement defines just 3 GuaranteeTerm states:

i) *NotDetermined* : no activity regarding the *GuaranteeTerm* has happened yet (or is currently happening) that allows an evaluation,

ii) *Violated* : currently the obligation represented by the *GuaranteeTerm* is violated,

iii) *Fulfilled* : currently the obligation represented by the *GuaranteeTerm* is fulfilled.

The transitions between states are shown in Figure 7. The only difference with WS-Agreement is the explicit inclusion of a choice point – denoting just that conformance checking (i.e. verification) occurs at periodic intervals (as defined by a *MonitoringPolicy*).

![Guarantee Term States Diagram](image)

**Figure 7: Guarantee term states**


2.5 **SLA Hierarchies**

SLA@SOI aims to provide comprehensive SLA management support across all layers in an IT stack and between multiple stakeholders and service providers. The automated process of translating and correlating high-level requirements and policies of all kinds down to infrastructure level creates a set of related SLAs, which is termed an **SLA Hierarchy**. This knowledge about a set of related SLAs can be utilised to support SLA management, service management, can be a guide in the process of discovering the cause of SLA violations, and may help to provide more accurate analyses that are needed to make appropriate adjustment or renegotiation decisions at runtime.

It is notable that there exists more than one relationship between SLAs, for example:

- **Compositional (or containment) relationship** – a compositional relationship exists if it is valid to state that "SLA A (partial) contains SLA B". In the context of the AdHocDemonstrator use case scenario, the Intermediate SLA (see Figure 9) is a composite SLA of the PaymentService SLA and InventoryService SLA. Hence there exists a compositional relationship between the Intermediate SLA, the Payment Service SLA and the Inventory Service SLA.

- **Equivalence relationship** – refers to all SLAs (and sets of SLAs) that have a “translates to” relationship, for example "the business-level SLA A translates to software-level SLA B". We assume here that “translates to” implies a semantically identical result for the consumer. Equivalence relationships are formed during the negotiation and translation process.

- **Compliance relationship** – refers to all sets of SLAs that conform to the statement "SLA A will be violated if SLA B is violated".

- **Dependency relationship** – A weaker form of compliance relationship is the dependency relationship, which states that "SLA A could be violated if SLA B is violated." In the context of the AdHocDemonstrator, the PaymentService SLA depends on the CardValidation Service SLA. The Payment Service (as a composed service) is functionally dependent on the CardValidation Service and hence the completion time guarantee of the Payment Service SLA depends on the completion time guarantee provided by the CardValidation service SLA. If the CardValidation Service SLA is violated, it is likely (but not necessary) that the Payment Service SLA is also violated.

- **Association Relationship** - The SLA association relates sets of SLAs that are produced within the same translation and negotiation context and will be discussed below.

- **Other relationships** – There could exist other SLA relationships depending on the use case or the provider/customer point of view which are not directly addressed in SLA@SOI. The definition of the SLA Hierarchy as given below has been chosen to be as broad and flexible as possible in order to cover a wider range of relationships and to be efficient from the implementation point of view.

Note that these relationships are not mutually exclusive. For instance the containment relationship includes the compliance relationship since a violated guarantee term found in an SLA will also be violated in a higher-level composed
SLA that contains the very same guarantee term. Hence, those two SLAs have also a strong compliance relationship.

As an example of the different kinds of relationships described above, for the ORC and the adhoc demonstrator we can identify that

- the business-level SLA has a compliance relationship with the intermediate SLA
- the intermediate SLA is a composition of Payment service and Inventory service SLAs
- the Payment service SLA is dependent to Card Validation and Card Debit services, as already mentioned before
- and all software services have a compliance relationship with the Infrastructure SLA.

In order not to restrict the SLA@SOI framework usage, a broader and easy to build SLA relationship, the so called association relationship, has been considered to be the relationship that encompasses the overall SLA Hierarchy.

### 2.5.1 The Association Relationship

The association relationship has been constructed in a practical and feasible manner in order to satisfy aspects of the distributed and polymorphic nature of SLAs in a multiple stakeholder/multi-tier-architectural environment. The association relationship between two sets of SLAs $A$ and $B$ has been defined, whereby $A$ is the *domain or client* set of SLAs and $B$ is the *subcontractor* SLA set. The two sets $A$ and $B$ under discussion are disjoint as per definition. Note that the association relationship relates SLA sets to each other; a set can also consist of a single SLA.

The SLA association relates sets of SLAs that are produced within the same translation and negotiation context. In the translation and negotiation process SLAs are related to each other, but the nature of the relationship is not rigorously defined. Even if it is certain that these SLAs are related, the impact that they can have on each other it is not immediately obvious. The association relationship covers most aspects of the aforementioned relationships (i.e. compositional, equivalence, strong compliance relationships). The assumption is that the association relationship also covers the weak dependency relationship. Hence the generated set of related SLAs enables the correlation of monitoring events and can be used to determine the root cause of SLA violations in the domain SLA including if they were initiated by the subcontractor SLA set.

In order to model other relationships that are not covered by the association relationship, the provided SLA@SOI framework will be open and extensible enough to provide support for these circumstances and will enable domain specific solutions.

The following notations are used for expressing the association relationship between two sets of SLAs $A$ and $B$:

- $A$ depends on $B$

or

- \{ a_1, a_2 \} depends on \{ b \}

or, to express the association between single SLAs $a$ and $b$ (whereby SLA $a_1 \in A$ and SLA $b \in B$):
• \( a_1 \) depends on \( b \)

Furthermore, a non-textual representation can be used:

• \( A \rightarrow B \), read as “\( A \) depends on \( B \)”.

Also a graphical representation can be used as depicted in Figure 9. Note that Figure 8 shows the special case that a set of SLAs, namely \( a_1 \) and \( a_2 \), rely on subcontractor SLA \( b \).

![Figure 8 - Graphical representation of the association relationship](image)

According to the definition, the association relationship is transitive. The SLA Hierarchy, as described in the following section, is established on the association relationship.

### 2.5.2 The SLA Hierarchy Graph

The SLA Association Hierarchy Graph, or SLA Hierarchy for short, is a directed graph representing the association relationships between sets of SLAs within (or across) tiers in a multi-tier architecture as well as across multi-stakeholder domains. Due to the distributed nature of SLAs in multiple stakeholder environments, calculating and representing a complete SLA Hierarchy (e.g. as shown in Figure 9) is probably not feasible due to the limited visibility and accessibility of SLAs in other domains. However, in a single domain (e.g. the software provider domain) it is possible to construct a partial SLA Hierarchy.

A (partial) hierarchy may be utilised to determine:

- transitive SLA associations
- the transitive closure in order to detect dependency cycles

As mentioned above, the association relationship is transitive (in mathematical terms). Hence it is valid to state that if \( A \rightarrow B \) and \( B \rightarrow C \) follows \( A \rightarrow C \). While the association relationship only relates adjacent sets of SLAs, the SLA Hierarchy establishes SLA associations across the whole stack in a distributed multi-domain/multi-tier architecture. This enhances the scope of the SLA compliance monitoring and the reasoning about root causes of an SLA violation.
Calculating the transitive closure in a (partial) SLA Hierarchy enables the detection of cycles in an SLA Hierarchy graph. A cycle indicates that an error might have occurred in the translation process or an unwanted or unnecessary dependency between services has been established. If either is the case, a cycle in an SLA Hierarchy indicates that the reasoning about SLA violation based on the SLA Hierarchy graph becomes a lot more difficult or even impossible. However, it is assumed that association cycles are rare special cases and that it is possible to avoid cycles completely by using a proper SLA design.

Figure 9 shows the SLA Hierarchy for the Year 1 AdHocDemonstrator use case. Note that the service hierarchy established by the SOA translation process (i.e. the construction of a service hierarchy) is congruent with the SLA Hierarchy at the software layer level. The relationship between SLA hierarchies and other hierarchies in multi-tier /multi-stakeholder architectures is the subject of ongoing research. The SOA translation process is discussed in more detail in the deliverable D.A3a.
Figure 9 - AdHocDemonstrator Hierarchy
3 Architectural Elements

Deliverable D.A1a explains the overall architecture adopted for the first project year. Here we will discuss some WP A5-specific elements, more specifically:

- The negotiation module, with some further focus on the template registry and the negotiator
- The provisioning module
- The monitoring module, as regards A5-specific details
- The adjustment module

3.1 Management of Templates

The negotiation of an SLA typically begins with a SLA template, which describes what the service is about (a service description) and what kinds of guarantees are offered. The SLA@SOI architecture assumes that SLA templates are developed by service providers and advertised through a SLA Template Registry. The SLA Template Registry thus serves as an initial locus of SLA-enabled service information, such that service discovery can be accomplished by searching SLA Template Registries for suitable SLA templates - each denoting a provider’s offer of a service. In this way, we treat service advertisements as the basis for further negotiation of a service’s usage guarantees (i.e. the SLA negotiation).

The SLA Model (defined in Section 2.1) formalises this basic approach by conceiving SLA templates as an enriched kind of service description. Correspondingly, the SLA Template Registry is conceived as an extended kind of ‘service description registry’ (e.g. a traditional ‘service registry’ such as a UDDI repository). This approach has the advantage that, while primarily enabling ‘SLA aware’ search, it none-the-less remains consistent with more traditional views of ‘service discovery’ based on purely functional Service properties.

Once a suitable SLA template has been discovered, the negotiation of a SLA can begin. In brief; the Customer modifies any free variables (optional agreement terms) available in the template in order to define a specific SLA for the delivery of a specific service. Additionally, it is possible for the customer to add terms which do not exist in the template, or modify values for terms that in the template appear to be constant/static (in this case, it is up to the provider to evaluate the offer, or simply reject it). This SLA is then provisionally offered to the Provider. According to some predefined negotiation scheme, there may follow various counter-offers - which continue until the two parties either reach an agreement or give up.

This section describes the functional requirements of the SLA Template Registry in terms of UML Use Case scenarios. Section 3.1.1 describes the various actors involved in these use cases. Section 3.1.2 outlines the use of the registry for service-discovery, while Section 3.1.3 describes use cases related to the development & maintenance of registry content. The SLA Template Registry itself is described in Section 4.1.

3.1.1 Use Case Actors

This section describes the functional requirements of the SLA Template Registry in terms of UML Use Case scenarios. The various actors involved are depicted in Figure 10, and are conceived as follows:
i) **Accountable**: an actor that can be held accountable (in a legal sense) for their actions,

ii) **RegistryUser**: an actor able to access (i.e. query or browse) the contents of the SLA Template Registry. A RegistryUser need not be Accountable (e.g. it could be a software process).

iii) **RegistryManager**: a RegistryUser with the additional responsibility to maintain the contents of the SLA Template Registry. Since this is a responsible role (with business impact), we assume all RegistryManagers will be Accountable for their actions.

iv) **Template Designer**: an actor which is capable of generating (and/or editing) SLA Templates. These actors have access to the SLA Template Registry (as a RegistryUser), but are not responsible for managing its content – i.e. requests for modification of content must be submitted to the RegistryManager.

---

**Figure 10: SLA Template Registry Use Case Actors**

The following sections describe the activities of these actors in regards SLA Template creation & management, and template-based discovery.

### 3.1.2 Use Case: SLA Template Design & Management

This section describes the use cases relating to SLA Template Design & Management - as depicted in Figure 11 (described below).

---

**Figure 11: SLA Template Design & Management**

The RegistryManager is the entity (which may be an individual or organisation) responsible for adding/removing SLA Templates to/from the SLA Template Registry. As described in the previous section, each registered SLA Template denotes the actual offer of a service (by some provider). The PublishTemplate use
case denotes the act of adding a template to the registry – thus corresponding to the making of a service offer (the process is depicted in Figure 12). Likewise, the RevokeTemplate use case denotes the act of removing a template from the registry – corresponding to revoking a service offer (the process is depicted in Figure 13 – following from the SLA Model, SLA Templates are assumed to be uniquely identifiable).

![Diagram of SLA Template Publishing](image1)

**Figure 12 : Publishing an SLA Template**

![Diagram of SLA Template Revoking](image2)

**Figure 13 : Revoking an SLA Template**

The Create Template and Edit Template use cases concern the creation and subsequent modification (resp.) of SLA Templates by the TemplateDesigner. While not directly related to the SLA Template Registry, these use cases are included here in order to make explicit constraints on the modification of registry content.

In brief, it is conceivable that SLA Templates could be generated and/or updated automatically – e.g. based on automated analyses of infrastructure resources, or aggregations of existing offered templates, etc. This implies that the TemplateDesigner could be a mechanical process – and in particular, that the TemplateDesigner need not be legally Accountable. In contrast, the decision to offer a service - by advertising an SLA Template through the registry - or to modify or revoke an existing offer, is by definition a business decision – implying at least accountability. Thus, while TemplateDesigners may create & edit templates – it is the distinct responsibility of the accountable RegistryManager to decide which templates are advertised.

One implication of this definition is that it is not possible to edit the contents of a registered SLA Template in situ – since this would entail a change to an offered service. The modification of an existing registered template thus requires the old version to be revoked (removed from the registry), before the new version can be published – which process is depicted in Figure 14.
3.1.3 Use Case: Template-based Service Discovery

This section describes a single use case relating to template-based service discovery - as depicted in Figure 15 (described below).

The use case SLATemplateQuery encapsulates the activity of template-based service discovery – i.e. discovery is conceived just in terms of users querying the registry for its content. The results of such queries essentially comprise a list of those SLA Templates which satisfy the constraints expressed in the query (explained further in sub-section 4.1.2). As described in the previous section, our approach is to employ SLA Templates as representations of queries – hence the basic query process is as shown in Figure 16.

Alternatively, query results may also be given as a list of SLA Template identifiers (e.g. in order to reduce communications load) – which implies a subsequent
query to retrieve individual SLA Templates on the basis of their identifiers\(^4\) (as depicted in Figure 17).

**Figure 17 : SLA Template (Service) Discovery (2)**

### 3.2 Establishing SLAs

In this section we are discussing the details of the process to establish SLAs, more specifically with regards to negotiating prospective agreements. We start the section by discussing various negotiation strategies – however, it should be underlined that these strategies are not implemented in Year 1, rather they are researched in anticipation of further project development. Then, we provide adhoc-demonstrator related negotiation details.

#### 3.2.1 Negotiation strategies

Negotiation is the process by which a group of agents come to a mutually acceptable agreement on a contract the required/provided service should satisfy. It is profoundly influenced by the way the SLA model is represented. In this Section we will describe how the SLA model impacts on the negotiation strategies and, in particular, on the main dimensions that characterise the negotiation process. As described in [6], these dimensions are: the negotiation protocol, the negotiation objectives and the agents’ decision model.

**Negotiation protocol**

It includes the set of rules that govern the agents’ interaction:

1. The role of participants and their multiplicity. Varying the number of participants for each role we foresee the following negotiation scenarios:
   - **1 (customer) – 1 (provider)**: it is a bilateral negotiation, in which the customer bargains with a provider for the definition of the SLA of the specific service. For the first year we are only providing customer initiated scenarios, while we will offer provider initiated scenarios in the next years. In this last case the customer has to encapsulate the desired service properties into an unsigned SLA and the provider has to

---

\(^4\) conceptually, the identifier can be treated as a shorthand representation of the SLA Template it identifies.
propose a valid contract to the issuing customer according to the stated requirements.

• **1 (customer) – N (provider):** it foresees a customer that negotiates one or more SLAs with a set of providers. This kind of negotiation can be either initiated by the customer who selects among a set of providers, or it can be initiated by providers who try to satisfy the service demand issued by the customer. This scenario is compatible with our SLA model, which does not constrain the number of contracts a customer can sign with different providers of the same abstract service. If the customer needs to select only one provider, for example opening an RFQ (Request for Quotes), providers will use a competitive strategy. Another possible scenario can be that service providers cooperate to share service provisioning (for example, splitting the service availability interval, or using other services as backups in situations of heavy load).

• **M (customers) – 1 (provider):** a provider bargains the service provisioning with several customers. The probability for a customer to get an advantageous contract depends on the provider resource availability. Resource availability depends, in turn, on the contracts that the provider has already signed with existing users (how many and on which negotiation parameters). For these reasons it makes sense for the provider to start an auction with the customer for selling the available SLAs.

• **M (customers) – N (providers):** It merges the two multi-party scenarios described above. It is worth to notice that customers always interact in a competitive way to get the best SLAs, while service providers can cooperate or compete. In the first case service providers constitute a partnership to the service provisioning in order to reach an integrated SLA. In the second case providers compete to get the best contract offers proposed by providers.

2. The negotiation policy model. It can be bargaining, RFQ and auction. In general the selection of a specific policy is strictly related to the multiplicity of the SLA contractors. Bargaining is a process of discussion and give-and-take among the negotiation participants until an agreement is reached (in which case the negotiation is successful) or not. It can be relatively cooperative, i.e., when some or all of the participants seek a solution that is mutually beneficial (commonly called win-win or cooperative bargaining), or it can be competitive (commonly called win-lose or adversarial) bargaining, when one side seeks to prevail over the others. In this year we focused only on this kind of negotiation strategy, while in the next years we will deal with RFQ and auctions. RFQ is activated when a customer sets up a competition among providers to choose the best offer for a good. It is tailored for 1-n scenarios. While auctions take place when potential buyers compete for a number of resources or fixed resources; they are tailored for M-1 scenarios. Finally for M-N scenarios it will be necessary to develop “hybrid” negotiation mechanisms combining RFQ and auctions.

3. The way communication must happen among negotiation parties. It includes the following aspects:
   a. The negotiation states, the events that cause the negotiation states to change and the valid actions of the participants in particular states
   b. The exit states
   c. The visibility of the exchanged messages.

The aspects mentioned above (excluding policy-specific transitions) have been already explained in Section 2.3, in which we defined the possible
states in which an SLA can be, from its proposal up to its enactment and termination. The only missing thing is to establish what happens if no agreement is reached. Looking at the SLA states, we can have this situation when an offer is always in a “Pending” state. A solution can be to define a maximum amount of times an SLA offer can be withdrawn, and if that amount is exceeded the SLA contract is considered as Rejected. Other situations may be faced when an agreement cannot be achieved. A solution able to improve the convergence to an agreement can be to fix a BATNA (Best Agreement to Negotiated Agreement) [7] that is an insurance any outcome should be compared to. It protects the parties from accepting unfavorable agreements and from rejecting favorable ones.

Another possible way to accelerate the convergence to an agreement can be improving communication between the negotiation participants, providing feedbacks to received offers. In general a feedback is implicitly encapsulated in the counter-offer, but it can also be a finer grained critique, which indicates acceptance/rejection of particular parts of the proposal and/or a degree of satisfaction of those parts. Hence a critique is more explicit than a counter-offer and agents do not have to guess the constraints and preferences of the opponents.

**Negotiation objectives**

The negotiation objectives represent the set of parameters over which an agreement must be reached. The operations allowed on these parameters are decided by the protocol, that is, whether the content of the offer can be changed (e.g., a counter-offer made) or it has to be accepted/rejected as it is. The parameters that can be negotiated are related to the SLA model: We want to negotiate the SLOs (Service Level Objectives) contained in the Guarantee Terms. They carry on the desired QoS properties the contractor expects from the application. We already described a first (common) set of negotiable QoS properties for an SLO, in Section 2.2.

Moreover the proposed SLA model allows agents to negotiate other kinds of properties:

- **Business Value.** It is possible to negotiate the cost parameter associated to a business value. It indicates the imposed rewards/penalties in case obligations are fulfilled or not. It should be possible to negotiate different kinds of costs. For example, a reduction in the service price to customer, or an upgrade in service level (moving a customer to an higher QoS level), or by means of credits issued to customer for future use in the same or different services by the same service provider. A penalty applied to customer may take several forms. It can be reflected in an increase in the service price or it may entail a degradation of the service conditions.

- **The monitoring policy,** describing the conditions under which the monitoring of the SLA is performed. For example it can be possible to negotiate the way SLA violation reporting is performed by the verifier: as soon as a violation occurs, periodically, or when the probability of violating a guarantee exceeds a specific value. Contractors may also negotiate the control actions that must be taken in order to recover from a monitored violation. Such control actions may be automated or may refer to operational processes alongside properties of these processes.

It is worth to notice that not all negotiable parameters appearing in an unsigned SLA (i.e. a SLA template) can be used during the negotiation and similarly, additional attributes, other than those, could be part of the negotiation
objectives. In this last case we assume that the new negotiated parameters can be of one of the categories described above (SLO, monitoring policy or business value) and they have to be coherent with the parameters advertised in the unsigned SLA. We may also think that new parameters can be added or removed when a contract is renegotiated.

The relationship between the final established SLA (signed) and the original one (unsigned SLA) could be of different types, such as:

- The unsigned SLA may refer to the average variability of the service characteristics, whereas a signed SLA is a commitment for specific values in those ranges. In this case the negotiation will generate a restriction on the values represented into the unsigned SLA.
- The unsigned SLA contains a sample of the possible service provisioning and a SLA refers to one (or some) of them. We encounter this situation when we differentiate QoS parameters interval or their price according to the time of the day. Another scenario may take place when the provider offers fixed service levels (“gold”, “silver” and “bronze”) and the customer has to select among one of them.
- The signed SLA may include new features that are not advertised to all users and may be not consistent with those published in the unsigned SLA, e.g. the service provider may decide to guarantee a certain response time to a consumer that he/she does not want to publish as a normal characteristic.

Other considerations are related to how a SLA resulting from a cooperative approach of various providers may be represented at the consumer’s side. The customer may not be aware of the distribution of the service among different providers. Another solution can be to represent explicitly the distinct SLAs offered by each provider; in this way, the customer will be aware of the distribution of responsibility. A possible application of this scenario can happen when the same service is negotiated with different providers, and at runtime one of them will be selected according to cost, availability or other criteria. Unifying the first two representations we can define parametric SLAs. These SLAs should encapsulate the function that will permit to select among a particular service provider, like preconditions (enabling conditions) on the guarantee terms related to one of the providers. For example, if a customer can choose among more than one mobility service provider, the best one is selected depending on the geographical position of the consumer. Finally, these enabling conditions can be helpful to distinguish guarantee terms that need to be applied depending on the actual consumer who is using the service.

**Agents decision model**

It includes the decision-making process the participants adopt to decide if they will accept or reject a contract proposal. In general this is domain-specific, and as such cannot be addressed in a universal way. The project is interested in creating architectures and primitives which allow easy introduction of such domain-specific, complex decision models.

The problem contains other sub-problems:

- Define the acceptable ranges/constraints for the negotiation objectives: there are parameters, like business value that can be negotiated according to stated ranges, while there are other negotiation objectives, like monitoring policy that can be associated with different choices having different degrees of acceptance.
- How to generate SLA offers/requests. A first assumption we can do is that they are generated from ranges of acceptable values previously defined by
the negotiation participants. It can be also possible to adopt a function to express how that offer is near to the desired one and an offer can be accepted if it is “sufficiently” near.

- Which strategy it is necessary to adopt to pursue a goal: the algorithm to decide the moves, how to react to the received offer, e.g. accept/reject based on improvements with respect to the previous answers; what objective to attempt, e.g. the one that is more convenient at that stage; how to compute a new offer/proposal, e.g. following an analytic approach (according to a pre-defined function), or using a learning approach (accounting for the reaction of the other participants on the previous bids).

The decision model we adopt may result from a negotiation planning activity. For each participant, this means:

- Assessing its own resources in order to define the negotiation alternatives and boundaries (what is possible to offer and to what extent);
- Prioritize the objectives and evaluate possible trade-offs among them;
- Recognize whether there are common interests with the other participants (this will determine the adversarial or problem-solving approach as explained below);
- Decide what information to share with the others (professional negotiators often exchange information about targets or initial proposals before negotiations begin);
- Have a profile of the other party (reputation, willingness to collaborate, organizational position on the market); understand his/her alternatives. If the other negotiator has strong alternatives, he or she will probably not be flexible during negotiation, will set high objectives and will insist on those.

This negotiation planning activity would have to rely on canonical forms representing the terms under negotiation, so that the parties involved can make informed decisions with less ambiguity as to the content of the terms.

3.2.2 Architectural details For Year 1

Currently (Year 1), the project is only supporting customer-initiated, 1-1 negotiations, without re-negotiation capabilities. This is due to the fact that:

a) The ad-hoc demonstrator use case did not include a requirement for provider-initiated negotiations,
b) The complexity of multi-party negotiations is not trivial and requires further research, and
c) Multi-party negotiation is needed for dynamic discovery and late binding, which is, at this time, work in progress.

The current architecture was built around the ad-hoc demonstrator and taking into account those requirements only. As the industrial use case requirements were released close to the end of Y1, it is expected that an architectural update will be required to address them.

Currently, the Negotiation Module is responsible for the complete process of negotiating and planning SLAs. The module includes the following components, as also shown in Figure 18:

- **Negotiator**
  It implements the createAgreement interface, which is invoked by the eContracting module, but also by itself in a recursion which establishes the SLA hierarchy. The Negotiator is that component which performs negotiation between services. It should be noted that in the ad-hoc demonstrator the negotiation is happening in a vacuum, and therefore it might be more accurate to refer to “emulation of service negotiation”.
However, from the point of view of agreement offer structure, there is no difference from using WS-Agreement with a SOAP/HTTP(S) transport (in other words, the data offered and received is the same as it would be for WS-Agreement based web service interactions).

As discussed earlier, the only negotiation scheme supported at this moment by the Negotiator, is customer initialised, 1-1 negotiation. Future releases will include multi-party negotiations, which might be initiated by any of the parties.

**Figure 18: Negotiation module internal architecture**

- **Translation**
  Implements a `translateOffer` interface, which is invoked by the Negotiator for the structural translation of received offers. The input to this operation is the result set of the Prediction component invocation as discussed below, or a subset of it, and the appropriate templates as they are retrieved from the SLA Template Registry. The Translation component combines the templates with the values from prediction, thus forming new SLA offers which the Negotiator will then use to establish lower-layer SLAs. The infrastructure requirements is a good example of that; the Prediction component is producing those requirements in a simple custom format, and the Translation component is including these requirements in the infrastructure templates that exist in the template registry, to produce an Infrastructure SLA offer.

- **Prediction**
  Implements the `doPrediction` interface, which is invoked by the Negotiator. The prediction component applies forecasting techniques to provide reasonable assumptions regarding either the decomposition of SLOs in the received offers, or the infrastructure required for well-specified software deployments, taking into account the guarantees in effect. Prediction is described in more detail in deliverable D.A6a.

- **Template Registry**
  The Template Registry is the source of templates for the service offerings. It exposes the `getTemplates` interface, which accepts a template as a
query, getTemplates is invoked either from external entities (e.g. eContracting module), or internally by the Negotiator (during recursion for building the SLA hierarchy) and the Prediction component (to find out infrastructure options).

The Template Registry is discussed in section 3.1.

- **Design-time Repository**
  The Design-time Repository offers storage for service design-time models. These models are the basis for prediction facilities, since they are used to describe service structure and behaviour, and therefore model future performance.

The Design-time Repository is described in more detail in deliverable D.A6a.

![Recursive nature of negotiator invocations](image)

**Figure 19: Recursive nature of negotiator invocations**

The negotiator is currently realised by the adapted WSAG4J library (more details in section 4.2). Within the implementation of the createAgreement operation, the doPrediction operation of the Prediction component and the translateOffer operation of the Translation component, are invoked as necessary. More specifically, createAgreement is initially invoked by the eContracting module and this operation receives as input a WS-Agreement-formatted offer. This invocation is followed by the (Negotiation module-internal) invocation of doPrediction, providing necessary input for this prediction facility. The Prediction component is making use of design-time models, in order to analyse requirements and estimate relevant SLOs for the services that compose the PaymentService, or to come up with infrastructure requirements for those services. After receiving those estimations, the Negotiator is passing them on to the Translation component, which creates the new offers in WS-Agreement format. This sequence of invocations is performing the SLA translation numerically and structurally, therefore building the SLA hierarchy. It should be noted here that the translation requirements are hard-coded for Year 1. For example, the fact that achieving a completion time target for the composite PaymentService is possible through achieving completion time targets for the services which compose it, is knowledge that has been hardcoded in our demonstrator framework. The project is
continuously researching the topic of proper generalisation of this problem, how to formalise and how to implement it in expectation of Year 2 results.

One thing that should be noted here, is that the ORC limitations (due to the single-provider scenario) pose certain limitations also to this negotiation and SLA hierarchy generation process (specifically for the ad-hoc demonstrator). Since there is a single provider and therefore a single set of resources to be shared by all services, re-invoking Prediction on all createAgreement() calls (for single services, e.g. for the card validation service only) would not return usable results. Therefore, we are only using it on the topmost level of software services. However, in a more generic scenario with multiple providers, prediction would be used for each one of them in each createAgreement() invocation.

The process discussed above can be modelled in a UML sequence diagram as follows:

![Sequence Diagram](image)

**Figure 20: Sequence diagram for negotiation process**

### 3.3 Provisioning

The Provisioning module of the SLA Management Framework architecture is responsible for the software service provisioning and configuration activities. Primarily, Provisioning module’s services are utilized by the Negotiation module in the following manner:

- During the negotiation process, Negotiation module interacts with the provisioning module to acquire information about the virtual appliances required to deploy and enable the services. Additionally, the lead time required to bootstrap the appliances is also provided to the Negotiation module. Negotiation module uses this information to interact with the Infrastructure Management module for infrastructure related SLA negotiation and agreements. Detailed description of this is given in one of the following sections.
- After an SLA has been successfully established, Negotiation module passes on the SLA which is stored within one of the components of the
Provisioning module, SLA Registry. If there is a hierarchy of SLAs produced as a result of a closed loop within a single service provider (as in the ad-hoc demonstrator scenario), a complete SLA hierarchy is provided for storage. Moreover, Provisioning module uses the information in SLA(s) to set timers for activating the service provisioning and configuration related activities once the virtualized infrastructure has been successfully provisioned.

Elaborate description of the Provisioning module’s interaction with other architectural elements of the SLA management framework is given in the upcoming sections.

### 3.3.1 Provisioning Module Architecture

The Provisioning module’s architecture is shown in Figure 21. The interfaces colour coded with blue represent the module public interfaces. These interfaces make the functionality of Provisioning module available to other modules in the framework. Detailed description of these public interfaces is given in the next section. This section focuses on the description of architectural elements of the Provisioning module.

![Figure 21: Provisioning module architecture](image)

Provisioning module’s architecture is composed of five main components, which are explained in the following paragraphs.

**Service Activation Engine**

The service level agreements established between the service provider and the customer could be meant to be activated at some point in the future rather than immediate provisioning. To provide for these scenarios, service provisioning activation schedules need to be maintained and triggered in due time. Service Activation Engine component of the Provisioning module serves this purpose. This component is used by the Negotiation module to register SLAs which need to be
activated in the future. The Service Activation Engine is comprised of three components which are described below:

**Activation Manager**

Activation Manager is the main component within the Service Activation Engine. It serves as the supervisory entity of the engine acting as the front-end of the engine. Activation Manager exposed the public operations (createSchedule & getSchedules) of the Provisioning module interface. It receives the requests and then dispatches them to the other components of the service activation engine. The main operations provided by Activation Manager for other components within the Service activation Engine are described below:

- **createSchedule:** This operation generates the activation schedule for the SLA / Service provisioning. It is invoked by the Negotiation module and it updates the agreement document passed as argument with information about the appliances required for SLA provisioning as well as the lead time required for the service bootstrapping.
- **finalizeSchedule:** This operation is complementary to the createSchedule operation. During the invocation of createSchedule operation, a provisional schedule is created and stored because the SLA has not been established. After establishment of the SLA, during storeSLA operation, this operation is called to finalize and activate the schedule corresponding to the SLA.
- **getSchedule:** This operation takes a string array representing the SLA ID array as argument and returns the corresponding schedules.

**Activation timer**

Activation Timer component maintains the "alarm clock" logic of the service activation engine. Activation Manager creates and registers schedules for the SLAs / Services which are to be provisioned in the future. Activation timer triggers the provisioning / configuration activities in due time according to the schedules registered by the activation manager. The main operations of the Activation Timer component are described below:

- **registerSchedule:** This operation registers a schedule with the activation timer. This registers an alarm with the activation timer. When the time comes, the service configuration for the corresponding SLA will be triggered.
- **unregisterSchedule:** This operation is complementary to the registerSchedule. It cancels a previously registered schedule. This operation will be used in cases, where an established SLA has not been provisioned yet and is terminated for whatever reason.

**Activation Schedule Repository**

Activation schedule repository component of the service activation engine serves as the persistent data store for activation schedules. Activation manager and activation timer rely on this component for storing the pending or registered activation schedules.

- **storeSchedule:** This operation stores the schedule into a persistent data store.
• **updateSchedule:** As mentioned previously, a schedule is stored provisionally during the `createSchedule` operation invocation. This operation can be used to update the schedule information e.g. changing its status from provisional to finalised.

• **fetchSchedule:** This operation can be used fetch information about a schedule which has been stored in the persistent data store. This operation is used under the hood by the `getSchedule` operation of the Activation Manager.

**SLA Registry**

This component is responsible for storing and maintaining established SLAs. The SLA Registry component is comprised of the following sub-components.

**SLA Registry Manager**

SLA Registry Manager component serves as the front end of the SLA Registry. SLA Registry Manager receives SLAs for storing them and allows retrieval of a SLA document given appropriate identification. It interacts with the data store to carry out its functionality. The main operations exposed by the SLA Registry Manager are described below:

• **storeSLA:** This operation will store a single SLA or a complete SLA hierarchy into the SLA registry. Based on the activation timing defined in the SLAs, the provisioning module will trigger the provisioning operations accordingly.

• **getSLA:** This operation will retrieve a SLA document.

• **getHierarchy:** This operation returns a “SLA Hierarchy” data type, based on the input argument which is the ID of a single SLA. The returned value is a sequence of “customer SLA” IDs, and a sequence of “subcontractor SLA” IDs, as also explained in section .

**SLA Registry Data Store**

The SLA Registry data store provides persistent data store capabilities for storing established SLAs. For the time being, this has been implemented as an XML database.

**Service Configuration Manager**

This component is responsible for the software / service configuration. Once the infrastructure provisioning has been completed successfully, this component is invoked and it incorporates the software / service specific configurations according to the SLAs established and customer requirements. The functionality of this component is service specific; hence, a plug-in based extensible architecture should be adopted to implement it. This component is responsible for bootstrapping the services, execution software e.g. application servers, web servers, databases, workflow engines etc. according to the order specific to the service environment. For example, database shall be started before the application / web servers. This coordinated bootstrapping is critical for correct functioning of the services. In year 1, the service configuration manager wouldn't implement the bootstrap logic but it would certainly be within scope from Year 2 and beyond. The Configuration Manager offers one operation:
• **startConfiguration**: This operation will be triggered by the Activation Timer when the alarm for an activation schedule goes off. It will invoke service specific bootstrapping and configuration activities.

**Event Manager**

This component is responsible for receiving the events and dispatching to the destined components within the module.

**Software Landscape**

This component is responsible for storing the software / service configurations and other service specific information. Software Landscape component will be used during the SLA planning process to access information about the appliances required to be deployed for SLA provisioning. Additionally, Software Landscape component will maintain lead times which refer to the time required for the boot up of the services. This information will be required by the Negotiation module to carry out infrastructure SLA negotiation with the infrastructure provider. The main operations provided by the Software Landscape component are described below:

- **getDeploymentInfo**: This operation is used to retrieve appliance related information for a service. It will be used during the createSchedule operation. The appliance information refers to the appliance identifiers and the lead time for the service bootstrapping and configuration. This is an internally visible operation meant to be used by the Negotiation module.
- **setServiceInfo**: This operation is an externally visible operation meant to populate information about the services and the respective service models containing information about the software elements required by the service, virtual appliance related information etc. The Service model refers to the landscape model designed in task A3.3.
- **getServiceInfo**: This operation can be used to query the software landscape to retrieve service models for services. This operation is the counterpart of the previous operation for storing service model information.
- **getApplianceDetails**: This operation works similar to the getDeploymentInfo operation. This operation takes an appliance ID and returns the appliance information. The purpose of this operation is to retrieve list of services which are packaged in an appliance. The prediction component will be the primary consumer of this operation.

### 3.3.2 Public Interfaces

This section focuses on two of the main operations of the Provisioning module. These operations play an important role in the SLA@SOI adhoc demonstrator. This section elaborates on the flow of execution and the activities performed within the course of execution of these operations. UML sequence diagrams are provided for comprehension. The two operations that are subject of this section are createSchedule and storeSLA.

**createSchedule**

The createSchedule operation is one of the main operations in the provisioning module's public interface, and is invoked by the Negotiation module. This section explains the detailed design of the createSchedule operation. The operation is
invoked with the agreementDocument as its argument. This agreementDocument refers to the agreement currently being negotiated. createSchedule operation populates the agreements document with extra information regarding the appliances required to provision the SLA and the lead time required for the bootup of the service middleware and services themselves. The detailed flow of information and execution is given below:

1. createSchedule operation invoked by the Negotiation module. The argument (as described above) is the agreement document currently under negotiation. ActivationManager component (part of Service Activation Engine) receives this operation.
2. The ActivationManager component extracts the list of service identifiers which are currently the subject of negotiation.
3. The ActivationManager invokes the getApplianceInfo operation on the Software Landscape component. The list of service identifiers is passed onto the getApplianceInfo operation.
4. The SoftwareLandscape component iterates through the list of service identifiers and extracts appliance-packaging related information from the repository for each service. As mentioned previously, this information comprises of the identifiers that can be sent to the infrastructure module as part of the infrastructure SLA negotiation process. These identifiers were received when the appliance images were uploaded to the infrastructure provider's appliance image repository.
5. The SoftwareLandscape component finds the common (unique) appliance identifiers from the list compiled in the previous step. Services can be deployed onto the same execution environment and hence share the appliance image. This intersection yields the list of appliances required to provision all the services listed in the SLA agreement document.
6. Appliance information is returned back to the ActivationEngine component. This information comprises of the list of appliance identifiers and the lead time for bootstrapping the services and execution environment e.g. application servers, databases, workflow engines and services.
7. ActivationEngine invokes the storeSchedule operation on the ActivationScheduleRepository component. This schedule is stored provisionally and is not active yet because the SLA is still not established at this point.
8. ActivationEngine component updates the agreementDocument with required information
9. ActivationEngine returns this updated agreementDocument to negotiation module.

A UML sequence diagram for the createSchedule operation is shown in Figure 22.
**storeSLA**

The `storeSLA` operation is invoked by the negotiation module once the SLA has been established. The `storeSLA` operation is invoked on the SLARegistryManager component. The arguments passed onto the `storeSLA` operation are the SLA document. Apart from saving the SLA document into a persistent registry, a number of other steps are performed. The detailed execution flow of the `storeSLA` operation is given below.

1. `storeSLA` operation invoked on the SLARegistryManager component.
2. SLARegistryManager extracts the SLA identifier.
3. SLARegistryManager invokes the `finalizeSchedule` operation on the ActivationManager component. A provisional activation schedule of this SLA was stored when the `createSchedule` operation was invoked. Since the SLA has been established at this stage, the activation schedule must be finalized and activated.
4. ActivationManager component receives the `finalizeSchedule` operation and invokes `setStatus` operation on the ActivationScheduleRepository. The activation status is maintained as a flag/field in the ActivationScheduleRepository.
5. After setting the status of activation schedule, ActivationManager invokes `registerSchedule` operation on the ActivationTimer component. ActivationTimer maintains a timer which triggers the activation/configuration operation in due time.
6. ActivationTimer component extracts the schedule information (appliance information and lead time) and registers an alarm in its internal clock/timer.
7. The control returns back to the SLARegistryManager component. At this point, the schedule has been finalized and activation alarm configured. SLARegistryManager invokes `storeSLA` operation on the SLARegistry which provides a persistent data store for storing established SLAs.

---

**Figure 22: UML sequence diagram for the createSchedule operation**

This UML sequence diagram illustrates the flow of the `createSchedule` operation.
8. storeSLA operation finishes and control is returned back to the negotiation module. The SLA provisioning will be triggered automatically in due time and services should be available for users to consume.

![UML sequence diagram for the storeSLA operation](image)

**Figure 23: UML sequence diagram for the storeSLA operation**

The UML sequence diagram for the storeSLA operation is shown in Figure 23.

### 3.4 Monitoring

This section discusses the design of the architecture for SLA monitoring at the Software service layer. Moreover, we also discuss how the monitoring at the Infrastructure service layer can be integrated with Software service layer monitoring through a publish-subscribe infrastructure.

The design of the SLA monitoring architecture relies on the following assumptions:

- SLA Monitoring is **event-based**;
- SLA Monitoring is decoupled from other modules of the SLA Framework;
- SLA Monitoring allows the flexible plug-in of new monitors (i.e. the entities which receive and process the events) within the reference architecture.

**Event-based monitoring.** Event-based, 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 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). From the point of view of the instrumentation of the service execution environment, the only instrumentation
required by non-intrusive monitoring is the one required to capture events within the service execution environment and made them available to the monitoring framework. With intrusive monitoring, such an instrumentation includes also monitoring features, i.e. mechanisms that allow the checking of the satisfaction of the properties for which monitoring is required. Although it has several benefits, intrusive monitoring implies tighter coupling of the monitoring features and the service execution environment.

Event-based monitoring fits 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 cannot 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.

As previously introduced, such a decoupling is achieved by separating the SLA monitoring framework from the instrumentation required in the service execution environment for capturing the required events and made them available to the SLA monitoring framework. The instrumentation of the service execution environment, in particular, is implemented by SOAP message captors within the Axis implementation for atomic services and by message captors deployed within the BPEL engine for composed services. BPEL message captors are part of a larger scope instrumentation, which includes methods for re-provisioning and re-configuration that are exploited by the adjustment features of the SLA@SOI Management Framework.

**Monitoring is decoupled from other modules of the SLA Framework.** As previously introduced while describing the principles of event-based monitoring, our objective is to achieve the decoupling between Monitoring and the other modules composing the SLA Management Framework. This is achieved by introducing a Publish-Subscribe infrastructure (Event Bus) which is used by the different modules to exchange messages, i.e. events relevant for monitoring. The Event Bus is also used by the instrumentation of the service system which captures events related to the ORC services.

**Flexible internal architecture for plug-in of new monitors.** Eventually, the architecture we developed for SLA Monitoring should be flexible enough to allow different monitors to be plugged-in (in case new monitoring features are required). In the literature, in fact, several monitors for service systems are provided, which focus on the monitoring of different aspects, such as functional properties, performance related properties, response time, or generic pre- and post-conditions on service execution [8][9][10][11]. In the architecture for Monitoring described in this section, we show how two different monitors, i.e. the one developed at CITY [8] and at the FBK [11], can be plugged-in into the architecture by implementing the interface described in the overall architecture of the monitoring module.

In this section, we first provide an overview of the SLA Monitoring architecture, focusing on the interconnection of monitoring with other functionalities of the SLA Management Framework. Then, we provide detail on the components in the Monitoring architecture.

**The Overall Architecture of the Monitoring module**

The overall black-box architecture of the Monitoring module within the SLA Management Framework is shown in Figure 24.
Before going into detail in the description of the provided and required interfaces of the Monitoring module and Event Bus, we briefly discuss the type of properties for which monitoring is required in the Year 1 demonstrator and the events that may be exchanged by the modules reported in Figure 24 for monitoring purposes. Within the Year 1 SLA@SOI demonstrator, the properties that have to be monitored are derived from the Guarantee Terms of the SLA established for the provisioning of the ORC services. Such Guarantee Terms may contain the following:

- A qualifying Condition on customer’s requests arrival rate;
- A Service Level Objective (SLO) on the average completion time of operation calls.

As a sample, considering the operation operation1 exposed by the service serviceA, a typical Guarantee Term may be expressed as “when the arrival rate of requests for serviceA is less than X req/s, then the average completion time of calls to operation1 must be less than Y ms”.

In order to monitor the aforementioned properties, the Monitoring module requires information, in the form of events, on the service operation calls and responses, and, in particular, on their timestamps. Hence, for the Year 1 SLA@SOI ad-hoc demonstrator, we consider two different classes of events:

- **Interaction Event Type**: They signify the actual time-stamped service operation calls and responses. In other words, this kind of events represent the elementary information which is used for detecting the violations of Guarantee Terms;
- **Monitoring Result Event Type**: They signify the violation of a Guarantee Term belonging to a negotiated and provisioned SLA. This kind of events is produced by the Monitoring module and made available to other modules of the SLA Management Framework.

More details on (i) the structure of the events and on their XML representation and (ii) on the list of actual events considered in the Year 1 demonstrator are given in Deliverable A3a.

The SLA Monitoring module represented in Figure 24 performs monitoring of SLAs at the Software Service Layer. In particular, the Monitoring module receives (i) the SLAs that have been negotiated and provisioned from the Provisioning module and (ii) events generated by the instrumentation of the services of the Open reference Case (Interaction Events). Interaction events are used to detect...
whether the SLAs submitted by Provisioning have been violated. When a violation is detected, a Monitoring Result Event is generated and pushed by the Monitoring module on the Event Bus. Note that the Infrastructure module in the SLA Management framework is able to perform internally monitoring at the Infrastructure service layer. Therefore, when a SLA at the infrastructure layer is violated, the Infrastructure module pushes a Monitoring result event on the Event Bus. External modules, such as the client application Ad-Hoc demonstrator and the Adjustment module of the SLA Management framework subscribe to Monitoring Result Events, in order to be notified of SLA violations. Thus, the Event Bus coordinates SLA monitoring by allowing monitoring-related event producers, i.e. Monitoring, Infrastructure and the ORC instrumentation, to disseminate events and event listeners, i.e. Ad-Hoc demonstrator and Adjustment modules, to receive the events to which they are interested (Monitoring Result Events).

In the following, we discuss in detail the interfaces implemented by the Monitoring module and the Event Bus.

**Interfaces provided and required by the Monitoring Module.**

**Monitoring module provided interfaces**

- **startMonitoring**: Allows Provisioning to start the monitoring of a negotiated SLA (with services EPRs). The SLA embeds information on when it becomes valid (i.e. when monitoring should start).
- **notifyEvent**: Explanation Allows the Event Bus to notify events relevant for monitoring into the Monitor Engine (Interaction Events). The Monitor Engine pushes rule violations on the Event Bus, the AdHoc Demonstrator consumes such events.

**Monitoring module consumed interfaces**

- **subscribeEventType** (see Event Bus component interfaces and relationships)
- **registerEventType** (see Event Bus component interfaces and relationships)
- **pushEvent** (see Event Bus component interfaces and relationships)

**Interfaces provided and required by the Event Bus**

**Event Bus provided interfaces**

- **subscribeEventType**: Used by external clients to subscribe to a list of event types relevant for monitoring available on the bus. AdHocDemonstrator and Adjustment will subscribe for receiving Monitoring Result event types. The Generic Monitor Engine will subscribe to events required for monitoring (e.g. Interaction Event Type).
- **registerEventType**: Used by external clients to advertise a list of event types relevant for monitoring. ORC will register Interaction Event types (collected by the service instrumentation). Monitor Engine and Infrastructure will register Monitoring Result Event types (violations of Guarantee Terms).
- **pushEvent**: Used by external clients to push an Event on the bus. Generic Monitor Engine and Infrastructure push Monitoring Result Events (SLA violations). ORC pushes Interaction events (from ORC instrumentation).

**Event Bus consumed interfaces**
**notifyEvent**: Used to notify an event to subscribers. Generic Monitor Engine receives notifications of Interaction Events. Adjustment and AdHoc Demonstrator receive notifications of Monitoring Result Events.

Figure 25 shows the internal architecture of the SLA Monitoring. According to the principle of developing an architecture in which different service monitoring engines may be plugged in, Figure 25 depicts only a Generic Monitoring Engine. In Deliverable A3a, while discussing service monitoring, we show how the implementation of the SLA@SOI ad-hoc demonstrator involves two of such generic service monitoring engines, i.e. the one developed at City University London and the one proposed by FBK. More details on the integration of the core monitors is given later in Section 4.3 while discussing implementation issues.

**Figure 25 – Grey-box architecture of the SLA Monitoring framework**

Generally, a Generic Monitoring Engine must be able to detect violations of an SLA during service provisioning and to communicate such violation to the Event Bus, i.e. produce an event of type Monitoring Result Event Type. For each Generic monitoring engine, the SLA monitoring framework involves a Rule Generator module and an optional Monitor User Interface:

- The Rule Generator module is able to derive, from a submitted SLA, the properties to be monitored expressed in the language required by the Generic Monitoring Engine. In the case of CITY monitor, for instance, this involves translating the Guarantee Terms in the SLA into suitable rules expressed in Event Calculus;
- The Monitor User Interface is a graphical user interface that can be used to check the results of the monitoring performed by the Generic Monitoring Framework. Such User Interface is not mandatory within the SLA monitoring framework.

Besides the Monitoring module, the SLA monitoring framework includes the Event Bus. The Event Bus implements a publish-subscribe infrastructure which allows the different modules of the SLA management framework to exchange events. More specifically, the role of each module of the SLA management framework interacting with Event Bus and the type of events published or subscribed by each module is shown in Table 1.

<table>
<thead>
<tr>
<th>SLA Management module</th>
<th>Type of Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interaction Events</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Subscribe</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>-</td>
</tr>
</tbody>
</table>

---

**Type of Event**

<table>
<thead>
<tr>
<th>SLA Management module</th>
<th>Interaction Events</th>
<th>Monitoring Result Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Subscribe</td>
<td>Publish</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>-</td>
<td>Publish</td>
</tr>
<tr>
<td>ORC Instrumentation</td>
<td>Publish</td>
<td>-</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Adjustment/Ad-hoc demonstrator</td>
<td>-</td>
<td>Subscribe</td>
</tr>
</tbody>
</table>

**Table 1 – Publish/Subscribe of monitoring-related events in the SLA Management framework**

The Monitoring module subscribes to the Interaction Event Types, i.e. service calls and responses, published by the instrumentation of the services in the ORC. Monitoring Result events, i.e. SLA violations, at the software service layer are published by the Monitoring module, whereas the Infrastructure module publishes the violations that occur for Infrastructure SLAs. Finally, the Adjustment and the Ad-hoc Demonstrator module subscribe to the Monitoring Result events. In particular, the Adjustment module uses monitoring results for producing further correlation among SLA violations and decide suitable control actions, whereas the Ad-Hoc demonstrator requires such events to show information on SLA violations to the customer of the SLA Management framework.

In more detail: the sequence diagram in Figure 26 describes the interaction of the Monitoring module and the Event Bus with the other modules of the SLA monitoring framework. This sequence diagram represents the flow of interactions required after a negotiated and provisioned SLA is submitted by the provisioning module to the Monitoring module.

Messages 1 to 3.2 show the detection of a violation of a Guarantee Term by the Monitoring module on the basis of Interaction Events (IE) produced by the ORC instrumentation. In particular, Interaction Events are pushed by the ORC to the Event Bus (1 in Figure 26). Since, in the offline configuration of the SLA Management framework, the Monitoring module has subscribed to the events published by the ORC, the Interaction Events are forwarded by the Event Bus to Monitoring (1.1). When a violation of a Guarantee Term is detected (2), the Monitoring generates a Monitoring Result Event (MRE) and submits it to the Event Bus (3). At this stage, the MRE can be forwarded to the modules, i.e. Adjustment and AdHoc Demonstrator, which have subscribed to Monitoring Result Events in the offline configuration (3.1. and 3.2).

Messages 4 – 4.2 also show how the Event Bus can be used to convey Monitoring Result Events generated at the Infrastructure Layer monitoring to the Adjustment and AdHoc Demonstrator modules.
3.5 Reacting

The previous section describes the SLA monitoring solution design in the SLA@SOI project. The monitoring module can detect software SLA violations but does not always provide sufficient information in order to decide the most appropriate course of action, when a violation occurs. Such decisions often require additional detailed information from the software and infrastructure layers that explains why a violation has occurred and can, therefore, indicate the appropriate response to it. Reacting capabilities are essential in order to avoid SLA violation and guarantee proper user experience. This is the main objective of the Adjustment Module described in this section, to trigger replanning/reconfiguration actions thus helping the service providers to guarantee their SLA commitments, and to alert the business layer.

The Adjustment module, as it has been designed and implemented in this initial phase of the project, accepts data from two different sources:

- **SLA Violations**: a violation of a GuaranteeTerm belonging to a negotiated and provisioned SLA. In the initial phase of the project, the adjustment only analyzes violations coming from the software layer.
- **SLA Warnings**: an event that is raised when a threshold has been exceeded, indicating that a problem is about to occur. In the initial phase, only the infrastructure layer emits this kind of events.

Both types of events follow the same structure based on a XML representation, which is described in Deliverable D.A3a. The correlation of the events from the different layers can help to determine the impact of the information provided by them (e.g., a software failure means that the QoS level delivered to a group of customers is not in par with the one indicated in the corresponding SLA for the contracted service). Then, a reasoning process is used to infer the appropriate corrective action and/or notification. Finally, the module orders the execution of the reconfiguration operations. This way, SLAs can be ultimately enforced or at
least automatically adjusted at the lowest possible level, namely the software layer or the physical infrastructure.

In the following subsections, a detailed description of the Adjustment module is presented. First, the relationships with other modules, including the interfaces and data types, are described. Then, the sequence diagrams for the scenarios of the initial phase of the project are shown, and finally the internal architecture of the module is presented.

3.5.1 Adjustment in the Overall Architecture

This section presents the relationship between Adjustment and other modules of the SLA@SOI framework for the ad-hoc demonstrator. Figure 27 is a simplified view of the technical architecture of the project, showing only the modules that are directly involved in reacting to SLA violations.

The Adjustment Module receives events from the Monitoring and Infrastructure modules through the Event Bus. As explained above, in this initial phase of the project, the Monitoring Module will raise an event when one of the guarantee terms of the software SLA has been violated (SLAViolation Event). On the other side, the Infrastructure layer will raise an event when some threshold has been exceeded indicating a potential problem (SLAWarning Event).

The communication between these modules and the Adjustment is done through an Event Bus. The messaging bus implements a publish/subscribe (PubSub) protocol [12][13] based on subscription to a dedicated channel. Detailed information about the event bus and the PubSub protocol can be found in deliverable D.A4a.

The correlation of the data coming from software and infrastructure layers will be used as a starting point to find the appropriate corrective action that restores the SLA to its normal status. Since both messages only contain data of their own layer (i.e. software or infrastructure), the correlation must make use of a mechanism to access the different SLA layers. This functionality is provided by the Provisioning module, which allows the retrieval of a given SLA, as well as the complete hierarchy of SLAs, as discussed in section 3.3.1.
Based on the data obtained from the software and infrastructure levels, the Adjustment module will be able to detect the source of the problem and to execute a corrective action. The actions implemented in this initial phase trigger a reconfiguration, either in the software or in the infrastructure layers. Furthermore, a notification can be sent to the business level (E-Contracting module) in case the SLA specifies the application of penalties or rewards.

### 3.5.2 Component and Interface specifications

In this section, the relationship with the other components of the framework is described in more detail through the specification of the interfaces and data types.

The interfaces consumed by the Adjustment Module have been summarized in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Exported by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createPubSubManager</td>
<td>Event Bus (Common.messaging)</td>
<td>Used to create a manager for the PubSub bus.</td>
</tr>
<tr>
<td>addMessageListener</td>
<td>Event Bus (Common.messaging)</td>
<td>Listener for the reception of events.</td>
</tr>
<tr>
<td>Subscribe</td>
<td>Event Bus (Common.messaging)</td>
<td>Used to subscribe to a channel.</td>
</tr>
<tr>
<td>businessViolation</td>
<td>e-Contracting</td>
<td>Used to inform the business layer that a specific Guarantee</td>
</tr>
</tbody>
</table>
Table 2: Module consumed interfaces

Concerning the data types used by these interfaces, they are being defined in their corresponding modules. The following table gives a brief description of them:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Defined in</th>
<th>Description</th>
</tr>
</thead>
</table>
| MonitoringResultEvent  | EventSchema in the Monitoring Module. Corresponds to SLAViolation and SLAWarning events. Deliverable D.A3a provides the complete definition. | This event signifies either the violation of an SLA at the Software Level, or a potential problem at the Infrastructure level. An event is specified by 3 fields:  
  • EventID  
  • EventContext (which identifies the sender, source, and timestamp)  
  • EventPayload  
  The payload for a monitoring result event is constituted by identifiers of the SLA and the Guarantee Term that has been violated. |
| BusinessViolationEvent | BusinessViolationEvent schema in e-Contracting. Deliverable D.A2a provides the complete definition. | This event contains the monitoring information and the reference to the business SLA that has been violated. |
| AgreementType          | WS-Agreement (inserted to the project through Negotiation module) | Data type representing a Service Level Agreement. |
| slaHierarchy           | Provisioning Module (as in section 3.3.1) | Data type representing the SLA hierarchy. The hierarchy of a given SLA contains the higher-level SLAs which rely on it |
3.5.3 Behavioural view

In this section we present the sequence diagrams corresponding to the reacting scenarios specified in the initial phase of the project. Deliverable D.A1a contains more information regarding these scenarios.

Customer fault

In this case, monitoring diagnoses an SLA violation caused by the customer. The adjustment alerts the business layer accordingly.

Figure 28: Customer fault sequence diagram

The sequence of the actions and messages is the following:
1. Software Monitoring Module sends a SLAViolationEvent, alerting that the GuaranteeTerm on the CustomerArrivalRate has been violated.
2. Adjustment consumes the event from the EventBus.
3. Adjustment finds that the customer must be notified of his incorrect behaviour.
4. Adjustment uses the software SLA identifier to retrieve the full hierarchy of SLAs.
5. Using the business SLA identifier embedded in the hierarchy, it retrieves the complete business SLA document.
6. Adjustment sends a message to e-Contracting.
Software-triggered hardware adaptation

In this case, a failure in the infrastructure is detected at the software level, as a SLA violation. The adjustment module will resolve it by reconfiguring the infrastructure. The diagram shows the complete sequence:

Figure 29: Software-triggered hardware adaptation sequence diagram

The depicted steps are explained here:

1. Infrastructure sends SLAWarning alerts that the CPU usage has exceeded a given threshold (70%).
2. Monitoring module sends a SLAViolation when the response time exceeds the value signed in the SLA.
3. Adjustment consumes both types of events from the Event Bus.
4. Adjustment correlates both events. In order to do that, the interface of the provisioning module for retrieving the hierarchy of SLAs is used. First, the full hierarchy of SLA is retrieved from the SLA registry, using the software SLA identifier as input.
5. The slaHierarchy obtained in the previous step provides access to the infrastructure SLA identifier. Then, the infrastructure SLA can be retrieved, allowing the correlation between the SLAViolation and the SLAWarnings.
6. Then, the following rule applies:
   • Preconditions:
     o GuaranteeTerm on ResponseTime violated.
     o CPU usage high (> 70%).
     o There is not inadequate customer behaviour; otherwise SLAViolations for CustomerArrivalRate would have been produced.
   • Diagnosis: insufficient capacity prediction
   • Action to be taken: infrastructure adaptation.
7. Adjustment re-provisions the infrastructure. The response will indicate if the request was successful.
A sub-scenario of the software-triggered adaptation sequence may happen when the SLA specifies the application of a penalty or reward in case such a violation occurs:

**Figure 30: Software-triggered hardware adaptation with business notification sequence diagram**

The only difference with the previous diagram is that the adjustment notifies the business violation to the e-contracting module.

**Software adaptation**

In this case, the Monitoring module detects a SLA violation. The Adjustment module will try to restore the normal situation reconfiguring the software. Figure 31 shows the complete sequence. The steps in the figure are explained below:

1. Infrastructure continuously sends SLAWarnings.
2. Monitoring Module sends a SLAViolation when the response time exceeds the value signed in the SLA.
3. Adjustment retrieves the full SLA hierarchy, as in the previous scenarios.
4. The infrastructure SLA is retrieved from the SLA registry. In this case, no correlation is found between the SLAViolation and SLAWarnings.
5. Then, the following rule applies:
   - Preconditions:
     - GuaranteeTerm on ResponseTime violated.
     - CPU usage ok (otherwise, SLAWarnings about CPU usage would have been produced).
     - There is not inadequate customer behaviour; otherwise SLAViolations for CustomerArrivalRate would have arrived.
   - Action to be taken: software adaptation.
6. Adjustment reconfigures the software engine.
3.5.4 Internal Module Architecture

The Adjustment module is composed of two main sub-modules: the Manageability Interface and the Autonomic Manager.

The Manageability Interface acts as a mediator between the Autonomic Manager and the application specific instrumentation. The SLA@SOI framework specification allows the integration of different SOA applications. 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 manageability provides a harmonized view on the different multi-layer service-oriented applications that can be easily integrated into the SLA management.

Since the Manageability Infrastructure is described with great detail in deliverable D.A3a, we will only provide here a high-level description of the component. Its design comprises three major components:

- Instrumentation
- Management Agent
- Manageability Interface

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 32 shows the Instrumentation of a software component.

![Figure 32: 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 Management Agent in turn provides a management interface that can be used by any management application.

Figure 33 shows the relationship between the Manageability Interface and the Autonomic Manager sub-modules, and serves as an introduction to the Autonomic Manager sub-module architecture. This component extends the one described in deliverable D.A3a, which was there restricted to the software level.

The EventReader accepts data through the Event Bus from two different sources and with different formats: from the software context in form of SLAViolations, and from the infrastructure layer in form of SLAWarnings. The Event Analyzer correlates both types of events, and using various analysis techniques, determines the impact, or potential impact, of this information. Then, a set of reasoning processes are used to determine the most appropriate corrective action (or set of actions) to take. Finally the Execution Manager orders the reconfiguration operations, either to the software or to the infrastructure layers. As explained before, the manageability interface is used to reconfigure the software processes. Besides this, a notification can be sent to the business layer when needed.

Note that the main challenge of the Adjustment Module respect the description provided at deliverable D.A3a is the inclusion of the infrastructure data, the correlation of data from different sources and the capacity of reacting on the different layers. All this features convert this module into a holistic adjustment across layers.

Despite of its name, the Adjustment Module in the initial implementation of the project module will not provide all the characteristics defining an autonomic system. It will be refined in following versions to achieve a real autonomic behaviour. More details about autonomic systems and autonomic-based adjustment are provided in deliverable D.A3a.
In order to better understand the role the Manageability Interface plays in the Adjustment module, the scenario shown in Figure 29 (triggering a software reconfiguration) will be refined. When the running service is composed of various atomic services, it is necessary to find out which atomic service is causing the problem. The Manageability provides an analysis of the response time of provided service operation call in relation to the response time of the integrated services. Therefore, the Autonomic Adjustment will be able to request the data to calculate the metrics and thus identify the cause of the SLA violation.

Figure 34 shows a sequence diagram explaining how the Adjustment uses the Manageability Interface:

1. The SLA Management calls the `getExecSCActions()` operation with a parameter identifying the operation on the composed service:
The Manageability Interface returns all executed service operation call instances of the operation.

2. The SLA Management requests for each of these instances the corresponding child instances.

Now, the Autonomic Adjustment can compare the elapsed time of the service operation call instances of the composed service with the elapsed time of the corresponding reference operation call instances of the operations of the atomic services. An analysis of the timing information will allow the Adjustment to find out if the failure was either caused by the engine that runs the service composition or in one of the atomic services.

When a reconfiguration of the software engine becomes necessary, it is also realized through specific methods provided by the Manageability Interface.
4 Implementation

4.1 Template Registry

For the 1st year prototype it has been decided to adopt WS-Agreement\(^5\) as the underlying data-model for representing SLA Templates, and to implement the “SLA Template Registry” in a trivial manner as an opaque store of WS-Agreement documents – accessible in bulk, or individually by identifier. Beyond the 1st year prototype, however, a more sophisticated form of SLA Template Registry is envisaged. In particular, we aim to develop a fully query-able SLA Template Registry enabling a template-based approach to the discovery of services. This section presents a preliminary specification for the SLA Template Registry in this extended sense. A working prototype of this registry has already been implemented, but (at the time of writing) has not yet been functionally integrated into the (prototype) SLA@SOI architecture.

This section assumes that readers are familiar with the SLA Model presented in section 2.1. The content is organised as follows:

- **Section 4.1.1 “Approach”** describes the template-based approach to service discovery which motivates the design of the registry.
- **Section 4.1.2 “Query Resolution”** explains the nature of the query resolution problem in regards the registry.
- **Section 4.1.3 “Data-Model”** describes a preliminary data-model for the representation of registry content (i.e. SLA Templates).
- **Section 4.1.4 “SLA Template Registry Interface”** defines the interface for the SLA Template Registry (in terms of operations over the data-model).

4.1.1 Approach

A basic assumption of the SLA@SOI project is that the use of a service is always regulated by an (implicit or explicit) SLA between service provider & customer. This SLA contains both a description of the offered/consumed service and various other agreed terms, including:

- Quality of Service (QoS) properties,
- behaviours that the customer or provider legally guarantee to the other party,
- penalties to pay in case of infringements,
- information on what, and how, to monitor for infringements.

In such a context, the concept of service discovery takes on added significance. While in the traditional view a customer typically searches for a service endpoint providing specific functional & QoS properties, in an SLA context the customer searches for a provider able to supply a specific kind of service and also willing to make certain guarantees and agree to specific terms. In other words, we assume that in real business scenarios, customers search for a specific kind of SLA – rather than just a specific kind of service. Moreover, we assume that final agreed SLAs are typically the result of a negotiation between customer and provider.

\(^5\) specifically; the WSAG4J (WS-Agreement for Java) API.
The templates that providers advertise should, in general, be understood as assertions of general intent – subject to modification by negotiation.

The SLA Template Registry is conceived with just this SLA-aware approach to service discovery in mind. In short, since an SLA Template (as defined by the SLA Model) represents a set of SLAs, then:

i) the SLA Template Registry – conceived as a publicly accessible repository of SLA Templates – provides a suitable means by which providers can advertise the kinds of SLAs they are willing to agree to (& by extension the services they offer).

ii) an SLA Template also provides a suitable means by which customers may express their needs – such that the service discovery process can be conceived in terms of matching customer queries, expressed as SLA Templates, against provider offers, again expressed as SLA Templates accessible through an SLA Template Registry.

iii) finally, an SLA Template – conceived as a customisable SLA – provides a suitable starting point for negotiation; the template’s free-variables and creation-constraints define the elements of the agreement which are (explicitly advertised as) open to negotiation.

It should be noted that the template-based approach to discovery described above has an essentially semantic character, which takes the discovery problem beyond that of formal (symbolic) pattern matching. In brief, since each SLA Template represents a set of SLAs, the discovery problem can be readily characterised as one of matching symbolic class definitions on the basis of their interpreted extension (discussed further in section 4.1.2).

Finally, implicit in the notion of template-based service discovery is the idea that SLA Templates can serve as service descriptions. This notion is captured formally in the SLA Model, which defines both SLA and SLA Template classes (read concepts) as extensions of the ServiceDescription class (summarised in Figure 35).

![Diagram](image)

**Figure 35 : Relations between SLAs, SLA Templates & ServiceDescription**

Accordingly, we can treat an SLA Template Registry as subspecies of generic Service (Description) Registry – as depicted in Figure 36 (a ‘registry’ is modelled as a composition of its contents). A side-effect of this approach is that it provides a means for application developers to incorporate legacy Service (Description) Registries into the SLA@SOI framework (not discussed here).

---

6 for completeness we can include ‘trivial’ negotiations comprising just an implicit acceptance of the SLA by the customer.
4.1.2 Query Resolution

Earlier we stated that the adoption of a template-based approach to service discovery has an essentially semantic nature – in that the resolution of SLA Template Registry queries entails matching symbolic class definitions on the basis of their interpreted extension. This section provides a brief explication of this assertion.

To begin, the SLA Model states that, conceptually, an SLA Template represents a set of SLAs – namely; those SLAs which may be generated from the template. As such, the template may be considered as defining a class of SLAs – such that the template is essentially a symbolic class definition (with the set of SLAs as extension). In addition, however, each SLA in turn conceptually denotes a set of potential service executions – namely; all the different, possible executions of the service which would conform to the SLA. Suppose, for example, that a particular SLA – regarding some service, S, comprising a single operation X – specifies a guarantee that the completion-time of X will be less than 5ms. For simplicity, we will represent this guarantee symbolically with the formula; \( \text{ct}(X) < 5 \). In principle, this guarantee can be fulfilled by any one of an infinite number of possible executions of operation X – namely; all those executions in which \( \text{ct}(X) < 5 \) holds true.

Thus, there are two levels of symbolic interpretation in effect; the first maps a symbolic SLA Template onto a set of symbolic SLAs, and the second maps each Symbolic SLA, in turn, onto a set of possible, physical states of affairs (the particular service executions fulfilling that SLA). It is this set of physical states of affairs which should be taken as the proper (interpreted) extension of the SLA Template.

Now consider the case that the service S (above) is advertised as a registered SLA Template – that is; a template containing the guarantee \( \text{ct}(X) < 5 \); whose extension we will denote as \( E_{\text{offer}} \) (=the set of possible service executions which fulfill the constraint \( \text{ct}(X) < 5 \)). Next, suppose a customer requiring service S but willing to accept a wider range of completion-times for operation X say: \( \text{ct}(X) < 10 \). The customer’s need is expressed as a query, represented by an SLA Template containing the symbolic guarantee \( \text{ct}(X) < 10 \); whose extension we will denote as \( E_{\text{query}} \). Put another way, we can state that the customer’s needs are fulfilled by any particular service execution which is a member of the extension \( E_{\text{query}} \). In this case, the extension of the offered template, \( E_{\text{offer}} \), is a subset of \( E_{\text{query}} \) – and we can thus conclude that the customer’s needs would be satisfied by agreeing to an SLA conforming to the offered template.

In sum, the query resolution problem is one of finding offered templates whose (interpreted) extension is a subset of the (interpreted) extension of the customer’s query. This problem is exacerbated when we consider that:

- SLA Templates may also contain customisable values – i.e. free-variables;
that such variables will likely have restricted ranges (cf. creation constraints), and

- in the most complex scenarios, that these ranges may be discontinuous.

In terms of implementation it should be apparent that query resolution necessarily entails fairly sophisticated mechanisms. Our current approach is to tackle these issues in terms of formal logic & reasoning. To this end, ongoing work on the SLA Template Registry involves the development of rules & representations adequate to the query resolution task. The data-model presented in the next section provides a simplified & preliminary snapshot of current state of these representations. We leave a complete specification to later deliverables.

4.1.3 Data-Model

This section presents a data-model for the SLA Template Registry – i.e. a representational model for the SLA Templates stored in the registry. The data-model is defined in terms of (UML) interfaces – in order to clearly distinguish data-model entities from their conceptual SLA Model class counterparts, and also due to the current (Java) implementation choices.

As implied in the previous section, the data-model and query-resolution mechanisms are heavily interdependent – in that the specification of each depends on the specification of the other. As such, and since work on query-resolution is still in its early stages, the data-model will necessarily change. For the present, therefore, we provide here only a simplified, preliminary overview of the data-model.

![Hierarchy of Service Descriptions (data model)](image)

Figure 37: Hierarchy of Service Descriptions (data model)

As an additional constraint, we should note that it is a project objective that the data-model (applied in a web-service context) be consistent with WS-Agreement.

As described in the conceptual SLA Model, an SLA Template - as a representational artefact - can be treated as a customisable SLA, and both SLAs and SLA Templates are in turn conceived as (augmented) Service Descriptions.

\[\text{a formal mapping from WS-Agreement to the SLA Template Registry data-model has been specified, but is not presented here.}\]
We exploit these (syntactic) relations in the data-model by assuming the hierarchy\(^8\) depicted in Figure 37.

The Entity interface (see Figure 37) serves as the root interface for all identifiable elements in the data-model, and corresponds to the Nominal class defined in the SLA Model (which provides a hook for annotations). The generic Set\(<X>\) interface denotes an abstract collection of elements (requirements for ordering the collection will be noted as they arise). The following sub-sections describe the ServiceDescription, SLA and SLATemplate interfaces.

**Service Description Interfaces**

The ServiceDescription interface (and related interfaces) is shown in Figure 38, and described below.

**Figure 38 : ServiceDescription Interfaces**

Following from the conceptual SLA Model, we treat a ServiceDescription as essentially comprising a set of (descriptions of) service operations. In its present state, the data-model restricts these operations to the simple exchange of messages (e.g. corresponding to operations as defined in WSDL). Individual operations are modelled by the Operation interface – which exposes the following operations\(^9\):

i) \(\text{name}()\): retrieves the name of the message as a simple String value,

ii) \(\text{inputArguments}()\): retrieves an ordered list of named & typed arguments (modelled as a set of Literal).

iii) \(\text{outputArguments}()\): retrieves an ordered list of named & typed arguments (modelled as a set of Literal).

iv) \(\text{errors}()\): retrieves a list of possible exceptions, or faults - where for simplicity we assume that each error is identified by some standard Term (for present purposes we assume each Term has a String representation).

---

\(^8\) the SLA interface is not strictly necessary.

\(^9\) for simplicity we do show only access operations. Corresponding operations for setting data values are assumed.
In addition to operations, and in support of extensibility through annotations, we also permit a simplistic form of metadata as ServiceDescription content. In its present state this metadata takes a conventional predicate form – attributing some valued-property to an Entity. These predicates are captured by the Metadata interface which exposes the following operations:

i) subject(): retrieves the Entity to which the property is attributed,

ii) predicate(): retrieves a standard Term denoting the predicated property,

iii) object(): retrieves the value of the predicated property (for present purposes modelled as a Literal).

The Literal interface encapsulates a typed constant value and exposes two operations:

i) stringValue(): retrieves a String representation of the constant,

ii) datatype(): a standard Term denoting the constant’s datatype.

**Service Level Agreement (SLA) Interfaces**

The basic elements of the SLA interface are depicted in Figure 39 - with additional detail given for service-constraints in Figure 40, and for business-values in Figure 41 – all of which are described below.

---

**Figure 39 : SLA Interfaces**

Following from the conceptual SLA Model, we treat an SLA as essentially a service description augmented with guarantee terms (codifying obligations). Accordingly, the SLA interface extends the ServiceDescription interface with operations for
accessing these guarantee terms. Convenience operations are also defined for accessing the various observables, service-constraints and business-values entailed by guarantees.

A guarantee term is modelled by the GuaranteeTerm interface, which exposes the following operations:

i) precondition(): retrieves the precondition on the guarantee, expressed as a service constraint (or null if no precondition is specified).

ii) guaranteed(): retrieves the service constraint which is guaranteed by the guarantee term.

iii) businessValues(): retrieves the set of business values associated with the guarantee term.

Service constraints (expressing the performative aspects of obligations) are modelled by a generic ServiceConstraint interface, which comes in 3 specific forms – as depicted in Figure 40 (described below).

---

**Figure 40: ServiceConstraint Interfaces**

The ServiceConstraint.Simple interface encapsulates a simple constraint over a single observable – i.e. a constraint of the form:

\[ \text{<observable> comparison-operator <value>} \]

e.g.: response_time_of_operation_X < 5ms

An observable is considered an Entity in its own right (i.e. it can be annotated), and is captured by the Observable interface, which exposes the following operations (see Figure 39):

i) property(): retrieves a standard Term denoting the observed property (e.g. standardised forms of; ‘completion-time’, ‘availability’, etc),

ii) parameters(): retrieves an ordered list of various parameters required to fully specify the observed property. For example, ‘completion-time’ is an observable property of service operations (the time it takes for the operation to complete). Hence ‘completion-time’ would require a
parameter specifying which operation is the subject of observation. At present, valid parameters include:

a. any Entity: e.g. an operation (as in the example above),

b. any Term: denoting some standard parameter (e.g. specifying required measurement units),

c. any Literal: as a catch all for any other kind of required parameter.

At present, the data-model does not define or prescribe any standardised set of observable property terms or their corresponding parameter requirements.

The Operator enumeration serves to define the available comparison operators – which, for present purposes are just the standard operators: =, != (not equal), <, <=, > & >=. The ServiceConstraint.Value interface is a generic interface encapsulating constraint values; for SLAs, these must be constants (Literal), while for SLA Templates, we permit also free-variables (as described shortly). The ServiceConstraint.And & ServiceConstraint.Or interfaces serve to model (nested) conjunctions & disjunctions (resp.) of service constraints – each of which is defined as an unordered set of ServiceConstraint.

Business values (expressing the utility aspect of obligations) are encapsulated by the BusinessValue interface (shown in Figure 41), which exposes the following operations:

i) obligatedParty(): retrieves a flag stating whether the business value applies to the PROVIDER or to the CUSTOMER.

ii) cost(): retrieves the ‘value’ of the business value, which for present purposes is treated as a constant (Literal).

iii) costType(): retrieves a flag stating whether the cost is a PENALTY or REWARD.

iv) priority(): retrieves a floating-point value indicating the importance of the business value.

![BusinessValue Interfaces](image)

**Figure 41 : BusinessValue Interfaces**

Finally, it should be noted that SLA monitoring policies are not explicitly represented in the (current) data-model. Instead, we presently rely on the use of annotations (cf. the Metadata interface) to provide any additional information required for monitoring.
**SLA Template Interfaces**

As described in the SLA Model (see also Figure 37) we can model an SLA Template, treated as a representational artefact, as a customisable SLA – i.e. an SLA augmented with range constrained free-variables. In the data-model, we encapsulate both variables & their constraints in a single CreationConstraint interface – instantiations of which can, in principle, be inserted into the SLA Template at any point at which a customisable value is required. In its present form, however, the data-model only permits the customisation of service constraint values – i.e. instantiations of the ServiceConstraint.Value interface. The SLATemplate interface extends the SLA interface, and for convenience provides access to the set of embedded CreationConstraints – as shown in Figure 42 (described further below).

---

**Figure 42 : SLA Template Interface**

The CreationConstraint interface is intended generic super-interface for various subtypes of constraint. It exposes only one operation:

- isNegated() : which returns true if the constraint is a logical negation.

The CreationConstraint.Simple interface encapsulates a the most basic type of creation constraint – namely; a constraint of the form:

```
comparison-operator <value>
```

e.g.: < 5ms

Permitted comparison operators are defined by the Operator enumeration (see Figure 40), while constraint values are restricted to constants (Literal). The CreationConstraint.And & CreationConstraint.Or interfaces serve to model (nested) conjunctions & disjunctions (resp.) of creation constraints – each of which is defined as an unordered set of CreationConstraint. Finally, the CreationConstraint.OneOf interface serves to encapsulate an enumeration of permitted values – and is thus modelled as a set of Literal.
4.1.4 SLA Template Registry Interface

This section completes the SLA Template Registry specification by defining the interface for the registry itself – which simply encapsulates the use cases described in Section 3.1.1 in terms of the data-model described in the previous section. The interface is shown in Figure 43 (SLA Template identifiers are modelled here as simple Strings, as are the acknowledgements returned by the publish & revoke operations).

![Figure 43: SLA Template Registry Interface](image)

4.2 WSAG4J Adaptation

This section describes the integration process of the WSAG4J WS-Agreement implementation [14] into the SLA@SOI codebase and introduces some classes related to handling agreements. Operations such as reading and validating templates, creating offers and establishing agreements are explained through some more detailed examples.

4.2.1 WSAG4J as a maven library

We used WSAG4J version 0.1.10 for this integration. Taking it as starting point and considering that implementation of SLA@SOI modules are maven-based dependencies, a “maven-ised” library of wsag4j was created and is used throughout the project codebase, at the points where management of WS-Agreement documents is necessary.

The procedure to create the library „wsag4jslasoi-wsag4j“ will be described here in brief. All wsag4j-kernel classes were extracted from the respective war file and decompressed. The kernel classes refer to the following sub-libraries: wsag4j-security, wsag4j-server and wsag4j-types. Maintaining the hierarchy of those extracted classes they were in turn compressed into a new Java Archive file (wsag4jslasoi-wsag4j-1.2.jar).

In order to be able to import this library from a maven project it was necessary to deploy it into a central maven repository. To do that, maven requires a descriptor file (pom.xml). This file must contain a list with all dependencies under WEB-INF/lib (as found in the wsag4j-war file). Note that this list must be formatted according to maven specifications.

Once the wsag4jslasoi-wsag4j-1.2.jar file and the POM file are created, the following command allows to publish the library as maven artifact:

Locally:
```
mvn install:install-file -DgroupId=org.slasoi.common.wsag4slasoi-wsag4j -DartifactId=wsag4slasoi-wsag4j -Dversion=1.2 -Dfile=wsag4slasoi-wsag4j-1.2.jar -Dpackaging=jar -DpomFile=./pom.xml
```

Maven Repository:
```
```
In this way the mavenized-library is available to use from within a maven module. The library can be referenced adding just the following dependency in the related pom.xml file:

```xml
<dependency>
    <groupId>org.slasoi.common.wsag4slasoi-wsag4j</groupId>
    <artifactId>wsag4slasoi-wsag4j</artifactId>
    <version>1.2</version>
</dependency>
```

### 4.2.2 WSAG4J Extension

This package extends the usability of wsag4j through the addition of wrappers and utility classes to be able to handle agreement and related objects easily. These classes are separated into following groups:

- **DAO Classes:** org.slasoi.common.wsag4slasoi.dao
  
  **Description:** Persistence, validation and Navigation utilities are supported via wrappers classes

  [Diagram of WSAGWrapper class]

  **Figure 44: WSAGWrapper class diagram**

- **Exceptions:** org.slasoi.common.wsag4slasoi.exceptions
  
  **Description:** Definition of exceptions for validation and parsing utilities

  [Diagram of Exceptions class]

  **Figure 45: Exceptions class diagram**

- **Miscellaneous:** org.slasoi.common.wsag4slasoi.misc
  
  **Description:** Utilities for loading and parsing DAO-collaboration classes are defined here.
4.2.3 How to use the extension library

To be able to use this extension-library, the following dependencies must be included in the module POM file:

```xml
<dependency>
  <groupId>org.slasoi.common.wsag4slasoi-wsag4j</groupId>
  <artifactId>wsag4slasoi-wsag4j</artifactId>
  <version>1.2</version>
</dependency>

<dependency>
  <groupId>org.slasoi.common.wsag4slasoi-extension</groupId>
  <artifactId>wsag4slasoi-extension</artifactId>
  <version>1.0.2-SNAPSHOT</version>
</dependency>
```

Figure 47: POM file definitions

Now that dependencies are resolved, some examples can be described: loading and validation of an agreement template document, loading and exploration of an agreement offer and as a last use case the persistence of an agreement.

**Loading and Validation of an Agreement Template**

**Involved classes:** org.slasoi.common.wsag4slasoi.misc.SLASOIUtilities
The validation of an agreement template is possible via the method validateTemplate of SLASOIUtilities class, as can be seen in the example above. If the template file is consistent and WS-Agreement compliant (i.e. syntactically correct), the validateTemplate method does not throw any exception, otherwise a TemplateValidatorException will be thrown describing all template inconsistencies. If the template includes customized schemas, they can be added to the validation procedure via the respective configuration file (catalog.xml).

**Loading and Exploration of an Agreement Offer**

**Involved classes:** org.slasoi.common.wsag4slasoi.misc.SLASOIEplorer

The class SLASOIEplorer provides the method loadAgreementOffer which returns an instance of AgreementOfferDocumentWrapper. From this wrapper the offer content such as agreementId, Context and Terms can be queried. A detailed list about available terms is depicted in the figure below:

**Figure 48: Template validation**

```java
public void validation()
{
    try
    {
        FileInputStream fis = new FileInputStream( "template.xml" );
        SLASOIUtilities.validateTemplate( fis );
    }
    catch ( TemplateValidatorException e )
    {
        e.printStackTrace();
    }
    catch ( Exception e )
    {
        e.printStackTrace();
    }
}
```

**Figure 49: SLASOIEplorer class**

```java
public void explorer()
{
    try
    {
        AgreementOfferDocumentWrapper offer = SLASOIEplorer.loadAgreementOffer( new File( "offer.xml" ) );
        AgreementType agreementOffer = offer.value().getAgreementOffer();
    }
    catch ( Exception e )
    {
        e.printStackTrace();
    }
```

The class SLASOIEplorer provides the method loadAgreementOffer which returns an instance of AgreementOfferDocumentWrapper. From this wrapper the offer content such as agreementId, Context and Terms can be queried. A detailed list about available terms is depicted in the figure below:
Persistence of an Agreement

Involved classes:
  org.slasoi.common.wsag4slasoi.misc.AgreementSerializableUtils

Persistence of agreements refers to how they can be stored and read from some stream. A stream can be represented by a File, a Database or some other data flow.

For reasons of flexibility, the persistence or serialization of an agreement object codifies it into a String java object. The figure below shows an example of how an agreement object can be serialized to and deserialized from a String:

```java
Figure 51: Agreement objects (de)marshaling

The methods getAgreementWrapperAsString and getAgreementWrapperFromString of class AgreementSerializableUtils allow the symmetric conversion of an agreement and a string object.

4.2.4 SLA Registry

The SLA Registry provides functionality to save and restore agreement documents and hierarchies of such associated documents. Since the library wsag4j does not implement serialization of those objects, the registry uses the marshalling facilities of WSAG4SLASOI-extension.
The DatabaseFactory represents the main class of the registry implementation. Its methods are described in following:

**getInstance:** implements the single-skeleton pattern for this class

**getSLA:** returns an agreement document handled by this registry

**getHierarchy:** returns a list of all agreements associated with the agreement which is identified by the operation’s input argument

**commit:** save in file the agreements which are cached at this moment

**load:** load in cache all agreements located in file

**storeSLA:** stores in the registry the agreements provided as argument. Note that to make the agreements persistent, a call to the commit() method is necessary.

The SLA hierarchy data type is defined in XML schema as follows. The “client” and “subcontractor” elements, of string type, are identifiers of the SLAs, as they are used by the getSLA() method.

```xml
<xsd:complexType name="slaHierarchy">
  <xsd:sequence>
    <xsd:element name="client" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="subcontractor" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

By default the factory reads from and writes to a file named “negotiation-database.dat” located in the classpath. This filename can be changed via system parameter “org.slasoi.negotiation.database”.

### 4.3 Monitoring Framework

In this section we discuss the internal implementation of the Monitoring module and of the Event Bus.

**The Monitoring module**

The class diagram representing the Monitoring module implementation is shown in Figure 52.

As discussed in deliverable A3a, while presenting service monitoring, the Monitoring module relies on two core monitor engines, i.e. the core monitor engine developed by CITY (citymonitor) and by FBK (fbkmonitor).

The outer layer of the architecture shown in Figure 52 (Monitoring class), therefore, implements the interface `startMonitoring()` to receive SLAs from the Provisioning module and has also the responsibility of initialising the `citymonitor` and `fbkmonitor` at the beginning of operations.

Each core monitor in the Monitoring module has been wrapped to comply to the scenario envisioned in the SLA@SOI Year 1 demonstrator. As a sample, Figure 52 shows the extensions made to the `citymonitor` core engine.
The CityMonitor class includes an EventBusManager object, to interact with the EventBus (i.e. subscribing to channels and pushing events), a MonitorMessageListener, for receiving events from the EventBus, and an EventManager object, for parsing events formatted according to the monitoring event format described in deliverable A3a.

Other extensions concern the generation of Monitoring Result Events (MonResultGenerator class), when a violation of a Guarantee Term is detected, and extensions required for generating monitoring rules to feed the core engine (slamanager.* classes).

In particular, the slamanager.* classes allow the citymonitor to:

- Parse a SLA received from the Provisioning;
- Generate a monitoring rule for each Guarantee Term in the SLA with which the core monitor engine can be fed;
- Maintain the correspondence between monitoring rules and SLAs;
- Retrieve information on Guarantee Terms and SLAs once the violation of a monitoring rule is detected.

Similar extensions have been implemented for the fbkmonitor.

![Figure 52 – Monitoring module implementation: class diagram](image)

Additional resources of the Monitoring module represented in Figure 52 are the EventManager class and the org.slasoi.monitoring.eventformat package. The former exposes basic functionality for marshalling andmarshalling events received from and pushed to the EventBus from their String to XML object representation. The latter includes the classes automatically generated by the JAXB 2.0 library for parsing and managing the XML format of events.

### The Event Bus

The Event Bus implements a Publish-Subscribe infrastructure that allows the module of the SLA Management framework to exchange monitoring-related events, i.e. Interaction events and Monitoring Result events.

Within the SLA@SOI ad-hoc demonstrator, we choose to rely on the Openfire open source implementation of the XMPP Pub-Sub specification for publish-subscribe infrastructures.

The XMPP PubSub specifications [15] are an extension of the eXtended Messaging and Presence Protocol (XMPP) for generic publish-subscribe functionality. The protocol enables XMPP entities to create nodes (topics) at a PubSub service and
publish information at those nodes; an event notification (with or without payload) is then broadcasted to all entities that have subscribed to the node. The OpenFire server [16] is a real time collaboration (RTC) server licensed under the Open Source GPL, which implements many of the XMPP specifications and, in particular, the XMPP PubSub specifications. In particular, a running instance of the Openfire server includes a PubSub service.

The Openfire server provides an API, namely Smack, for exploiting its ability to implement a publish-subscribe infrastructure. Smack is a library for communicating with XMPP servers to perform real-time communications using the Openfire server. Smack is meant to be easily embedded into any existing JDK 1.5 or later Java application. It has no external dependencies and is optimized to be as small as possible. The library is constituted by the following JAR files to provide more flexibility over which features applications require:

- smack.jar -- provides core XMPP functionality and is the only required library. All XMPP features that are part of the XMPP RFCs are included.
- smackx.jar -- support for many of the extensions (XEPs) defined by the XMPP Standards Foundation. In particular, it includes support to the XMPP PubSub protocol extension.

Smack provides an API for external applications to get connections to an XMPP server and, for what concerns publish-subscribe, it allows the creation of new channels, to push messages, and to implement message listeners to an existing XMPP PubSub service.

As shown in Figure 53, the Event Bus is thus implemented as an ad-hoc wrapper of the API provided by Smack to the Openfire server implementation. In particular, the interface exposed to other modules in the SLA Management framework is the one described in Section 3.4.

![Figure 53 – SLA@SOI wrapping of the Smack-Openfire API](image.png)

Figure 54 shows a Class diagram concerning the actual implementation of the EventBus. The wrapping the Smack API is made in two stages. A first wrapping is provided by the PubSubManager class, belonging to the org.slasoi.common package, which exposes primitive methods for connecting and interacting with a server. The class EventBusManager implements the actual interface of the EventBus described in Section 3.4.
4.4 **SLA Enforcement Mechanisms**

This section describes the implementation details of the adjustment module introduced in section 3.5.

As stated before, the Adjustment Module is composed of two sub-modules: the Autonomic Adjustment and the Manageability Interface.

### 4.4.1 Autonomic Adjustment

In this initial phase of the framework, the Adjustment Module only implements a basic reacting functionality. Therefore, its design and implementation are kept simple and involve only a small amount of classes. Consequent versions will lead to a more elaborated architecture, and a real autonomic system. Figure 55 shows the class diagram for the Autonomic Adjustment sub-module of the initial phase of the project.

The EventReader class has an instance of PubSubManager (implemented in the org.slasoi.common.messaging package of the project) which provides a connection to the event bus. Every EventReader is subscribed to a channel from which it receives events. The Autonomic Manager can be composed of one or more EventReaders, allowing a dedicated reader to be instantiated for each channel in use (i.e., a dedicated channel for infrastructure data and another for software events). The format of the events is defined in the Event schema in deliverable D.A3a.

The EventAnalyzer class, after unmarshalling the xsd file into EventInstances, will try to correlate the events coming from the software layer with the ones coming from the infrastructure level. It uses a specific data type, TableEvents, to store the received events, the associated information (e.g., the corresponding SLAs) and the correlations between software and infrastructure events.
The Planner class has the responsibility of finding the most appropriate corrective action to restore the normal behaviour of the system. It will also decide if a notification should be sent to the business layer. The decision is taken based on a set of previously defined policies.

![Class diagram of the Autonomic Adjustment Module](image)

**Figure 55: Class diagram of the Autonomic Adjustment Module**

The main class is the AutonomicAdjustmentManager. The Manager controls the activity of the EventReader, Analyzer, Planner and ExecutionManager.

The ExecutionManager takes as input the action decided by the Planner, and orders its execution. Therefore, it implements the interfaces to the Manageability Interface and to external modules (Infrastructure and e-Contracting).

### 4.4.2 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, 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.

Figure 56 shows the class diagram of the Manageability Data Model.
Figure 56: Manageability Data Model

ServiceComponentActionDefinition and ServiceComponentAction are the central classes of the Manageability Data Model. 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.

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.

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.

A complete description of the Manageability Interface can be found in deliverable D.A3a.
5 Conclusions

5.1 Summary

This document provides an overview of the results of the work that WP A5 did during the first year of SLA@SOI project execution. The main output of our efforts was:

- A complete, coherent, requirements-driven SLA model. It formalises actors, actions, relationships, and entity structure with regards to the agreement documents, offers, templates, etc;
- A SLA template registry starting from this model definitions, with the final aim of a query-able service that can be used for SLA-aware service discovery;
- The adaptation of an existing WS-Agreement implementation to be used in form of a library throughout the project;
- The design and ad-hoc implementation of the SLA-aware provisioning mechanisms, which take into account SLA scheduling issues with regards to service activation lead times and relevant constraints;
- The formalisation of the SLA hierarchy problem, and the materialisation of such a hierarchy in the context of the ad-hoc demonstrator;
- The implementation of event-driven monitoring facilities that detect SLA violations and trigger the respective reactions;
- The foundations for a proper autonomic reaction framework, to be used when such violations take place.

This work is necessarily considered as a first approach to the problem, and will be refined based on additional requirements as explained in the upcoming Section 5.2.

5.2 Outlook on Future Work

In the Introduction section, we already referred to the fact that the current design was its roots in the Open Reference Case (ORC). Having a use case to build upon early on allowed us to make a number of assumptions and design choices, which will be revised in Year 2 in light of the new requirements, stemming from the industrial use cases analysis. This is an extensive analysis taking into account also comments from related projects (for more information, please see deliverable D.B1a). It already allowed us to extract a set of generic, necessary features that the project framework must accommodate; as such, these requirements will drive the upcoming architectural update and the consequent development in the coming two years. In Section 5.2.1, below, we are listing these requirements with brief discussion on each one.

5.2.1 Major requirements for the SLA@SOI framework

Deliverable D.B1a is elaborating on requirements gathered from a number of use cases. Starting from those requirements we can identify certain A5-related functionalities that our architecture and framework must offer, and principles on which our ideas should build:
• Well-defined and extensible set of agreement terms, business values and constraints
The framework must be based on a complete model, which offers a way to define service description terms, their properties, target values (guarantees) for these properties, constraints over acceptable target values, and rewards/penalties for violating the agreement regarding the guarantees. A well-defined small set of terms common throughout all use cases must be available by default, but the model must be extensible so that users can define new such constructs and use in their own use cases.

• Well-defined agreement term composition semantics
When we are examining composite services and their SLAs, we have to examine “composite SLAs” as well. However, while service composition is relatively well understood, SLA composition could mean largely different things – e.g., two SLAs concatenated in one, or the terms of two SLAs combined in a set of differently represented terms (i.e. different language, different semantics, etc). Term composition semantics are of course use-case specific and depend on the terms themselves, however some minimal definitions for the default set of terms must be defined.

• Definitions for management of SLA hierarchies
Starting from agreement composition as mentioned above, we have to define all those conceptual and architectural constructs for managing hierarchies. How are SLA relationships stored? Where are they kept? How can one discover those relationships and traverse hierarchies as such? The framework should answer to these questions and provide the capability to perform all respective actions.

• Negotiation and renegotiation
Negotiation refers to the very establishment of the SLA, and is the point in time when the parties involved make their offers, until the offers of one party become acceptable by the other one. According to the requirements, there is a strong need for multi-round negotiation, with counter-offers for each offer, until equilibrium is reached. Negotiation must be able to start from any of the parties involved, and may include more than two parties (e.g. auctions). Similarly, re-negotiating SLAs should be possible (unless the original SLA forbids it), and should be possible to initiate from any of the parties involved.

• Autonomic principles for SLA management
Establishing, provisioning and monitoring SLAs cannot scale when done centrally. It is clear from the requirements that autonomic principles must be applied to the framework, so that different services and service providers can negotiate, plan, execute and adjust SLAs for themselves.

• Adjustment of SLA provisioning
Scaling the infrastructure, or re-configuring the software could mitigate a violated IT SLA. Here we refer to the latter: When we have a violation, the framework should be clever to find out what went wrong, and explore re-configuration recipes before resolving to the elastic capabilities of the infrastructure.

• Monitoring as a SLA term
Many use case requirements indicate that monitoring is (naturally) one of the most important parts of the framework. However, monitoring needs are not constrained to identifying violations; rather, the framework must allow fine-grained definition of monitoring policies, e.g. what monitoring data is stored, who has access to it, etc. That kind of policies must be negotiated during SLA establishment, therefore making monitoring also a part of the agreement.

• SLA planning and activation mechanism
SLAs can be negotiated to become effective immediately, or to start in the future. As such, a mechanism to plan for them during negotiation, and
activate them when the time is right, is needed to be available in the framework. Activation of IT SLAs includes provisioning the software and infrastructure, configuring the software, etc.

- **Synchronous (online) SLA management**
  A way should be provided for human operators to perform manual negotiation (i.e. use the framework only for reasons of formalising agreements in machine-readable ways), to monitor performance/state of agreements in real-time, to manually start a re-negotiation or even cancel them and pay the penalties as defined in them.

- **Standards compliance**
  It has been made clear from the requirements that it is expected to support web services and relevant standards: WSDL/SOAP, WS-Agreement, etc.

- **SLA-template-based service discovery**
  This requirement has affected the early stages of the project already. Sometimes the line between functional and non-functional properties of a service is blurry. As such, service discovery and SLA template discovery can be combined into one step, as long as templates include a proper description of what the service does. This way, the discovery and establishment mechanisms become better integrated, and the whole framework becomes more efficient.
6 References

Appendix A: Glossary

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

Appendix B: Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAM</td>
<td>Autonomic Adjustment Management</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BATNA</td>
<td>Best Alternative To Negotiated Agreement</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>DAO</td>
<td>Data Access Objects</td>
</tr>
<tr>
<td>GRAAP</td>
<td>Grid Resource Allocation and Agreement Protocol</td>
</tr>
<tr>
<td>HTTP(S)</td>
<td>HyperText Transfer Protocol (Secure)</td>
</tr>
<tr>
<td>IE</td>
<td>Interaction Events</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>MRE</td>
<td>Monitoring Result Event</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-functional property</td>
</tr>
<tr>
<td>ORC</td>
<td>Open Reference Case</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RFQ</td>
<td>Request For Quotes</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLAT</td>
<td>Service Level Agreement Template</td>
</tr>
<tr>
<td>SLO</td>
<td>Service Level Objective</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>WSA4J</td>
<td>Web Services Agreement for Java</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
</tbody>
</table>