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

Deliverable D.A5a
Foundations for SLA Management

Keywords:
Service Level Agreement, SLA Management, SLA Model, SLA Hierarchies

Due date of deliverable: 31st July 2010
Actual submission to EC date: 10th September 2010

Start date of project: 1st June 2008
Duration: 38 months

Lead contractor for this deliverable: UDO
Revision: V.1.0 (29th July 2010)

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

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Document Status

<table>
<thead>
<tr>
<th>Document Status</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable Lead</td>
<td>Costas Kotsokalis (UDO)</td>
</tr>
<tr>
<td>Reviewer 1</td>
<td>John Kennedy (INTEL)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Davide Lorenzoli (CITY)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Wolfgang Theilmann (SAP)</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>22/06/2010 (Chapter 8 update on 05/07/2010)</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>06/07/2010 (final on 07/07/2010)</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>07/07/2010 (final on 09/07/2010)</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>09/07/2010</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>29/07/2010</td>
</tr>
</tbody>
</table>

Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDO</td>
<td>Costas Kotsokalis (editor); Peter Chronz; Kuan Lu; Miguel Rojas; Edwin Yaqub</td>
</tr>
<tr>
<td>ENG</td>
<td>Keven Kearney; Francesco Torelli</td>
</tr>
<tr>
<td>TID</td>
<td>Beatriz Fuentes; Alfonso Castro Escudero</td>
</tr>
<tr>
<td>CITY</td>
<td>Theocharis Tsigritis; Howard Foster; George Spanoudakis</td>
</tr>
<tr>
<td>FBK</td>
<td>Farhana Zulkernine</td>
</tr>
<tr>
<td>SAP</td>
<td>Ulrich Winkler</td>
</tr>
</tbody>
</table>

Notices

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

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

This work is licensed under a Creative Commons Attribution 3.0 License.
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>30/05/2010</td>
<td>(As in contributors’ list)</td>
<td>Initial draft</td>
</tr>
<tr>
<td>0.2</td>
<td>17/06/2010</td>
<td>(As in contributors’ list)</td>
<td>Added sections on monitoring and provisioning/adjustment</td>
</tr>
<tr>
<td>0.3</td>
<td>22/06/2010</td>
<td>(As in contributors’ list)</td>
<td>Pre-final editing, executive summary and conclusions / future work</td>
</tr>
<tr>
<td>0.8</td>
<td>05/07/2010</td>
<td>(As in contributors’ list)</td>
<td>Integrated/edited monitoring chapter update</td>
</tr>
<tr>
<td>0.9</td>
<td>09/07/2010</td>
<td>(As in contributors’ list)</td>
<td>Updated according to internal review</td>
</tr>
<tr>
<td>1.0</td>
<td>29/07/2010</td>
<td>(As in contributors’ list)</td>
<td>Integrated PMT review comments</td>
</tr>
</tbody>
</table>
Executive Summary

This document outlines the main achievements of SLA@SOI Work Package A5 (“SLA Management and Foundations”) during the 2nd project year (“Y2”). In order to keep the document manageable from a size and complexity point of view, there has been an effort to present work in the form of high-level technical summaries.

After the introductory chapters 1-3, where the main innovations and the architectural context are discussed, the document continues with the presentation of modelling concepts. This is followed by a discussion on the management of templates (also for service discovery), templates being the starting point for the establishment of a SLA, by means of negotiating according to the proposed free variables and the respective constraints. Topics related to negotiation mechanics and protocols follow next, and the discussion continues with the intelligence that drives the negotiation process – that is, our work on planning/optimization. After the establishment of a SLA, it must be monitored, and corrective actions must take place during its runtime should something go wrong. Thus, the following two chapters address exactly these topics. The document’s last chapter concerns the first implementation of a graphical management console that supports framework administrators. Conclusions and appendices follow, closing the deliverable.

Overall, the Work Package made significant progress over this year, integrating with the rest of the project’s work following the very important architectural update. We have succeeded in extending on Y1 results consistently, towards implementing the SLA Manager – the framework’s cornerstone for SLA management throughout their complete lifecycle. Interoperability, multi-round negotiation, intelligent planning and adjustment, are only some of the achievements during Y2. In upcoming months and during Y3 we will continue on the same track, building on the new architecture and the completed integration. We wish to focus on the production of further theoretical results and novelties of high scientific and practical value, for successful multi-layer SLA management.
# Table of Contents

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

2 Contributions Overview ........................................................................................ 12
  2.1 Key Innovations.................................................................................................. 12
  2.2 Framework Contributions ................................................................................. 14
  2.3 Task Level Activities ......................................................................................... 15

3 Architecture .............................................................................................................. 17

4 Modelling .................................................................................................................. 20
  4.1 SLA Model High-Level Overview ................................................................... 20
     4.1.1 Objectives & Approach ............................................................................ 20
     4.1.2 Abstract SLA(T) Syntax .......................................................................... 21
  4.2 SLA Model BNF Syntax ................................................................................... 30
  4.3 Aspect-Oriented Modelling for SLA-based Service Provisioning ................. 34
     4.3.1 Problem Description .............................................................................. 34
     4.3.2 Extended Open Reference Case (ORC) .................................................... 34
     4.3.3 Research Objective ................................................................................ 36
     4.3.4 The SODA Approach ............................................................................ 36

5 Template Management ............................................................................................. 39
  5.1 Template Registry ............................................................................................. 39
     5.1.1 SLAT Registration, Persistence & Retrieval .......................................... 40
     5.1.2 SLA Template Registry Queries ............................................................. 42
     5.1.3 SLAT Validation & Domain-Specific Term Extensions ....................... 43
  5.2 Advertisements System ....................................................................................... 44
     5.2.1 Apache ActiveMQ ................................................................................... 44
     5.2.2 Features of Advertisements System ...................................................... 45
     5.2.3 Advertisements System Brokers ............................................................. 46
     5.2.4 Channels .................................................................................................. 46
     5.2.5 Plug-in based Client .............................................................................. 46
     5.2.6 Functional Interfaces .............................................................................. 47
     5.2.7 Authorization ........................................................................................... 48

6 Negotiation ................................................................................................................ 49
  6.1 ProtocolEngine .................................................................................................. 49
     6.1.1 Design ...................................................................................................... 49
     6.1.2 Interface .................................................................................................... 52
  6.2 Generic Protocol ................................................................................................. 53
     6.2.1 Negotiation Models ................................................................................. 53
     6.2.2 Design ...................................................................................................... 53
     6.2.3 Code Snippets ......................................................................................... 54
  6.3 Syntax-Conversion for Interoperability .............................................................. 55
     6.3.1 Purpose ..................................................................................................... 56
     6.3.2 Design ...................................................................................................... 57
     6.3.3 Status of Implementation ...................................................................... 58

7 Planning ....................................................................................................................... 59
  7.1 Software Planning and Optimization Component ............................................ 59
     7.1.1 Design ...................................................................................................... 59
     7.1.2 Software Service Manager and Service Evaluation .............................. 59
     7.1.3 Plan and Optimize ................................................................................... 61
  7.2 Infrastructure Planning and Optimization Component ..................................... 62
     7.2.1 Functionality ............................................................................................ 63
     7.2.2 Architecture and implementation ........................................................... 64
  7.3 Formalization of SLA translation ........................................................................ 72
     7.3.1 Service Dependency Graph and Property Dependency Graph ............ 72
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GSLAM architecture</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Publish/Subscribe advertisements bus</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Main negotiation cycle (SLAM components only)</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Conceptual, Abstract &amp; Concrete</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Relations Between SLA(T) Modelling Levels</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Top-Level SLA(T) Classes</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>SLA(T) Party Classes</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Interface Declarations</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>SLA(T) Agreement Terms</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>Guaranteed States</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>Guaranteed Actions</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>The ORC scenario for multiple SLAs</td>
<td>35</td>
</tr>
<tr>
<td>13</td>
<td>Declaration of agreement terms in a SLA</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Mapping table for SLA to SLO and KPIs</td>
<td>36</td>
</tr>
<tr>
<td>15</td>
<td>Service decomposition diagram [8]</td>
<td>37</td>
</tr>
<tr>
<td>16</td>
<td>Services and aspects for the ORC scenario</td>
<td>37</td>
</tr>
<tr>
<td>17</td>
<td>SLA Template Registry Interface</td>
<td>40</td>
</tr>
<tr>
<td>18</td>
<td>SLA Template Metadata</td>
<td>41</td>
</tr>
<tr>
<td>19</td>
<td>QoS-Based Query Results</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>Advertisements System</td>
<td>46</td>
</tr>
<tr>
<td>21</td>
<td>Publish/Subscribe Client</td>
<td>47</td>
</tr>
<tr>
<td>22</td>
<td>Interfaces of Advertisements Service</td>
<td>47</td>
</tr>
<tr>
<td>23</td>
<td>A Bird Eye View of Protocol Engine</td>
<td>50</td>
</tr>
<tr>
<td>24</td>
<td>Tiers of Protocol Engine</td>
<td>50</td>
</tr>
<tr>
<td>25</td>
<td>Component Diagram of Protocol Engine</td>
<td>51</td>
</tr>
<tr>
<td>26</td>
<td>A General Purpose State Machine</td>
<td>53</td>
</tr>
<tr>
<td>27</td>
<td>Interactions of SyntaxConverter with other GSLAM components</td>
<td>57</td>
</tr>
<tr>
<td>28</td>
<td>SWPOC negotiating a Service with no Dependency</td>
<td>60</td>
</tr>
<tr>
<td>29</td>
<td>SWPOC negotiating a Service with multiple Dependencies</td>
<td>61</td>
</tr>
<tr>
<td>30</td>
<td>Simple dependency graph</td>
<td>62</td>
</tr>
<tr>
<td>31</td>
<td>Sequence diagram of negotiation phase</td>
<td>63</td>
</tr>
<tr>
<td>32</td>
<td>Sequence diagram for provisioning request</td>
<td>64</td>
</tr>
<tr>
<td>33</td>
<td>Component diagram of Infrastructure SLAM</td>
<td>65</td>
</tr>
<tr>
<td>34</td>
<td>Detailed IPOC diagram</td>
<td>66</td>
</tr>
<tr>
<td>35</td>
<td>Multi-domain resource provisioning</td>
<td>66</td>
</tr>
<tr>
<td>36</td>
<td>Final profit</td>
<td>67</td>
</tr>
<tr>
<td>37</td>
<td>Failure rate</td>
<td>68</td>
</tr>
</tbody>
</table>
Figure 38: Problem definition ................................................................. 68
Figure 39: Price per unit and failure rates for subcontractors .................. 69
Figure 40: UML class diagram of IPOC’s algorithm ................................. 69
Figure 41: Experimental results .............................................................. 71
Figure 42: Example Service Dependency Graph ...................................... 73
Figure 43: SDG (left) and PDG (right) ....................................................... 74
Figure 44: Negotiation and Translation process ....................................... 75
Figure 45: Overview of SLA@SOI’s monitoring system ............................ 78
Figure 46: Overall monitoring specification process ................................. 79
Figure 47: EVEREST RCG Translator using EC-aware FTL templates ....... 82
Figure 48: AST for agreement term at1 .................................................... 84
Figure 49: Provisioning and Adjustment sequence diagram ...................... 91
Figure 50: PAC Class Diagram ............................................................... 93
Figure 51: Structure of Eclipse RCP application ..................................... 98
Figure 52: SLA Query Configuration ...................................................... 99
Figure 53: SLA list global information view ........................................... 100
Figure 54: Detailed SLA information view ............................................ 101
1 Introduction

The present deliverable presents the achievements of SLA@SOI project’s Work Package A5 – “SLA Management & Foundations”.

In the first year of the project, we explored the area of SLA management at large, and set a solid footing for the rest of the project – mostly by means of the conceptual SLA model developed. Our progress during that time was reported in the first incarnation of this deliverable, submitted on Project Month 12 (M12).

This document is an update that elaborates on all changes and additions to provide the complete picture of the progress and current state of WP A5. Taking into account the updated project architecture, to the formation of which WP A5 was instrumental, we are now in a position to have a working version of our SLA management stack in the form of the SLA Manager (SLAM) and its various incarnations. Within the document, we will briefly discuss the SLAM architecture, and how it relates to the other architectural artefacts. Then we will analyse the functionality of its parts, and illustrate the coherent whole that they constitute. In general, we make an effort to keep the document more illustrative of the conceptual design and its functionality after implementation, than dive into elaborate details of purely technical nature.

As within the first project year (Y1), we will rely to some extent on the essential concepts of the Open Reference Case (ORC) to showcase our work. As such, the document is related to deliverables from WP B2. The architectural discussion of Section 3 must be read in conjunction with deliverables from WP A1. Since WP A5 brings together the work of the other A-line WPs (A2-A4, A6), familiarity with the respective deliverables would benefit the reader. Nevertheless, this is not considered as a prerequisite, and we have made an effort to keep the document as self-contained as possible.

The deliverable is organized as follows:

• Section 2 provides specific information on the main innovations achieved so far within the project. It then discusses contributions to the framework, and finally links the above to task-level activities as described in the Description of Work (DoW).

• Section 3 goes into more architectural detail, illustrating the internals of the SLA Manager. The following chapters are conceptually organized around the various parts of the SLAM.

• Section 4 looks into issues of modelling work. We briefly review the conceptual SLA model, and provide more concrete details on the abstract SLA syntax. Then, discuss issues related to aspect-oriented modelling within, and its applicability to our work.

• Section 5 concerns the management of SLA templates. It provides insight to the SLA Template (SLAT) registry, and the publish/subscribe mechanism for advertising templates.

• Section 6 focuses on negotiation issues. More specifically, it illustrates how the Protocol Engine (PE) works, and the generic negotiation protocol that it implements. This discussion is complemented by the Syntax Converter (SC) description, which is used for over-the-wire communications and interoperability.

• Section 7 elaborates on the planning facilities inside the Planning and Optimization Component (POC), within the software and the infrastructure.
SLAMs. The discussion is accompanied by a formalization of the SLA translation problem, as presented in a peer-reviewed publication.

- Section 8 focuses on the facilities to implement the translation process, from the SLA requirements to the definition of the framework required to be deployed for monitoring the SLA. The Event-Calculus-based EVEREST framework is used for this purpose.

- Section 9 is about the Provisioning and Adjustment Component (PAC), that is responsible for the very deployment of the SLA (including all service artefacts and monitoring infrastructure), as well as the SLA’s adjustment when it is violated or near-violated. Adjustment may refer to reconfiguration of the service (re-provisioning), or triggering a re-negotiation.

- Section 10 describes the SLA Management Console, which visualizes information about executing and past SLAs and allows administrators to manage their SLA portfolios.

Finally, Section 11 concludes the document with a summary of the discussion, and the outlook to next steps. At the end, the appendices provide more technical detail for the interested reader, on things such as the SLA model.
2 Contributions Overview

2.1 Key Innovations

WP A5 is looking into the area of SLA management at large, addressing issues such as SLA modelling, standardization and implementation of automated SLA negotiation, SLA outsourcing and necessary conversion ("translation") facilities, evaluation of SLA utility, etc. In this section we briefly discuss the main innovations achieved as part of this work package.

SLA Model

Our work in the first project year has continued during Y2, extending and adapting the model according to further input from the use cases as they tried to apply it and model their SLA templates and SLAs. We prepared an abstract syntax, that is, a syntax which is independent of the final rendering in a specific language, that implements the conceptual model from Y1. To the best of our knowledge, there exists no prior art in such detail and completeness, while previous efforts for SLA modelling are always bound to specific technologies – usually XML. Conversely, with our abstract syntax we have already achieved a Java rendering of the model, a custom XML rendering, and also an XML rendering that can be embedded into WS-Agreement compatible documents.

Modeling SLAs with Binary Decision Diagrams

As part of the research for SLA planning, we have explored the possibility of using (Reduced Ordered) Binary Decision Diagrams (BDDs), as a SLA representation. BDDs are a graph—based data structure well known in the area of VLSI design, where they are being used for two decades as the main method of model verification. By modelling SLAs as a set of facts, conditions and propositions, which always evaluate to True or False, we can generically express SLAs of any kind and perform domain-independent decision-making as regards the likelihood of a SLA to be fulfilled or violated.

BDDs are helpful for SLA planning, as they provide a minimal, canonical form of the SLAs. Therefore, they allow to find out easily whether a SLA can be fulfilled at all, given some optimal/ideal conditions. Making this decision on complete SLAs is computationally challenging. BDDs also allow comparisons of SLAs; this task is normally an instance of a very difficult combinatorial problem (the graph isomorphism problem). With BDDs, it is possible to perform these comparisons (and subsequently make decisions for outsourcing part of the service) in linear time.

This work has been designed and already presented in an international conference; however, it has not been integrated in Y2 and is scheduled for Y3. As such, in this document it is not further described. In the Y3 update of this deliverable, the full methodology will be provided alongside descriptions of integration details.

SLA-Template-based Service Discovery

In SLA@SOI, we assume that the use of a service is always regulated by an (implicit or explicit) SLA between service provider & customer. This SLA contains both a functional description of the offered service, together with various non-functional terms, including:

- Quality of service (QoS) parameters;
• Behaviours that the customer or provider contractually guarantee to the other party;
• Penalties to pay in case of infringements;
• Monitoring policies relating to the detection of infringements.

In this context, the concept of service discovery takes on added significance. In the traditional view, a customer typically searches for a service endpoint providing specific functional properties. In an SLA context, however, the customer searches for a provider able to supply a specific kind of service and also willing to make certain guarantees and agree to certain 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.

One of the key innovations in SLA@SOI is the development of a fully queriable SLA Template Registry, including the specification of a suitable SLA Template query language and the development of sophisticated query resolution algorithms. The registry is already available in a form where metadata-based searches are possible, and we are currently continuing our work to achieve full template-based queries.

**SLA Negotiation**

Currently, with WS-Agreement being widely used, most SLA implementations use the "take-it-or-leave-it" approach, where an offer can either be accepted or rejected without further modifications. Although specific extensions to this approach exist, no generic and configurable solution is currently available. Task A5.4 worked together with A2 to evolve the existing scientific negotiation proposals into a standardised generic protocol plus use case-specific extensions. As a result, we have built a generic, rule-based negotiation engine that is independent of the exact protocol being used. The rules can be modified and driven by business requirements, while the protocol used on-the-wire can be different as the engine is combined with custom syntax converters that address this requirement. For instance, there is already a syntax converter for WS-Agreement, and one for a simpler XML (project-specific) protocol.

**Infrastructure SLA Planning**

In this work, which is ongoing in collaboration with WP A4, we have designed and are currently implementing methods for a number of issues related to infrastructure SLAs; namely suitable in-memory representations, outsourcing parts of SLA requests, price quotations given a proper cost model, and the problem of cases when resources become scarce – such as, for example, when some part of the data centre fails. In this context we have developed a SLA representation based on Binary Decision Diagrams; a subcontractor selection method based on price and reputation; a model that calculates service costs while taking into account failure probability and profit requirements; and a knapsack-problem instance that tries to minimize the overall penalty when some of the established SLAs cannot be honoured due to lack of enough resources.

**SLA Translation**

Within task TA5.3 we have looked into the topic of converting SLAs among different levels, to both enable subcontracting, and respond to the requirements posed by service hierarchies. We currently have a solid definition in the form of a mathematical model, and a formalization of the relevant conversion process. The complete implementation of a data model that can be used for this purpose is ongoing work to be completed in Y3.

**Aspect Oriented SLA Modelling and Monitoring**
Guarantee terms in SLAs define the service level objectives (SLOs) for a particular service: the functional and QoS properties of the service “as agreed” that the service “as delivered” should guarantee. These QoS and functional properties should be monitored during service delivery in order to check potential SLA violations.

However, these properties are concerns that depend on cross—layer characteristics (service/software/infrastructure deployment and monitoring instrumentation) of the system.

We intend to investigate the use of Aspect Oriented Design techniques to model different concerns of SLA (e.g. SLA at the business, service or infrastructure level, as well as cross-cutting concerns in SLA). Moreover, we will investigate the use of aspect—oriented techniques for monitoring cross—cutting concerns to allow the separation of monitoring tasks at different levels of abstraction.

An internal deliverable document has been prepared, which defines the problem and the research goals with respect to the ORC use case scenario and the SLA template defined for the SLA@SOI framework. The document also presents an elaborate state—of—the—art of the existing Aspect—Orientation approaches. The next phase of the research will focus on developing a prototype using the existing Service Oriented Design using Aspects (SODA) approach proposed by Kim et al. Unlike other existing approaches, the SODA approach proposes a generic model for a SOA that is independent of the deployment framework. We will, thereby, enhance the model to overcome its limitations with regards to flexible SLA—based system modelling and deployment in a multi—layer service provisioning and monitoring infrastructure.

2.2 Framework Contributions

WP A5 is addressing the part relevant to the Generic SLA Manager (GSLAM), and some specific instances of it. Deliverable D.A1a ("Framework architecture - full lifecycle") illustrates how the (G)SLAM fits the complete framework. Section 3 provides more details in the internal architecture of this component.

On a functional level, this WP contributed to the framework with the following functionalities:

- **SLA & SLA Template management**: Chapters 5 and 10 discuss template management (the SLAT registry and the advertisements / discovery system), and the SLA management console respectively.
- **SLA Negotiation**: Chapter 6 provides the details on the SLAM’s negotiator, consisting of the syntax converter and the protocol engine;
- **SLA Planning and Optimization facilities for certain problem instances**, as examples of how to use these domain—specific parts of the SLAM architecture. In Chapter 7 we discuss this topic in more detail.
- **Deduction of suitable SLA Monitoring Framework instances**, based on the specifics of each analyzed SLA. This facility, and the process based on which monitoring framework instances are produced from SLAs, are described in Chapter 8.
- **Orchestration of service provisioning, and initiation or execution of adjustment actions when the SLA is violated or in danger of being violated. To some extent, this is a domain—specific process, and our contributions are meant to offer generic methods but also reference implementations for these tasks. We discuss this topic in Chapter 9.**
2.3 Task Level Activities

The tasks of WP A5 largely correspond, to the organization of our work and the contributions as presented so far. In the following table, we provide the mapping between WP tasks, and chapter or section in the upcoming text.

Table 1: Mapping of WP tasks and document sections

<table>
<thead>
<tr>
<th>WP A5 Task</th>
<th>Document Chapter/Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA5.1 - SLA Modelling</td>
<td>4 - Modelling</td>
</tr>
<tr>
<td>TA5.2 - Service/SLA template registry and discovery</td>
<td>5 - Template Management</td>
</tr>
<tr>
<td>TA5.3 - SLA Translation</td>
<td>7.3 - Formalization of SLA translation</td>
</tr>
<tr>
<td>TA5.4 - SLA Negotiation</td>
<td>6 - Negotiation</td>
</tr>
<tr>
<td>TA5.5 - SLA Planning and Optimization</td>
<td>7.1 - Software and 7.2 - Infrastructure Planning and Optimization Component</td>
</tr>
<tr>
<td>TA5.6 - SLA Monitoring</td>
<td>8 - Translation of SLAs into Monitoring Specifications</td>
</tr>
<tr>
<td>TA5.7 - SLA Provisioning &amp; Adjustment</td>
<td>9 - Provisioning and Adjustment</td>
</tr>
<tr>
<td>TA5.8 – SLA Interface Access Layer</td>
<td>10 - SLA Management Console</td>
</tr>
</tbody>
</table>

In all tasks there was progress made throughout the second project year:

- The SLA model was advanced from a conceptual representation to an abstract syntax, allowing building different renderings, all compatible to the initial model.
- The template registry was integrated with the rest of the architecture, complemented by a system for advertising templates and receiving such advertisements among different SLA managers.
- With regard to TA5.3, we devised a formal definition of the SLA translation problem, which can be applied to different domains as part of the custom Planning/Optimization Component (POC) of the generic SLA manager.
- As regards the negotiation task, we built a generic, customizable protocol engine that responds to negotiation messages and orchestrates different protocols. Separating the protocol implementation (accomplished by the protocol engine) from the domain-specific intelligence (implemented by the POC) allows having a generic part in the SLA manager that can be reused independent of the specific use case.
- The POC, part of task TA5.5, is now a stateless component that can operate on concepts of utility and not bother with the mechanics of negotiation protocols. In this task we implemented various methods for infrastructure planning; and connected to the Software Service Evaluation from WP A6 so that we can receive and evaluate options for software services.
- In Task TA5.6, implemented within the SLAM’s Monitoring Manager, the core task was to compute monitoring framework instances and reasoning component configurations for specific SLAs; these are then fed into a process of “translation” into the final monitoring specifications. In this context, the work from Y1 was significantly extended, also to comply with
the new SLA model (in Y1, WS-Agreement was used throughout the project components).

- Task 5.7 had some progress in Y1 (although initially scheduled to start in Y2); during the second project year, its results were better integrated in the architecture and generic concepts of SLA adjustment were introduced.
- Task 5.8 started in Y2, so its overall progress took place within this year.
3 Architecture

In this section, the document provides a brief discussion of the A5-specific architecture, placing it in the context of the overall (A1) architecture and elaborating to some extent on the specifics.

The main outcome of A5 is the Generic SLA Manager (GSLAM), illustrated in Figure 1. Its components are the Template Registry, the Protocol Engine, the Service Manager Registry (to be developed in Y3), the SLA Registry, the Monitoring Manager, and the Advertisements Bus. These are generic components, available to users without any required customization.

As evident in Figure 1, the GSLAM also includes a component for Planning and Optimization (POC) and one for Provisioning and Adjustment (PAC). These two components are considered use-case specific (although the PAC includes some reusable parts as well). As such, there are no generic implementations provided; they are shells that must be filled in, via the implementation of their pre-defined generic interfaces. The reason for this choice has to do with the highly use-case-specific process of creating a deployment plan and optimizing it, depending on the provider’s business strategies and priorities, as well as technical constraints. Furthermore, theory says that there is no single optimization method that outperforms all others in all domains [1].

Conceptually, the starting point for using a SLA Manager is the publication of SLA Templates (cf. Section 4.1). As soon as a new template becomes available in the template registry, it is converted to the formats supported by the various Syntax Converters (SC) (currently there exist reference implementations for WS-
Agreement and for an XML representation of the A5 model). Then, it is published in this new form, for other SLAM implementations to collect and store locally in the "provider-facing" side of the template registry (illustrated in Figure 2 as "ad-cache").

**Figure 2: Publish/Subscribe advertisements bus**

Having templates stored, both the ones that are "local" to a SLAM, and the ones that are "remote" (provider-facing), it is possible for SLAMs to become full members of the SLA management ecosystem and negotiate with each other for services. By design, a SLAM can be a customer of other SLAMs, a server to other SLAMs, or both. This autonomy is explored in the framework, having a Business SLAM (BSLAM), a Software SLAM (SSSLAM) and an Infrastructure SLAM (ISLAM), implementing a stack of SLA managers that communicate mainly for (re)negotiation purposes. The objective of the negotiation process is to converge on a SLA (a contract) that governs future interactions in the context of service consumption. The main negotiation cycle is illustrated in simplified form in Figure 3 (including only the interactions among SLAM components).

**Figure 3: Main negotiation cycle (SLAM components only)**

Negotiation messages are initially converted to in-memory representations by the appropriate syntax converter, and passed on to the Protocol Engine (PE). The latter validates the message with respect to template constraints, and protocol
implementation. Compliance to templates is confirmed via querying the SLA Template Registry\(^1\) (SLATR). As an example of protocol-related validation, one can imagine the case where a SLAM receives a message of final agreement to a negotiation. Each ongoing negotiation has an ID, that the PE keeps track of it. If the received (final) agreement message contains an unknown ID, it must apparently be rejected. Such negotiation mechanics are taken care of by the PE.

If protocols and templates are respected, the message is passed on to the SLAM’s POC. This component carries the intelligence/knowledge to evaluate an incoming request, and produce outgoing requests for SLAs. All this is supported by the Service Evaluation component if it is available for the use case, as well as certain information received by the service manager (e.g. about resource capacity and availability). As part of this planning process, a POC also consults with the Monitoring Manager (MM) to confirm the capability to monitor a SLA (if it is eventually established) \(^2\). If a proposed SLA is feasible and satisfactory, it is agreed upon and stored locally (for each participating SLAM) in the SLA Registry (SLAR).

The response from the Monitoring Manager, as mentioned above, is a monitoring system configuration that is also used as an additional part of the provisioning process. The latter is taken care of by the PAC, which receives it alongside a service implementation plan produced by the POC. The service implementation plan is the main output of the POC, to be used locally. It describes activities and their timing. The PAC then includes both generic facilities to process plans, and use-case-specific knowledge to process the included activities. Using the latter, the PAC can orchestrate complex provisioning tasks, where for instance the instantiation of one single service requires the deployment, configuration and initialization of numerous software components (database, application server, etc).

Except for negotiation and provisioning, the SLAM also participates in the adjustment process, that is, the tasks that must take place when a violation occurs or is about to occur. More specifically, the PAC receives monitoring information that signals such an event, and, based on the current state, it proposes an action. Should a new plan, or even a complete renegotiation of the agreement, be required for this adjustment action, the POC will be asked to take the respective necessary actions, following which re-provisioning of the service is initiated with new parameters to address the new (updated) SLA.

\(^1\) All components can interact with all registries, although not shown in Figure 1 for reasons of presentation.
4 Modelling

4.1 SLA Model High-Level Overview

This section outlines year 2 progress on the SLA & SLA Template Model - henceforth SLA(T) Model. In year 1 the focus was on requirements gathering & conceptual modelling, that is; on abstracting the necessary & relevant distinctions & concepts underlying SLA(T)s. In year 2, working from this conceptual foundation, the goal has been to first develop an abstract syntax for SLA(T)s, and from this to derive various concrete representations - in particular; a Java API, XML Schema and BNF Grammar. The relations between the different modelling levels are shown in Figure 4.

The following subsections briefly outline the objectives & approach taken in developing the abstract syntax, and provide an overview of the result. To illustrate the concrete instantiation of the syntax, Section 4.2 presents a specification of the BNF Grammar. We assume the reader is familiar with the basic notions of SLA and of SLA Template.

4.1.1 Objectives & Approach

The primary objectives in developing the abstract SLA(T) syntax were:

i) to support the varied requirements of B-Line use-cases - notably; supporting SLAs for non-IT services (cf. B6 'e-Government').

ii) to ensure that SLA(T) information content is sufficiently well-specified to support A-Line solutions & scientific innovations (e.g. concerning automated approaches to SLA negotiation, evaluation & monitoring).

iii) to support, where applicable, integration with existing standards (e.g. for web-services; WS-Agreement & WSDL).

iv) To facilitate exploitation beyond the immediate scope of the project. In particular, as contribution to the reference architecture developed as part of the NEXOF-RA project.

The basic approach to meeting these primary objectives was conceived in terms of the following technical objectives:

• that the model should be developed in a form which is transparent to any particular exchange syntax or programming language, serving, effectively, as a reference model whose concrete representation can be adapted to specific technical & problem domains.
• that the model should have a well-defined, complete and unambiguous semantics. This is critical for the following key reasons:
  
  - SLAs constitute formal agreements, or contracts, between providers & customers. Given the potential legal implications of SLAs, both providers & customers need to be clear on what precisely they are agreeing to.
  
  - an SLA which cannot be monitored for violations is not an effective SLA. Hence, SLA content must encode precise criteria for the detection of violations (in a form accessible to monitoring configuration systems).
  
  - to support negotiation, planning & QoS-based service discovery (discussed elsewhere in this document), it must be possible to match the QoS requirements of customers (e.g. SLA proposals, or QoS-based search queries) against advertised & actual provider capabilities. As described in Chapter 5 (SLA Template Management), the matching of QoS properties necessarily entails semantic comparisons.

• the model must support open-ended & domain-specific definitions of QoS terms - i.e. it should be customisable.

Ostensibly, these objectives present two broadly conflicting constraints. On one hand, generality and rich expressivity are required to ensure applicability to a wide range of domain-specific contingencies. While on the other, the strong semantic requirements imply high specificity & tight control of SLA(T) content. The abstract SLA(T) syntax presented here provides a means to accommodate these opposing objectives; loosely, it serves as a high-level language, tightly constraining the form of the various expressions that comprise an SLA, but placing only minimal constraints on the specific terms employed in these expressions. The expressions have a recursive structure permitting a high-degree of expressivity, but are sufficiently constrained to support (a reasonable degree of certainty in) QoS-based matching algorithms. The individual terms (or tokens) employed in the expressions are specified separately in the form of "pluggable" domain-specific vocabularies (or ontologies) - whose semantics can be defined to the degree of precision required by the domain.

The next section provides an overview of the abstract syntax.

4.1.2 Abstract SLA(T) Syntax

The abstract syntax is constructed in hierarchical fashion as schematised in Figure 5, and described below.
The lower levels of the hierarchy specify:

i) **data primitives**,  
ii) **ground expressions**: encapsulating **constraints, events & functions**, and  
iii) **service descriptions**: encapsulating the functional properties of services (e.g. interfaces & operations).

These lower-level modelling elements are not specific to SLAs, but rather serve as a common information model for the project as a whole. Their specification extends the Core Model work carried out in A1 in year 1, and for this reason, we present here only a high-level summary of the lower-levels of the hierarchy. A more complete account is provided as part of the A1 ‘Framework Architecture’ deliverable (Section 5.2).

The abstract SLA(T) syntax proper builds on this common base to define the information content of SLAs and SLA Templates; encapsulating the non-functional (QoS) descriptions of services, and formalising the notion of a **QoS Guarantee**.

At each level the model is **abstract**, in that it only specifies coarse grained syntax & semantics, without committing to any specific syntactic tokens, or to any specific interpretations of these tokens. As stated earlier, specific tokens are instead defined in a separate vocabulary. The tokens employed within SLA@SOI are referred to as the ‘Standard Terms’.

Finally, business-related aspects of SLAs – that is; terms relating to service **costs, penalties, exclusion clauses, termination conditions**, etc - are modelled at an additional hierarchical level, referred to as the ‘Business Product’ layer, which provides a complete business-oriented solution. These business aspects are developed as part of A2 and are not covered here.

The following sections provide brief summaries of the models at these different layers. At each layer the model is formally specified in terms of UML class diagrams.

**Common (Low-Level) Entities**

**Primitives**
The lowest level of the model hierarchy provides a classification of the data-type primitives assumed by subsequent layers. The key primitives with respect to the abstract SLA(T) syntax are listed below:

- **TEXT** an uninterpreted character string,
- **ValueExpr** a generic super-type of all expressions which can be used as function parameters (described shortly), including: all the primitives listed below, functions and events.
- **ID** an identifier with local scope (i.e. which is not guaranteed to be universally unique),
- **UUID** a universally unique identifier,
- **STND** a universally unique identifier which specifically denotes a *standard vocabulary term*. The abstract SLA(T) syntax is customised by defining domain-specific standard vocabulary terms.
- **CONST** an explicitly typed constant (or literal) value
- **TIME** a value denoting a specific point in time.

Building on the primitives, 4 types of ground expression are then defined:

### Annotations

Defined as simple key/value pairs. Annotations are defined at a low-level so as to be available if required, at all superior layers. Annotations are ignored (uninterpreted) by domain-independent mechanisms in the framework.

### Functions

A *generic* class of expressions which have functional form and semantics\(^2\) – i.e. formally; an *operator* (the function’s name), and a set of *parameters*:

\[
<\text{operator}> ( <\text{param1}> , <\text{param2}> , ... )
\]

Parameters can be any *ValueExpr*, including functions – thus permitting arbitrary nesting.

The various QoS metrics which may be employed in SLA(T) guarantees are all conceived in functional terms. The metric “service availability”, for example, would be conceived as a function mapping a “service” to a *measure* of that service’s “availability”:

\[
\text{availability(service)} \rightarrow \text{measure-of-availability}
\]

Accordingly, function expressions provide the key mechanism supporting domain-specific customisation. All the standard QoS metrics employed in SLA@SOI, for example, are defined as functions (see Appendix C 'Standard Terms').

### Events

A *generic* class of expressions denoting identifiable *events* – including, for example; time-signals, the raising of an exception (fault) during a service operation, or the violation of an SLA guarantee (among others).

---

\(^2\) Functions are interpreted as a mapping relation from input to output domains - i.e. functions can, in principle, be *evaluated*, and have an *output type*. The output type is not included in the abstract syntax since it is a semantic, rather than syntactic property (e.g. it does not explicitly appear as part of the SLA(T) content). A formal specification of both input & output types, however, *is* required in order to support SLA(T) validation (see Standard Terms section below).
Structurally, event expressions are identical to function expressions, in that they comprise a name (identifying a class of events), together with a set of parameters (constraining the event class):

\[ \langle \text{event-class} \rangle \ [ \langle \text{param1} \rangle, \langle \text{param2} \rangle, \ldots ] \]

The event of a service operation raising a particular exception, E, for example, could be expressed as:

\[ \text{fault}[ E ] \]

Although they are structurally identical, the abstract syntax requires that event expressions be formally distinguishable from functions (indicated here, for example, by the use of square as opposed to curved brackets).

**Constraints**

A generic class of expressions representing constraints and logical combinations of constraints. As with events, constraint expressions are essentially a sub-species of function, distinguished primarily because of the special role they play in expressing SLA(T) guarantees. As functions, all constraints evaluate to boolean (truth) values.

Formally, an atomic constraint is a binary function with the (infix) form:

\[ \langle \text{LHS} \rangle \langle \text{comparison-operator} \rangle \langle \text{RHS} \rangle \]

.. where \( \langle \text{comparison-operator} \rangle \) denotes a specific relation between \( \langle \text{LHS} \rangle \) and \( \langle \text{RHS} \rangle \) value expressions (i.e. expressions of type ValueExpr, such as; identifiers, constants or functions). All of which is just a complicated way to say that constraints include expressions like “1 < 4” or “A is-a-member-of \{A,B,C\}”.

A compound constraint is an \( n \)-ary function of the form:

\[ \langle \text{logical-operator} \rangle ( \langle \text{constraint1} \rangle, \langle \text{constraint2} \rangle, \ldots ) \]

.. denoting a logical operation (e.g. conjunction, disjunction or negation) over a set of constraints. Examples include; “1 < 4 and 2 > 3” or “A is-a-member-of \{A,B,C\} and not(A == B)”

In terms of SLA(T)s, the principle role of the ground expressions is to represent:

- measured values (i.e. functions, and in particular, QoS metrics),
- constraints on these measured values, and
- the points-in-time at which the measurements take place (i.e. events).

What is missing is a means to express the object of the measurement itself, which is provided by the next layer in the model hierarchy; service descriptions:

**Service Descriptions**

In the conceptual model developed in year 1, the functional properties of services were conceived in terms a service-interface, comprising one or more service-operations (each operation denoting a distinct functional capability of the service). Accordingly, service descriptions in the abstract SLA(T) syntax were initially conceived as a generalisation of the WSDL 2.0 specification\(^3\). In light of subsequent use-case input, however, the abstract syntax now formally distinguishes 3 kinds of service:

\[^3\] WSDL 2.0 specification : http://www.w3.org/TR/wsd120/
i) **message-based-services** (corresponding to the original WSDL 2.0 generalisation): services defined as functional interfaces, in the classic Object-Oriented sense – i.e. with explicitly specified functional capabilities (operations), which can be invoked by end-consumers.

ii) **open-message-based-services**: similar to ‘i’, but with *unspecified* functional capabilities (examples can be found in the B3 ‘ERP Hosting’ use-case). Since no explicit operations are defined for these services, QoS terms can not be defined in terms of the invocation of operations, but must instead relate to the ‘messaging’ properties of the communication medium – e.g. for HTTP-based communications; the receipt of HTTP requests. As such, a description of the communication media, or protocol, is the defining feature of the service description.

iii) **resource-type-services**: relating specifically to *infrastructure resources* (e.g. virtual-machines, database servers, humans, etc.) offered to consumers as ‘services’. The nature, and hence description of these services is entirely domain-specific.

We use the term *interface specification* to denote descriptions of (open-)*message-based-services*[^4], and the term *resource specification* to denote (domain-specific) descriptions of resource-type-services.

The abstract SLA(T) syntax is defined over these common modelling elements, as described in the next section.

**SLA & SLA Templates**

The abstract SLA(T) syntax subdivides the content of an SLA Template into 5 content types (as shown in Figure 6):

i) SLA Template attributes (e.g. unique identifier for the template)

ii) Details of the agreement parties,

iii) Service descriptions; encapsulated as *interface declarations*,

iv) Variable declarations, and

v) Agreement terms; specifying the QoS guarantees.

An SLA is simply an SLA Template with an extended set of attributes specifying the time at which the SLA was agreed, the duration of the agreement (the time period during which it is *in effect*) and a reference to the SLA Template on which the SLA was based.

Variable declarations are provided as a syntactic convenience to improve readability and avoid repetition of content; they simply permit the assignment of an *expression* to a *variable*, which variable can then be used as a stand-in[^5] for that expression in subsequent expressions. As such, they will not be discussed further here (see wiki documentation[^6] for full details). The remaining content types are described in the following paragraphs.

[^4]: An *open-message-based-service* is described by an interface-specification with zero operations

[^5]: i.e. similar to the use of macros in programming languages

[^6]: /Workpackages/Action_Line_A/A5_SLA_Management_%26_Foundations/TA5.1_SLA_Model
Agreement Parties

At the level of the abstract SLA(T) syntax, it is sufficient that agreement parties are simply identified and assigned a role in the agreement (e.g. as provider, customer, or some other, domain-specific role). Depending on the use-case and the kinds of guarantees on offer, it can also be useful to define party sub-categories - termed “operatives” – e.g. in order to distinguish different classes of service consumer, or provider agents\(^7\). As with parties, it is sufficient just to identify these operatives. If required, both party & operative content can be annotated with additional information (e.g. names, addresses, contact info. etc).

The abstract SLA(T) syntax for parties & operatives is shown in Figure 7.

---

\(^7\) Provider “agents” are the entities responsible for actually delivering (i.e. implementing or executing) the service (see year 1 conceptual model).
**Interface Declarations**

Service descriptions are incorporated into SLA(T)s as *interface declarations*, the abstract syntax for which is shown in Figure 8 and explained below.

![Diagram of Interface Declarations]

**Essentially**, an interface declaration is used to assign a local identifier to an interface, where an interface is one of the following (refer to Service Descriptions section above):

- an explicit *interface specification*; describing a functional interface
- a reference to an external *interface specification*, i.e.; whose content can be downloaded from some publicly accessible location,
- a *resource-specification*; describing an infrastructure resource (encapsulated by the class ResourceType in Figure 8)

Each interface declaration must also specify the agreement party responsible for providing the interface. Any and all parties to the agreement, irrespective of their role, can provide interfaces. An SLA might specify a *customer* interface, for example, in order to provide a precise functional specification of customer notifications (callbacks).

Optionally, interface declarations may also be associated with specific endpoints, where the description of an endpoint comprises the endpoint’s location, communication protocol, and local (i.e. SLA(T) specific) identifier.

As with all SLA(T) content, interfaces, interface declarations and endpoints may all be annotated with domain-specific information. Annotations may be used, for example, to keep a record of web-service specific information (e.g. SOAP bindings) when translating from WSDL documents to the more generic (not web-service specific) interface specifications employed by the abstract syntax, thus enabling round-trip translations.

**Agreement Terms**

Agreement terms provide the substantive content of SLA(T)s; providing precise specifications of party obligations in respect of declared services. These obligations are formalised as *guarantees*, which come in two flavours; guaranteed states, and guaranteed actions (described shortly). As shown in Figure 9, each

---

8 the term ‘interface’ is used for historical reasons (see Service Descriptions section)

9 explicit; meaning just that the full content of the interface specification is given as part of the SLA(T) content.
agreement term specifies one or more guarantees, together with an optional precondition (stating the conditions, if any, under which the agreement term holds). All agreement terms and guarantees are identifiable and can be annotated with domain-specific information. The abstract SLA(T) syntax also permits the use of local variables within agreement terms.

Figure 9: SLA(T) Agreement Terms

A guaranteed state, as shown in Figure 10, is essentially a constraint on some measurable (QoS) aspect of a service, e.g. that:

- the completion-time of a particular service operation is always less-than 5ms, or
- the arrival-rate of consumer invocations must be less-than 40 transactions-per-hour,

Guaranteed states may also be associated with additional preconditions and a (domain-specific) indication of “priority”.

Figure 10: Guaranteed States

A guaranteed action (see Figure 11), instead, is an obligation on the part of an agreement party to perform a specific action (beyond the explicit functional capabilities specified for the services they offer). A provider, for example, may be obliged to send periodic reports to customers on the state of an SLA, or to pay a penalty in case of certain violations. In which case, reporting and payment are both guaranteed actions.
Figure 11: Guaranteed Actions

Each guaranteed action must:

- state whether the action is **obligatory, optional or forbidden** (i.e. specify the action **policy**),
- identify the agreement party who must (or may, or must not) perform the action,
- specify the (pre)conditions under which the action must (or may, or must not) be performed, and
- describe the action itself, i.e. specify its **post-condition**.

In the current abstract SLA(T) syntax only two kinds of action (post-condition) have been defined, namely:

- **invocation action**; the action of invoking an interface operation (optionally at a specific endpoint, and with specific predefined parameter values),
- **custom action**; a place-holder for a domain-specific action description.

Business-related actions, such as the payment of a penalty or the renegotiation or termination of an SLA, are defined at the Business Product modelling level (see Figure 5). In year 3, a syntax for more complex, compound actions (e.g. workflows) will also be developed.

**Model Summary**

The preceding sections provided a high-level overview of the abstract SLA(T) syntax. We call this model a **syntax** because it defines **formal** (structural) constraints on the information content of SLA(T)s. We call it **abstract** because it defines these constraints only at a coarse level, without committing to any specific syntactic tokens. To instantiate the abstract syntax requires an enumeration of these specific tokens, and depends on:

i) the **language** chosen to instantiate the abstract syntax. In Y2, for example, we have developed concrete versions in XML, Java and BNF (presented for illustration in section 4.2),

ii) independent of the language, a **vocabulary** of domain-specific terms – e.g. particular terms for ‘comparison-operators’, ‘agreement-roles’, ‘endpoint-protocols’, etc. In respect of the abstract syntax, these
vocabularies effectively enumerate the permitted values of occurrences of the STND primitive.

The “default” vocabulary of domain-specific terms developed & employed within SLA@SOI is referred to as the “Standard Terms” vocabulary (provided for reference as “Appendix C”).

While the abstract SLA(T) syntax per se is ambivalent to the specific terms employed, the various tools and services comprising the SLA@SOI framework are not. In particular, the specific mechanisms & algorithms developed for QoS-based SLA(T) matching and SLA(T) validation assume a small, fixed set of “built-in” functions & operators. The built-in “Standard Terms” include:

- standard logical operators (and, or, not)
- standard comparison operators (<,>,<=>,=, etc)
- basic arithmetic functions (+, -, *, %, etc)
- basic set operations (summation, mean, min/max, etc)
- time-series (temporal sequences of metric values)
- ‘temporal context’ (current-time, day-of-week, etc)

Similarly, for effective SLA(T) validation, all domain-specific terms should be classified according to the role they play in SLA(T) and, for parametric expressions, any required parameters and/or result-types. Thus, for example, the Standard Terms vocabulary (see *Appendix C* for a complete listing) includes the following:

- “sla:mandatory” : assigned to the role of an action-policy in SLA(T).
- “core:completion_time” : assigned to the role of a metric which evaluates to a ‘duration’ and requires a single ‘message-type-service’ identifier as parameter.

The classification scheme for STND terms is formalised as a terms meta-model. The validation mechanisms developed for SLA@SOI require that all domain-specific terms are formally described in terms of this meta-model. Both the validation and QoS-based matching mechanisms are implemented as part of the SLA Template Registry, which is described more fully in Section 5.1.

### 4.2 SLA Model BNF Syntax

The Backus–Naur Form (BNF) grammar defined here is intended just as a semi-formal syntax for SLA(T)s, for human use only. Its purpose is just to provide a common shorthand notation for use in examples or documentation. In particular, the syntax is not designed to be machine parsable, and strict adherence to the grammar is not enforced. Because the goal is ‘semi-formal’, we have also been fairly lax in the definition of the grammar & adopted some non-standard conventions which may be more familiar to programmers. Briefly:

- spacing (white-spaces, tabs, carriage-returns, etc.) are not explicitly specified, but merely indicated by layout
- non-terminals are indicated just with bold text
- terminals are delimited by either:
  - double-quotes, "", if fully specified, or
  - question marks, ?, if only informally or incompletely specified
• plain brackets are used for grouping alternatives, e.g. (A | B) denotes either A or B
• square brackets denote optional elements, as follows
  o [A] = A occurs with multiplicity 0..1
  o [A]* = A occurs with multiplicity 0..*
  o A [A]* = A occurs with multiplicity 1..*
• finally, we adopt C style comments (i.e. ‘/* .. */’ and ‘// ’)

The BNF Syntax for SLAs & SLA Templates is as follows:

```plaintext
/* --- SLAs --- */
sla ::=
  "sla{"
  "agreedAt = " ( TIME | "na" )
  "effectiveFrom = " TIME
  "effectiveUntil = " TIME
  "templateId = " UUID
  sla_template_content
  "}"  

/* --- SLA Templates --- */
sla_template ::=
  "sla_template{"
  sla_template_content
  "}"  
sla_template_content ::=
  "uuid = " UUID
  "sla_model_version = sla_at_soi_sla_model_v1.0"
  agreement_party [ agreement_party ]*
  interface_declr [ interface_declr ]*
  [ variable_declr ]*
  agreement_term [ agreement_term ]*

/* --- Agreement Parties --- */
agreement_party ::=
  "party{"
  "id = " ID
  "role = " agreement_role
  [ operative ]*
  "}"  
agreement_role ::= STND /*see [see Appendix C]*/
operative ::=
  "operative{"
  "id = " ID
  "}"  

/* --- Interface Declarations --- */
interface_declr ::=
  "interface_declr{"
  "id = " ID
  "provider_ref = " provider_ref
  [ endpoint ]*
  [ interface_ref | interface_spec | resource_type ]
  "}"

provider_ref ::= ID | "provider" | "customer"
endpoint ::= "endpoint{"
  "id = " ID
  "location = " UUID | "[custom_location]"
  "protocol = " protocol
  "}"  
protocol ::= STND /*see [see Appendix C]*/
interface_ref ::= "interface_ref( UUID )"
resource_type ::= "resource_type"

?domain-specific-details?
```
/* --- Variable Declarations --- */

variable_declr ::=
  ID "is" ValueExpr |
  "is" domain_expr "<<" CONST ">>"

/* --- Agreement Terms --- */

agreement_term ::=
  "agreement_term{"
  "id = " ID
  [ "precondition(" constraint_expr ")" ]
  [ variable_declr ]*
  guarantee [ guarantee ]*
  "}"

guarantee ::= guaranteed_state | guaranteed_action

/* --- Guaranteed States --- */

guaranteed_state ::=
  "guaranteed_state{"
  "id = " ID
  [ "priority = " CONST ]
  [ "precondition(" constraint_expr ")" ]
  constraint_expr
  "}"

/* --- Guaranteed Actions --- */

guaranteed_action ::=
  "guaranteed_action{"
  "id = " ID
  "actor = " ID
  "policy = " action_policy
  "trigger = " ( EventExpr | ID )
  action_postcondition
  "}"

action_policy ::= STND /*see [see Appendix C]*/

action_postcondition ::= invocation |
  business_action |
  compound_action |
  custom_action

invocation ::=
  "invocation{"
  "endpoint =" ID
  "operation =" ID
  [ invocation_param ]*
  "}"

invocation_param ::=
  "param{"
  "name =" ID
  "value =" ValueExpr
  "}"

business_action ::= ?defined as part of business model?

compound_action ::= ?to be defined in year 3?

custom_action ::= "custom{"
  ?any text?
  "}"

/* --- Interface Specifications --- */

interface_spec ::=
  "interface_spec{"
  [ "name =" ?any-text? ]
  [ extension_list ]
  [ operation ]*
  "}"

extension_list ::=
  "extension_list{"
  ID [ ID ]*
  "}"

operation ::= "operation{"
  "name =" ID
  "}"
```plaintext
[ input ]*
[ output ]*
[ related ]*
[ fault_list ]
"

input ::= "input{" property_content "}"
output ::= "output{" property_content "}"
related ::= "related(" property_content ")"

property_content ::= "name =" ID
        "datatype =" STND
        "domain = (" domain_expr ")"
        "auxiliary =" BOOL

fault_list ::= "fault_list{"
            [ STND ]*
        "}"

/* --- Service References --- */
service_ref ::= "service_ref{" ID [ ID ]* "}"

/* --- Constraint Expressions --- */
constraint_expr ::= ValueExpr domain_expr { "<|>|=/" CONST "|>" }
        | "not(" constraint_expr ")"
        | ["("] constraint_expr ( "and" | "or" )
            constraint_expr ["]"
        |
        CID // version 1.1 only
domain_expr ::= comparison_op ValueExpr
        | "not(" domain_expr ")"
        | ["("] domain_expr ( "and" | "or" ) domain_expr 

comparison_op ::= /*see [see Appendix C]*/
        // or for BNF, the symbols: ==,=,!=,<,<=,>,>=

/* --- Event Expressions --- */
event_expr ::= _op "[" Expr ["," Expr ] "]"
event_op ::= STND /*see [see Appendix C]*/

/* --- Functional Expressions --- */
func_expr ::= func_op "(" ValueExpr [ "," ValueExpr ] ")"
        | ValueExpr func_op ValueExpr // infix alternative
func_op ::= STND | arithmetic_op | set_op | time_series_op
        | context_op | misc_op | qos_term
arithmetic_op ::= /*see [see Appendix C]*/
        // or for BNF, the symbols: +,-,*,/,%
set_op ::= /*see [see Appendix C]*/
time_series_op ::= /*see [see Appendix C]*/
context_op ::= /*see [see Appendix C]*/
misc_op ::= /*see [see Appendix C]*/
qos_term ::= /*see [see Appendix C]*/

/* --- Value Expressions --- */
Expr ::= constraint_expr | domain_expr | ValueExpr
ValueExpr ::= ID | BOOL | CONST | TIME
        | func_expr | event_expr | service_ref
CONST ::= ?the value of the constant? standard_datatype
standard_datatype ::= STND /*see [see Appendix C]*/
PATH ::= ID ":=" ID
CID ::= ID
STND ::= ?any text (no white spaces, no quotes)?
PATH ::= ID
```
### 4.3 Aspect-Oriented Modelling for SLA-based Service Provisioning

Aspect-oriented Modelling (AOM) techniques enable clear separation of crosscutting concerns as aspects, which can be dynamically weaved into the main model as required, and thus, facilitate the design of flexible, maintainable and reusable Service-based Systems (SBSs). Based on our study of the state-of-the-art research in the area of aspect-oriented modelling, and the application of the various modelling approaches for services and SBSs [5][6][8] in particular, we have selected an existing artefact proposed by Kim et al. [8] to construct our preliminary model. The artefact presents common concepts and a framework, *Service Oriented Design using Aspects* (*SODA*), for design and verification of SBSs using AOM techniques. It, however, lacks the support for multi-layer business oriented SBSs, translation of SLAs for the selection of aspects in order to customize service instances, and automated deployment of the model at the infrastructure level. In the course of research, we will extend the SODA approach with a view to proposing a complete solution for designing flexible business processes to be deployed using the SLA@SOI framework. This work is planned to take place within the last project year (Y3).

#### 4.3.1 Problem Description

Based on the agreed SLA, a basic SBS that is composed of service components needs to be enhanced to support additional features to meet customer-specific SLAs. The simplest solution is to have multiple SBS configured according to the different possible SLAs, but that limits the flexibility of the system to support additional structural and behavioural requirements in the future. In order to build a flexible and extensible SBS, additional features ought to be added as required based on the SLA at hand. From the management perspective, for guaranteeing the SLAs, monitors need to be activated at the proper service execution or integration points. In case of bank transactions, additional aspects like security and authorization checks should be addressed. Therefore, based on the SLA not only different system (service/software/infrastructure) components need to be instantiated but also monitored at the different layers for complete SLA-driven service provisioning. AOM allows defining system models that consist of a base model and multiple aspect models. The aspects are weaved into the base model at predefined join-points based on some matching criteria defined by a set of point-cuts. AOM, therefore, provides an effective way to define flexible SBS models that can be adapted by addition of aspects as required by specific SLAs.

#### 4.3.2 Extended Open Reference Case (ORC)

In the general case of the ORC scenario, an inventory service is used to perform the accounting for the sales and update the inventory, while an external banking service is used to carry out the financial transactions. To illustrate SLA-based service provisioning we consider that there are two types of cash-desks in the store, *automated* and *cashier-assisted*. Automated cash-desks at the self-checkout lanes have a different SLA at the business layer than those of the cashier-assisted cash-desks. Consequently, the two types of SLAs are mapped to
different types and measures of service requirements. An automated cash-desk SLA specifies a 2 second service response time at the service layer, which consists of the inventory service and an automated payment service supporting only debit card transactions. The latter requires additional user interfaces to support customer interactions. Conversely, the cashier-assisted cash-desk SLA specifies a 3 second service response time at the service layer and consist of the inventory service and a general payment service that supports cash, debit and credit card payments. The customer can also place an order in which case the store or provider sends a notification to the customer when the product arrives at the store. Figure 12 and Figure 13 show the example scenario and their corresponding agreement terms in the SLAs.

Figure 12: The ORC scenario for multiple SLAs

```plaintext
// Customer selects automated cash-desk with only debit card payment option
agreement_term {
  id = AGREEMENT_TERM_SLA_1
  guaranteed_state {
    id = GUARANTEED_STATE_1
    RESPONSE_TIME <= 2 sec 
  }
}

// Customer selects cashier assisted cash-desk with cash/debit/credit card payment options and deliver-order option for which the provider sends notification once weekly after the item arrives at the store
agreement_term {
  id = AGREEMENT_TERM_SLA_2
  precondition( CUSTOMER_SELECTION = SLA_2_ITEM_ORDERED )
  guaranteed_action {
    id = GUARANTEED_ACTION_1
    actor = provider
    policy = mandatory
    // will give a reminder max. 4 times
    trigger = weekly max <= 4 times
    invocation {
      operation = CUSTOMER_INTERFACE_ID::OPERATION_1_NAME
      param {
        name = INPUT_1_NAME
        value = "Your order is ready to be picked up at the store" 
      }
    }
    guaranteed_state {
      id = GUARANTEED_STATE_2
      ITEM_AVAILABILITY <= 10 days
      RESPONSE_TIME <= 3 sec 
    }
  }
}

Figure 13: Declaration of agreement terms in a SLA
4.3.3 Research Objective

Given the above problem description, the goal of this research is to propose an approach that would facilitate SLA provisioning in a multi-layered SOA using AOM. The solution will maintain synergies with WP A3, both for the monitoring mechanisms within the overall monitoring framework and the dynamic configuration of the monitoring infrastructure. More specifically the research is targeted towards the following specific objectives:

- Provide a mapping approach for casting the higher-level SLAs to lower-level atomic service requirements at the different layers of a SOA. With the A5.3 SLA translation model (cf. Section 7.3) focusing on the extraction of Key Performance Indicators (KPIs) of the component services, this complementary work achieves a qualitative mapping of the SLA as shown in Figure 14 to identify the various aspects that need to be added to the base model for specific SLA requirements.

- Provide a modelling approach to design multi-layer SBSs that support flexible adaptation of the system based on the SLAs, which would include:
  - An approach to define the base model for the SBS
  - An approach to identify and model the aspects
  - Definition of the join-points and point-cuts in order to enable automated weaving of aspects into the base model
  - Definition of a dependency model for the aspects for prioritization of aspects when multiple aspects are weaved at any one join-point

- Development of a tool to automate weaving of aspects to the base model
- Development of a tool to automate generation of the deployment code for the various components of the weaved model at the different layers.

4.3.4 The SODA Approach

We deem the concepts and techniques in the Service Oriented Design using Aspects (SODA)-based approach proposed by Kim et al. [8] as a feasible and appropriate approach to use for defining our initial model, which has to be extended as described in our research objective in Section 4.3.3. The SODA-based approach defines some core concepts about an atomic service, service-chain, and service-net, and considers every component of a SBS including an aspect as an atomic service. UML2 component diagram as shown in Figure 15 is used for graphical representation of the SBS, which shows three different types of aspects – the ‘common’ aspects depict common cross-cutting properties such as

---

**Figure 14: Mapping table for SLA to SLO and KPIs**
logging; the ‘policy’ aspects are used to specify QoS guidelines, and the ‘value-added’ aspects are used to customize the behaviour of services by altering behaviour of the messages. SODA uses an extended relationship named “\(<\text{crosscut}>\)” with the tagged value “pointcut” that designates where and how to crosscut a service element. The value of “pointcut” contains one of five crosscutting methods, which are before, after, around, proceed, and flow.

The authors propose three XML-based languages as parts of the SODA approach namely, the Service Markup Language (SvML) and the Aspect Markup Language (AsML) to specify the structural description of the services and the aspects respectively, and the Petri-Net Markup Language (PNML) to specify the behavioural information of the cross-cutting concerns, which supports five different join-points. The authors also present a tool for aspect weaving using the above languages with a focus on verification of the weaved-model using existing Petri-Net tools. We would use the concepts and languages proposed in the SODA-based approach to specify the base and aspect models for a SBS, and thereby, explore our research goals with a view to defining a complete AOM framework for multi-layer deployment of business processes.

Figure 15: Service decomposition diagram [8]

For the ORC example described in Section 4.3.2, the base model would consist of the Inventory service and a general Payment service configured for cash payment. For both SLA1 and SLA2, a Monitoring service would be added as a Performance policy aspect for monitoring the performance with different KPIs. For SLA1, the Payment service would need to be customized by triggering a chain of external services such as Card Validation and DebitCard services as value-added aspects. For SLA2, the Order service will be added after the Inventory service as a value-added aspect so that the consumer can order the product if it is not in the store. Also a Payment type service would be added as a value-added aspect.
before the Payment service, which would trigger preferably dynamic selection and weaving of the appropriate value-added aspects (Card-validation and a DebitCard or Credit-card service).

The next phase of research will focus on describing the services and aspects using the SvML, AsML, and PNML specifications for developing a prototype based on the SODA approach with more detailed design of the SLA translator. The primary goal at this stage would be to generate the appropriate deployment code for multi-layer SBS from the model. We would also study the feasibility of developing a graphical tool to generate the SvML, AsML, and PNML specifications for the SODA approach, which currently have to be done manually, and study the possibility of using other specification languages to better link the model to our existing service monitoring approaches used in the SLA@SOI framework.
5 Template Management

The management of SLA Templates (henceforth SLATs) is achieved by a dedicated SLA Template Registry component and the Service Advertisements Bus, a publish/subscribe system for template advertisements, both developed under Task A5.2. The SLA@SOI architecture specifies that each SLA Manager in the framework incorporates two SLAT registries, one for customer-facing SLATs (those offered by the SLA Manager to customers), and another for provider-facing SLATs (those offered by external providers to the SLA Manager). In the current implementation, both customer- and provider-facing registries are implemented by a single concrete registry, with the different SLAT roles distinguished by a metadata “flag”. The following section presents a brief summary of the functional capabilities provided by the SLA Template Registry; then Section 5.2 presents the Service Advertisements Bus.

5.1 Template Registry

The basic functionality of the SLA Template Registry is to:

i) implement a persistent and accessible store for SLATs.

ii) enable the QoS-based discovery of SLATs, i.e. to provide a query mechanism which supports matching SLATs according to non-functional, as well as functional, service properties.

iii) enable metadata-based retrieval of SLATs, e.g. according to whether the SLAT is customer- or provider-facing, the time at which it was registered, its creator, and so forth.

The realisation of the SLA Template Registry, and in particular the QoS-based query mechanism, is naturally tightly coupled with the SLA Model (described in Section 4.1) and to the “Standard-Terms” employed within SLA@SOI (presented in Appendix C). SLA(T) validation (e.g. the problem of verifying the conformance of an SLA to an SLAT) is also closely related to QoS-based query resolution, and as such it is convenient to extend the functionality of the SLA Template Registry to include:

iv) validation of SLA(T) content and SLA conformance.

As noted in Section 4.1, effective SLA(T) validation requires that domain-specific terms are formally classified according to a terms meta-model - specifying the role that each term plays in an SLA (e.g. ‘action-policy’, ‘metric’, etc.) and any associated parameters and/or data-types. For the SLA@SOI Standard Terms, this classification is automatically available to the SLA Template Registry. For domain-specific extensions/alternatives to the Standard Terms, however, the registry also needs to support:

v) the registration of formally specified domain-specific vocabulary extensions.

The SLA Template Registry exposes all this functionality as a single interface (summarised in Figure 17), which is accessible to all the internal components of a SLA Manager. The various query mechanisms are also encapsulated as a web-service for use by external components (in particular, the framework’s Business Manager).
The SLA@SOI architecture also specifies that the SLA Template Registry interacts with the Service Advertisements Bus in order to automatically publish and subscribe SLATs (according to policies dictated from the business level). Implementation of this functionality is scheduled for Y3. The following sections provide an overview of the functionality that has been implemented in Y2.

### 5.1.1 SLAT Registration, Persistence & Retrieval

Registration of SLATs is effected through the SLA Template Registry method:

- `addSLATemplate(slat:SLATemplate,metadata:Metadata):Warning[]`

The SLATemplate class, encapsulating the SLAT itself, is defined as part of the abstract SLA(T) syntax (described in Section 4.1.2). For administrative purposes, every registered SLAT has associated metadata, encapsulated by the class Metadata (shown in Figure 18), which is essentially just a dictionary of key/value property pairs\(^\text{10}\). Among other things, this metadata includes information identifying the provider of an SLAT, its creator and the time at which it was registered.

---

\(^\text{10}\) The class ‘Annotated’ is defined at the ‘Ground Expressions’ level of the SLA Model hierarchy (Figure 5).
Upon invocation, the `addSLATemplate(..)` method first checks the validity of both the SLAT and its metadata:

i) metadata is checked for completeness & for consistency with previously registered information (e.g. to avoid duplicate template UUIDs),

ii) SLAT content is validated, as described below in Section 5.1.3.

If any critical issues are detected (such as; duplicate identifiers, inaccessible service-descriptions, inconsistencies, etc), then registration is refused and an exception is thrown. Otherwise, the SLAT is accepted and stored (warnings are generated & returned for non-critical issues).

In line with technical integration specifications from WP A1, persistent storage is effected by a backend MySQL database. Each SLAT is stored in two forms:

i) as an opaque object (BLOB) for rapid retrieval, and

ii) as a indexed collection of rules and facts\(^\text{11}\); supporting the reasoning mechanisms required for QoS-based query resolution.

For modularity, the various interface-specifications (i.e. functional service descriptions) embedded in, or referred to, by SLATs, are extracted (or downloaded) and also stored in the registry (as opaque objects).

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

- `getSLATemplate(templateId:UUID):SLATemplate`
- `getInterfaceSpecification(interfaceName:String):Specification`
- `getMetadataProperty(templateId:UUID,key:STND):String`
- `getMetadata(templateId:UUID):Metadata`

More complex (query-based) retrieval methods are described in the next section.

The SLA Template Registry does not support in-line modification of SLAT content. Instead, to modify a SLAT, either:

i) the desired SLAT must first be removed from the registry, modified, and then re-registered, or

\(\text{11}\) In Y2, the rule/facts base & reasoning mechanisms are implemented in Prolog.
ii) new “versions” must be registered with distinct UUIDs (version tracking is also supported by the metadata properties 'previous version' & 'replaced by version').

The removal of SLATs is effected by the method:

• removeSLATemplate(templateId:UUID):void

The removal of any SLAT, however, has a direct impact on SLA management. In particular, since each SLA has a built-in reference to the SLAT from which it was derived, then removal of that SLAT from the registry can lead to broken references. Given this tight-coupling the removeSLATemplate() method is unavailable in Y2, and will be implemented in Y3 following more detailed workflow analysis. SLAT metadata can, however, be modified, using the method:

• setMetadataProperty(templateId:UUID,key:STND,value:String):void

Finally, the SLA Template Registry provides some support for asynchronous notifications:

• addListener(listener:Listener):void
• removeListener(listener:Listener):void

For Y2, notifications are posted just in case of (un)successful registration events.

5.1.2 SLA Template Registry Queries

The SLA Template Registry supports both metadata-based queries, and QoS-based queries, cf. the methods:

• query(metadata_query:ConstraintExpr):UUID[]
• query(query:SLATemplate,metadata_constraint:ConstraintExpr):ResultSet

Metadata-based queries are fairly straightforward. Since SLAT Metadata comprises a simple list of key/value property pairs, a query over metadata can be readily expressed as a (compound) constraint\textsuperscript{12} on property values. The result of a metadata query is just a list of (UUID) identifiers of those SLATs whose metadata meets the expressed constraints.

QoS-based query resolution is more complex. The basic idea is to support discovery of SLATs according to both the functional and non-functional properties of the services they describe. A customer, for example, may require a service not just with particular functional capabilities, e.g. the function foo(), but also with particular QoS properties, e.g. completion-time of foo() < 10ms. To support this kind of discovery, we require:

i) a means to express the customer’s QoS requirements, i.e. a formalism for expressing QoS-based queries, and

ii) a means to match the QoS constraints expressed in queries against the QoS constraints described in SLATs.

For representing queries, we assert that the content of queries is essentially identical to the content of SLATs (both express QoS constraints), such that we can simply re-use (i.e. overload) the abstract syntax for SLATs. We thus define a

\textsuperscript{12}Constraint expressions are encapsulated by the ConstraintExpr class, defined at the ‘Ground Expressions’ level of the SLA Model hierarchy (see Section 4.1.2).
QoS-based query as a partial\textsuperscript{13} SLA Template. The problem of QoS-based query resolution is then equivalent to finding the set of SLATs with QoS constraints which are at least as stringent, or more so, than the QoS constraints expressed by the query SLAT. To complicate the problem, we also need to deal with the possibility that different SLATs express similar constraints in different ways (e.g. in different units, or logical combinations), or more significantly, that the presence of a constraint may even be contingent on other constraints (i.e. in SLAT terms, there may be preconditions on guarantees).

To resolve QoS-based queries thus entails the use of some fairly substantial formal reasoning mechanisms. Indeed, in certain cases, it may not be possible to compute a definitive match at all, but rather only a measure of probability of match. To this end, QoS-based query results are encapsulated in a “ResultSet” (Figure 19), which, among other things, provides a level-of-confidence for each possible match.

![Figure 19: QoS-Based Query Results](image)

As a minimum requirement, the QoS-based query mechanism assumes that both queries and registered SLATs are at least valid. To this end, the SLA Template Registry also performs SLAT validation, which is described in the next section.

### 5.1.3 SLAT Validation & Domain-Specific Term Extensions

The SLA Template Registry exposes a single method for validating SLATs:

- `validateSLATemplate(slat:SLATemplate):Warning[]`

This method is automatically invoked whenever an SLAT is registered (see Section 5.1.1 above). The validation process checks for the following conditions:

- the SLAT must have a non-null & unique identifier.
- all mandatory template content must be present, and entity relations must obey the cardinalities defined by the SLA Model.
- the IDs of all identifiable content elements are cross-checked for uniqueness.
- all internal variables are resolved & cross checked for semantic correctness, e.g. interface identifiers should identify interfaces, party identifiers should identify parties, and so on.

\textsuperscript{13} Some of the constraints on SLAT content – e.g. that at least one party must be specified – may be dropped for QoS-based queries.
• All references to externally sourced interface specifications are resolved, and the corresponding specification downloaded and cached in the backend database.

• all (domain-specific) standard terms are checked against registered terms, and all expressions, and parameters to event & functional expressions, are checked for type validity.

In order to download externally sourced interface specifications, the SLA Template Registry invokes a pluggable component called a ‘ReferenceResolver’. The default implementation of this component assumes that specifications are provided as WSDL 2.0 documents, and are accessible by standard HTTP request. To override this default behaviour, the registry provides a method for using domain-specific ReferenceResolver implementations:

• setReferenceResolver(resolver:ReferenceResolver):void

Finally, validation of standard terms requires that all terms be formally specified (according to the terms meta-model, see the introduction to this section). The formal definitions for all the SLA@SOI “Standard Terms” (Appendix C) are built-in to the registry. Any domain-specific extensions/alternatives, however, must first be formalised and then registered. In Y2, these formalisms are encapsulated as the class ‘Extensions’, instances of which can be (un)registered using the methods:

• addExtensions(extensions:Extensions):void
• removeExtensions(extensions:Extensions):void

In Y3, the abstract SLA syntax will be extended to allow extensions to be specified (or referenced) as part of the SLA(T) content itself, thus providing for a more flexible and domain-independent registry implementation.

5.2 Advertisements System

The SLASOI framework is meant to enable distributed components and parties to come together in order to effectively negotiate, collaborate and manage a Service Agreement platform. For this, the various modules need to discover and interact with each other through their corresponding SLA Managers. However, viewing the distributed nature of the system, there is a need to asynchronously exchange information that will allow the various SLA Managers to communicate with each other. The Advertisement System steps in to fill this gap by allowing the sharing of SLA templates in a seamless, interoperable and asynchronous manner (see also Figure 2). At the heart of the Advertisement system there exists a Publish/Subscribe system - used to disperse the SLA Templates between interested parties. By using the SLA Template, SLA Managers are able to discover each other and perform various interactions. The Templates can then be stored locally in the TemplateRegistry of the involved SLA Manager. Whenever a change occurs, the Advertisement system propagates this to all the subscribed parties. This allows the SLA Templates to stay updated despite various copies being stored in geographically distant locations. During run-time service providers can publish new templates to the system.

5.2.1 Apache ActiveMQ

The Advertisements System is a Java Message System based on the Apache Message Broker [15]. Apache ActiveMQ 5.3 is an open source message broker which fully implements the Java Messaging Service 1.1 [16]. The publish and subscribe models provided by ActiveMQ enable sending messages to multiple
recipients. Subscribers may register their interest in receiving messages about a particular topic. In this model, neither the publisher nor the subscriber know about the existence, or the communication endpoint of the other. In ActiveMQ, the publisher is able to publish messages at any time. The subscriber has to remain continuously active to receive messages, unless it has established a durable subscription. In that case, messages published while the subscriber is not connected/available will be sent whenever it reconnects. The durability feature of a subscription is configured at the initialization of client.

5.2.2 Features of Advertisements System

The choice of the ActiveMQ technology was based on several criteria that it satisfies. First and foremost, in addition to being open source and standards-based messaging middleware, the facilities required by the SLASOI project are mostly available out of the box. This is complemented by the ease of its adoption. The Advertisements system wraps the ActiveMQ and provides proxies for the SLASOI providers and customers that enable them to publish and subscribe to the system without having to install heavyweight software or machinery as a prerequisite. This also maintains the interoperability with the overall architecture of the project.

The broker plays a central role in any advertisement system. ActiveMQ provides an efficient mechanism to plug in a broker and pause or resume its operation. In the unlikely event where the main broker becomes unavailable, the system automatically connects another broker seamlessly. Another interesting feature, worth considering was the fact that the ActiveMQ system is already available as an OSGi [17] bundle and is also compatible with Spring [18]. As we have made extensive use of these technologies within the SLA@SOI platform, this reduced significantly the respective integration effort and time. If at any stage, it is required to extend or modify the ActiveMQ system, there would be no obstacles as it is published with the Apache license. The most important features of the Advertisements System are summarized in the following table.

Table 2: Main features of Advertisements System

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interoperability</td>
<td>Java, C, C++, C#, Ruby, Perl, Python, PHP</td>
</tr>
<tr>
<td>JDBC message store</td>
<td>Supports very fast persistence using JDBC along with high performance journaling</td>
</tr>
<tr>
<td>OSGi integration</td>
<td>Broker and clients are available as OSGi/Spring bundles</td>
</tr>
<tr>
<td>Clustering</td>
<td>The system supports a collection of JMS brokers and clients will connect to any one of them;</td>
</tr>
<tr>
<td>Protocols</td>
<td>Supports pluggable transport protocols such as in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA</td>
</tr>
<tr>
<td>Security</td>
<td>• Provides pluggable security through various different providers</td>
</tr>
<tr>
<td></td>
<td>• LDAP Authentication via JAAS Plugin</td>
</tr>
<tr>
<td></td>
<td>• Broker-to-Broker Authentication and Authorization</td>
</tr>
</tbody>
</table>
5.2.3 Advertisements System Brokers

The Broker component in the Advertisements System is responsible for the delivery of templates between two or more SLA-managers. A broker can be aligned with others into a cluster or hierarchies of them allowing the configuration of complex structures for increasing reliability and redundancy of the Advertisements system in the SLASOI framework. A set of brokers can be either specified in design time via a configuration file or the brokers can be discovered dynamically, as illustrated in Figure 20. Broker #1 is pre-configured in the SLASOI framework while Brokers (#2, #n and #m) can join together dynamically. The SLA-Templates published by “Publisher-X” in the system will be propagated across all known brokers. Each Subscriber (A, B and C) will get a copy of these templates.

![Figure 20: Advertisements System](image)

5.2.4 Channels

As mentioned above, the Advertisements System of the SLA@SOI framework enables the interaction between several SLA Managers by exchanging SLA templates before negotiation for reasons of caching. This exchange supports two types (renderings) of SLA templates at the moment: WSAgreement and SLA@SOI Model. The Advertisements System defines one channel for each SLA Template type. Those channels enable interoperability between the SLASOI framework and legacy systems, allowing publishers and subscribers to share SLA Templates in different formats. It is straightforward for someone to implement a new Syntax Converter and make it part of a SLAM, to implement a different protocol in some framework instance.

5.2.5 Plug-in based Client

The SLASOI framework provides a plugin-based client for interacting with any available Broker. This client offers the basic functionality for subscribing to a specific channel as well as facilities for publishing templates. The client is configured by default for connecting to a predefined broker in the SLA@SOI Framework. As shown in Figure 21, the Pub/Sub Client (small brown-box) is
integrated within the Generic-SLAM, enabling it with access to each component from any domain specific SLA-manager.

![Figure 21: Publish/Subscribe Client](image)

### 5.2.6 Functional Interfaces

The Advertisements System is modelled by the following interfaces:

**ServiceAdvertisement**

- `getIPublishable`: sends the templates via the internal broker to current subscribers
- `getISubscribable`: notifies the internal broker about a new subscriber/agent and links it to the specified channel

**IPublishable**

```
publish(Template[] slaTemplate, ServiceAdvertisementChannel channel): sends the templates via the internal broker to current subscribers
```

![Figure 22: Interfaces of Advertisements Service](image)
ISubscribable

- `String subscribe (ISubscriber subscriber, ServiceAdvertisementChannel channel):` notifies the internal broker about a new subscriber/agent and links it to the specified channel. Returns an Identifier of the Subscription process.

- `void unsubscribe (String subscription-ID):` unbinds the subscriber from the specified channel

## 5.2.7 Authorization

The default authorization mechanism of the Advertisements System relies on plugin provided by ActiveMQ. Using this plugin it is possible to define users and groups directly in the broker's xml configuration. The plugin-based client for the Advertisements Systems is provided with a default configuration to be able to interact with any Broker in the SLASOI framework.
6 Negotiation

Negotiation has a central role in SLA management frameworks. It has multiple aspects, all of which must be amicably accommodated by the framework. In order to convey a clear picture, this section defines negotiation in terms of a negotiation process. A negotiation process consists of two distinct elements:

- Negotiation Protocol
- Intelligence needed to create/process an offer or counter offer.

Of these, the former is explained in more detail in section 5.2 where we present a generic approach that may be used as such or taken as a reference to write custom protocols. The latter is embodied as a separate component called the “Planning and Optimization (POC)” component. To execute the negotiation process on behalf of the negotiating parties, a component called the “Protocol Engine” has been developed.

6.1 ProtocolEngine

The Protocol Engine is an integral part of the Generic SLA Manager and therefore is used by other SLA Managers as such. It allows a customer to interact with a provider by implementing the INegotiation interface (section 6.1.2). The actual behaviour of the negotiation is regulated by the Negotiation Protocol being used by the engine. Two or more parties negotiating together use the same protocol and the component needed for its execution is the Protocol Engine. It also implements an IControl interface that allows a certain protocol to be configured per SLA Manager. This empowers administrative modules to view and modify negotiation policies at runtime.

In Year 3, Protocol Engine will be extended to enable multi-party negotiations, in addition to its current bilateral functionality. Moreover, the control feature will be extended to allow unique policy configurations per conversation conducted by its SLA Manager.

6.1.1 Design

The Protocol Engine is designed to meet the needs of its actors. These include a customer, a provider and a protocol writer. Each one of these has a certain role to play and the Protocol Engine helps to connect them together. Figure 23 captures this relationship from a high level view. An important design goal, worth mentioning here, is that although the protocols tend to be domain specific, the Protocol Engine remains domain agnostic.
**Multi-tiered Design**

The Protocol Engine harnesses a multi-tiered design to have a good division of labour. This is captured in Figure 24.

*Message Handling:* This tier acts as a façade to the Protocol Engine and assumes (for Year-2) a reliable communication. Reliable communication is independent of the Negotiation Protocol being used.

*Negotiation Management:* The Protocol Engine is a stateful component and maintains a session object per negotiation. The negotiation management tier deals with session management, it generates events against the invoked operation, passes these events to the State Management tier and interfaces with the POC where needed.

*State Management:* This tier contains an internal State Engine component which is a wrapper around a rule engine. The rule engine is used to execute rules that constitute the protocol (section 6.2). This helps maintain the state of the ongoing negotiation. This tier benefits the design by acting as a mediator where a rule engine of choice is plugged into the Protocol Engine. Currently, the rule engine of choice is “Drools” by JBoss.

---

**Figure 24: Tiers of Protocol Engine**

To maintain compatibility/interoperability with WSAG based clients, a Syntax Converter module (section 6.3) acts as a façade to the outside world or a proxy to the Protocol Engine. It converts WSAG (or other) models to SLA@SOI SLA model in-memory representation, and vice versa.

At this stage, it becomes meaningful to take a closer look at the architecture of the Protocol Engine component where all the design concepts discussed so far
come together in Figure 25. The figure clarifies how two SLA Managers, each using its Protocol Engine, communicates with the other (via a Syntax Converter) and how the multi-tiered design of Protocol Engine is realized through 3 internal components, namely Message Handler, Negotiation Manager and the State Engine. Various external and internal interfaces are also depicted, along with their required and provided dependencies.

**Figure 25: Component Diagram of Protocol Engine**
6.1.2 Interface

INegotiation

The Protocol Engine exposes its negotiation functionality through the "INegotiation" interface, which has the following operations:

initiateNegotiation: Request to start a negotiation session. A session ID is generated and is used for subsequent operations.

negotiate: Submit an offer or counter-offer to the other party. The Protocol Engine consults the POC and returns the result from the POC back to the caller.

createAgreement: Request the negotiating party to accept the submitted offer and return the SLA. The negotiating party may also refuse the request. This operation requires consultation with the local POC(s).

renegotiate: Request to start renegotiation over an already created SLA. The Protocol Engine verifies the SLA, prepares a new negotiation session, assigns it an ID and returns to the caller – to be used with the negotiate operation that follows. The receiving end may decide to refuse renegotiation and ignore the request.

cancelNegotiation: Cancel ongoing negotiation. Any party may decide to quit negotiation by providing a cancellation reason.

terminate: End the Service Level Agreement with the other party. Any party may decide to terminate an SLA by providing a termination reason. The receiving end may decide to refuse renegotiation and ignore the request.

provision: Provision a service already agreed upon in an SLA. This helps to coordinate when a hierarchy of SLA Managers is involved. This operation requires consultation with the POC.

IControl

In addition to the negotiation operations, the Protocol Engine exposes its Control functionality through an "IControl" interface which has the following operations (explained in simple terms):

setPolicies: Set the list of policies corresponding to the provided policy type. If the policy type is "Negotiation", the Protocol Engine creates or overwrites the Protocol. If the policy type is "Adjustment", the call is delegated to the Provisioning and Adjustment Component (PAC).

getPolicies: Fetch and return the list of policies against the provided policy type. Delegate to PAC if type is "Adjustment".
6.2 Generic Protocol

Negotiation takes place as per the Negotiation Protocol. The Negotiation Protocol consists of a negotiation model and necessary constraints to ensure that the integrity of this model and related business checks is maintained throughout a negotiation session. Before introducing the Generic Protocol designed under WP A5, it is worthwhile to briefly mention some of the Negotiation Models.

6.2.1 Negotiation Models

These are real world business models used for negotiations. Some of these models, their IT counterparts, cardinality {customer – provider} and visibility [among providers] are shown below:

- Fixed Price (Take it or leave it) \{1-1\} [n/a]
- Sealed Bid (Contract Net Protocol) \{1,n\} [not visible]
- British Auction (Outcry) \{1,n\} [visible]

Other examples include Reverse British Auction, Vickrey Bids, Dutch Bids, Double Auction, Market Place, etc. The semantics of these models and their employed usages helped realize a generic approach to protocol description with the resulting protocol termed as the "Generic Protocol". The Generic Protocol is kept flexible and extensible by design as well as loosely-coupled with the Protocol Engine – the component which executes it (or one of its specialized forms).

6.2.2 Design

In SLA@SOI, the Generic Protocol implements a Negotiation Model as a State Machine. The state machine captures 1) The states in the model, 2) The sequence of messages allowed, 3) Restrictions on a state, and 4) Transitioning between states.

The state-machine-based design provides a solid foundation to structure the protocol as it bridges negotiating parties based on their preferred model for conducting business. At any stage during negotiation, the state of the machine at one party stays synchronized with the other – until one changes; triggering the other to attempt the same.

![Figure 26: A General Purpose State Machine](image-url)
The Generic Protocol uses a general-purpose state machine (Figure 26) which can be used, for example, to model the Sealed Bid Negotiation Model (along with several others) requiring multiple rounds of negotiation [19].

**Rule Oriented Approach**

Logic-programming rules were selected as a technology of choice to encode these and other protocol constraints. The approach provides rich expression semantics without compromising on configurability. A lightweight object model is developed that provides basic classes to structure the state machine (among other concepts). The protocol is written as a collection of rules and is made version-wise available via a repository/location inside a SLA Manager.

A taxonomy of Events has also been realized. These are generated by the Protocol Engine, corresponding to the operations invoked on its negotiation interface (section 6.1.2). An event serves to trigger a protocol behaviour, which is executed provided that related constraints are satisfied. This results in a negotiation that is regulated or controlled through the rules.

Encoding Protocols as a set of rules provides benefits inherent in the rule-oriented approach. Some of these are as explained below:

1- The protocol enjoys declarative style encoding, which is better compared to coding it using an imperative programming language. It is considered that logic cluttered in programming instructions is hard to maintain.

2- It is possible to overwrite protocol behaviour at runtime using the IControl interaction (section 6.1.2).

3- The protocol file remains human readable and machine interpretable. The intuitive style of writing rules helps shorten the learning curve for the domain experts who may have to act as protocol authors.

4- Once encoded, the protocol acts as a reusable component that is plugged into the Protocol Engine and used by multiple participants.

Rules are activated upon the arrival of an Event or through other Rules – as their qualification clauses become satisfied. Rules may perform a variety of functions, such as to create or remove instances (using the object model) in working memory as needed to implement the state machine; implement transitioning behaviour between states; setting hop count on a state; validate whether an incoming Event is allowed in the current state; make business validations on incoming offers and counter-offers in order to ensure that constraints on QoS are not violated or whether QoS values are in a certain range; set timeouts between two negotiate calls; and so on. A rule may even generate an Exception in case it needs to interrupt the operation invoked on the Protocol Engine that led to a negative assessment of some conditions by the corresponding rule(s).

**6.2.3 Code Snippets**

To get a first hand impression of how the Generic Protocol or its specializations actually look, a few code snippets are presented below.

An example (bootstrap) rule that spins a State Machine into action looks as follows:

```
rule "BootstrapStateMachine_Rule"
```
when
  event : Event(eventName == EventName.StartNegotiationEvent);
then
  insert(new State(StateName.START));
  retract(event);
end

Another example that limits negotiation rounds on the NEGOTIATE state.

rule "Limit_Negotiation_StateRule"
when
  negotiateState : State(name == StateName.NEGOTIATE, status == StateStatus.RUNNING, currentHop >= numberOfHopsAllowed );
then
  negotiateState.setStatus(StateStatus.READY_TO_TRANSIT);
  update(negotiateState);
end

The following rule performs a business validation that the Agreement Terms within an incoming offer are not null. This rule is executed with the arrival of AgreementRequestedEvent on the provider’s side and is bound with the createAgreement operation of the ProtocolEngine’s negotiation interface. Since the rule is negatively evaluated, it sets the processedSuccessfully property to false and sets an appropriate message which is used by the Protocol Engine to throw an OperationNotPossibleException back to the caller.

rule "Negotiate_To_Decide_Transition_AgreementRequestedEvent_Rule"
when
  negotiateState : State(name == StateName.NEGOTIATE, status == StateStatus.READY_TO_TRANSIT );
  event : Event(eventName == EventName.AgreementRequestedEvent, eval(offer.getAgreementTerms() == null));
then
  event.setProcessedSuccessfully(false);
  event.setProcessingAfterMath("The Offer Template has no Agreement Terms to consider");
  retract(event);
end

6.3 Syntax-Conversion for Interoperability

The Syntax Converter is the component of the GSLAM that provides interoperability and separates the GSLAM from specific representations of SLAs, SLA-templates and further required data types. Example representations are the XML representation of SLA@SOI’s SLA model, WS-Agreement for representing SLAs or possibly additional formats in languages such as JSON or RDF. The GSLAM-internal interface of the syntax-converter provides means to its related components to execute operations using the SLA-models Java representation. On the other side the syntax-converter implements various interactions as web services and provides support for specific protocols and representations. As such,
the syntax-converter is a central communication point for the GSLAM and shields other components from representation-specific implementation.

### 6.3.1 Purpose

The syntax-converter’s purpose can be separated into two categories. The first one is the conversion between various data-types. The second purpose of the syntax-converter is to provide an entry point to the GSLAM.

In its current state the framework provides an elaborate model for representing SLAs and SLA-templates, as described in Sections 4.1 and 4.2. The model itself is defined using the abstract syntax expressed with BNF and textual descriptions. To apply the model to a use case, a concrete implementation is necessary. One such implementation is the Java-representation, which is used by components throughout the framework. For on-the-wire communication purposes two XML-representations exist. One directly implements the abstract syntax in XML, which is referred to as the XML-representation of the SLA@SOI SLA model. The other one embeds the direct representation of the SLA-model within WS-Agreement. This extended WS-Agreement representation directly uses the SLA model’s XML representation within the WS-Agreement frame. Those textual representations are mostly used whenever a SLA needs to be serialised and read by a machine at a later stage again. Example uses are serialisation for communication over the wire such as using web-services and serialisation for storing SLA-documents in the SLA-registry and the SLA-template registry. In addition there exists a textual representation of the SLA-model that is conveniently readable by humans in JSON-notation. This representation however exists for convenience and is not used by the syntax-converter currently.

In this context the syntax-converter provides bidirectional conversion mechanisms between the SLA-model’s Java- and XML-representation. It also provides a bidirectional mechanism for converting between the Java-representation and the XML-representation embedded in WS-Agreement.

Additionally the syntax-converter is able to convert between the SLA-model’s interface-specification and WSDL 2.0 in both directions. This is needed when an interface is provided externally that is described in WSDL. In the other direction this mechanism is necessary to provide a commonly understood description of a web-service interface to external parties who do not know SLA@SOI’s interface specification model.

The syntax-converter’s external interface provides the means for GSLAM’s internal components to communicate with external components. Those external components may belong to the framework itself or to components implemented by third parties. Components within the framework that are interacting using the syntax-converter’s external interfaces are the GSLAM itself and the business-manager. A GSLAM will communicate with another GSLAM using the syntax-converter as if it were any external component. In its current state the syntax converter provides SOAP-based web-services described using WSDL 1.1. For certain interactions it additionally provides client-classes that wrap the web-service specific logic and data types. The services themselves also shield the components used, by providing wrappers that translate serialised data types to and from the corresponding Java implementation.
Web services for the following interactions are currently implemented in the syntax-converter:

**negotiate:** The negotiate interaction is provided using a custom created WSDL-file. All calls to this interaction are translated to their Java-representations and delegated to the Protocol Engine.

**query:** The query interaction is realised for the Template Registry as well as for the SLA registry. Again calls are translated to the Java-representation and delegated to the template registry and to the SLA registry.

**provision:** The provision interaction is currently contained by the WSDL-description for the <<negotiate>> interaction. Here again the syntax-converter acts as a proxy-component for the protocol engine.

All those interactions are also available as clients. Those clients provide convenient means to interact with the related web-services as if the remote components were locally available.

Additionally for the <<control>> and <<track>> interactions client functionality as described above is provided.

### 6.3.2 Design

The following section will briefly summarise design considerations taken into account during the syntax-converter’s implementation.

Firstly extensibility plays an important role, since it should be possible to extend the syntax-converter easily to support future representations of SLAs and SLA-templates. For this the component is separated into parsers and renderers. Parsers convert a serialised SLA into the Java-representation. Renderers perform the inverse operation. Both parsers and renderers are separated into ElementParser/Renderer classes. These encapsulate the main logic needed to convert within a specific language (e.g., XML). Currently one type of element-parser/render exists, namely one that can convert to and from XML. This approach has the advantage that it is easy to implement new parsers and renderers for the same language. As an example, the existing parser for the SLA@SOI representation and the parsers for the WS-Agreement representation both use the same element-parser (i.e., XmlElementParser). Both the WS-Agreement parser and SLA@SOI parser are provided with minimal implementation effort. The mentioned XML-element parser is also used to provide parsing capabilities for conversion from WSDL 2.0 to SLA@SOI’s Java representation of the interface-specification type.
6.3.3 Status of Implementation

The current implementation of the syntax-converter provides all basic functionality for the direct XML-representation of the SLA@SOI’s SLA model as described above. Parsing and rendering of WS-Agreement based documents is also implemented, however the WS-Agreement related interactions are not implemented i.e. the WS-Agreement web service is not yet offered.

The syntax-converter uses the following technologies for implementation:

• *XMLBeans* is used as an intermediary parser and renderer
• *XPath* is used heavily as tool in parsing and rendering
• *Axis2* is used to generate and deploy web-services and generate client-stubs
• *XMLSchema* defines the SLA@SOI SLA model and further types for the various interactions in XML format
• *OSGi* is used as a plug-in/bundling mechanism and especially for deploying the web-services in combination with Axis2

Future steps include a full implementation of the WS-Agreement interface, to provide compatibility with existing WS-Agreement implementations.
7 Planning

7.1 Software Planning and Optimization Component

The Software Planning and Optimization component (SWPOC) has the role of a local executive controller within a SLAM that manages software services. First, it provides the functionality to assess and customize offers / counter-offers exchanged between the customer and provider, in which they argue over acceptable values of free variables (QoS terms) of a template. The Software POC therefore, works in liaison with several other components in order to achieve its objectives. These are the Software Service Manager (SSM) and the Software Service Evaluation (SSE). Secondly, when an SLA is created, the SWPOC asks the Provisioning and Adjustment component to provision the service when so requested.

7.1.1 Design

The SWPOC exposes its functionality through the IAssessAndCustomize interface. As mentioned above, the SWPOC depends on several other components and needs to use different models (SLA, SCM\textsuperscript{14} and SSE). Due to the many dependencies which are necessary for the operation of the SWPOC, and the complexity associated with processing multiple different models and bringing them together as needed, our effort in Y2 has focused mostly on these integration activities; however, activity on theoretical aspects of the problem was also initiated and will become our main focus for the 3\textsuperscript{rd} project year.

This section aims mainly to describe the interfacing of the SWPOC to the SSE (the module developed under WP A6). The essential concept of this interaction is that SSE carries the knowledge to compute what is feasible from a performance point of view; then the SWPOC can associate the various options with some utility, and select the best solution.

7.1.2 Software Service Manager and Service Evaluation

To understand how the SWPOC works, let us first consider a simplistic hypothetical scenario where the target service has no dependency on any other service. Figure 28 captures the sequence where the SWPOC is consulted, in order to assess an offer coming in from a negotiating partner. The SWPOC communicates with the SSM to identify the Service Type of the SLA Template and afterwards queries for possible Service Implementations of this Service Type. It chooses one implementation and looks for any dependencies mentioned therein. It then requests the SSM to create a Service Builder. Following this, the builder object is passed on to the SSE component, which returns evaluation results for the given values of QoS attributes. The SWPOC picks one, and invokes the reserve operation on SSM, in order to tentatively reserve any resources before returning the counter-offer. Should the customer choose to accept this counter-offer, the reserved resources would be confirmed through a book operation.

\textsuperscript{14} Service Construction Model / SCM; also see D.A1a and D.A3a
In a more realistic scenario, a service has multiple dependencies on other services – at the very least, infrastructure services. A dependent service may offer multiple SLA Templates. The SCM assumes a 1-1 relationship between Dependencies, and Templates (as expressed in the Service Binding class). As such, the SWPOC will typically create multiple Service Builders per Dependency per SLA Template. The outer loop in Figure 29 illustrates these SSM queries. For each Service Builder, the SWPOC also needs to resolve dependencies (i.e. corresponding SLA Templates need to be retrieved from the SLA Template Registry and added to the Service Binding, which is then added to the Service Builder before submitting it to SSE for evaluation).

The SSE operates on concrete values for dependent services. Therefore, if one of the templates in one of the builders contains a range, the SWPOC will quantize this range and invoke the SSE multiple times, as shown in Figure 29 with the inner loop. When a returned evaluation set satisfies the use-case-specific utility assumed by the Plan operation, it is selected and the negotiation process continues by means of reserving resources.
7.1.3 Plan and Optimize

Services have multiple QoS attributes that may also be correlated e.g., availability and cost of a service. The negotiating parties may have conflicting goals (e.g. clients want maximum service availability for the least possible price while providers may want to maximize the profit for their service). Moreover, services may be dependent on other services.

The SWPOC solves the problem of choosing appropriate dependent services for a service under negotiation. The dependent services may be software or infrastructure services. This situation is illustrated in the dependency graph of Figure 30, where service 1 is depending on service 2 and on service 3. A number of QoS criteria $q_i$ from a predefined set of QoS criteria $Q$ is associated with each service. The problem of planning and optimization lies in satisfying the QoS demands for service 1 with the available QoS offers from the required sub-services by making correct assignments between sub-services to the requested service. These assignments will then be governed by a fully negotiated SLA. During the negotiation phase for these sub-SLAs, the service provider and the service consumer may well have contrary objective functions, as discussed earlier for price and availability. In both cases the QoS terms and their respective intervals are predefined by the SLA-template under negotiation.

Figure 29: SWPOC negotiating a Service with multiple Dependencies
A number of approaches are being investigated to find a solution to the stated problem. One approach is to use evolutionary algorithms for finding a solution in the problem space [20][1]. The usage of evolutionary algorithms may well lead to quality solutions but also needs to be critically evaluated with regards to the computational complexity. Other approaches use various local-search methods, such as Tabu search, Simulated Annealing, Ant Colony optimization, etc, to find a reasonably good feasible solution in the problem space. For most problems that contain multiple service dependencies and a large number of potential sub-services, it may be impossible to find an optimal solution within a short timeframe which needs to take negotiation time-outs into account. So, quite often, a suboptimal solution is chosen – potentially by interrupting the executing algorithm after a certain computation time has passed. Another approach being considered is adapting the Gale-Shapley algorithm [22] to solve this planning problem.

To provide a possibility to easily evaluate a number of planning and optimization algorithms, the SWPOC implements the Strategy design pattern. This allows the user to easily swap algorithms during development time. Future directions include finding efficient algorithms that are able to perform “fruitful” traversal of the solution space within reasonable time. This could lead to more successful negotiations and as such to a higher success rate in establishing SLAs. From the perspective of time and space complexity, it remains to be assessed whether a feasible solution to our problem exists at all. In addition, in the future it might be possible to choose an algorithm for planning and optimization during run-time depending on the specific problem. Criteria for choosing one algorithm over another may include the number of dependent sub-services, the number of potential sub-services, the nature of dependency, the number of QoS terms involved and the specific negotiation parties.

### 7.2 Infrastructure Planning and Optimization Component

The infrastructure layer is the basis on which the project’s multi-layer SLA hierarchy for IT services is established. The infrastructure layer will typically compare the infrastructure resource requests within incoming SLA requests, with the actual infrastructure resources, and then make an optimal plan on what resources to commit to this SLA.
The Infrastructure Planning and Optimization Component (IPOC) receives requests for infrastructure, queries the infrastructure service manager for potential provisioning solutions, selects and reserves the optimal one and requests the Infrastructure Provisioning and Adjustment Component (IPAC) to provision the selected plan as appropriate. If local resources cannot satisfy the request (e.g. due to lack of availability or specification discrepancies), the infrastructure planning and optimization component can attempt to outsource to third party providers, in order to satisfy the request.

More specifically, the core ideas of the infrastructure planning and optimization component are to achieve a minimum acceptable profit, a maximum acceptable failure risk, and try to minimize implementation costs.

### 7.2.1 Functionality

**Negotiation**

![Sequence diagram of negotiation phase](image)

**Figure 31: Sequence diagram of negotiation phase**

Figure 31 illustrates the negotiation phase for infrastructure services. The IPOC receives the SLA template, in which the information regarding the necessary infrastructure resources is defined. Then it queries the monitoring manager to
assess the monitorability for the candidate SLA, and also queries the infrastructure service manager about the quantity of resources, according to which an optimization strategy will be executed. If the service provider can supply the demand, it will reserve the required resources before sending an acceptance message back to the protocol engine, otherwise, a revised SLA template will be sent back without any reservations. This negotiation template may include the maximum available amount of resources, for the resource type requested by the user. Depending on policies, a provider may also simply reject the request without providing further information.

**Provisioning**

![Sequence diagram for provisioning request](image)

Figure 32: Sequence diagram for provisioning request

Figure 32 illustrates the provisioning phase of the IPOC. In order to accommodate both native SLA negotiation, as well as out-of-band negotiation (i.e. SLAs are negotiated outside the system and then simply inserted into it), we have designed the IPOC such that the provisioning notifications/requests are always coming from the SLA registry (via invocation of the IPOC’s iNotification interface). Following that, the IPOC will create a plan based on the content of the incoming SLA, and then it will invoke the planExecution interface of the IPAC to let the provisioning plan finally be executed.

### 7.2.2 Architecture and implementation

In this section we introduce the generic architecture of the whole Infrastructure SLA Manager (ISLAM), from which all the components and their relationship are represented clearly.
**Generic architecture**

**Figure 33: Component diagram of Infrastructure SLAM**

Figure 33 depicts the interactions between the IPOC and the other components inside and outside the infrastructure SLA manager.

- **assess/customize** is triggered by the Protocol Engine. The interaction is used to forward negotiation and runtime payload to the planning and optimization component, for reasons of decision-making and synchronization. The offer is assessed and either accepted, rejected outright, or customized and returned as a counter-offer.

- **track/query** is triggered by the Business Manager for manipulation and observation of SLA provisioning to support business aspects such as cost.

- **plan** illustrates the invocation of the IPAC for initiating a (re) provisioning.

- The IPOC can trigger the (re)negotiation of any SLA, using the **trigger_(re)negotiation** interaction.

- **check_monitorability** is used to verify whether a service is monitorable or not with respect to an SLA; this invokes the Monitoring Manager.

- Finally, it is possible to trigger **prepare_infrastructure_service** for the reservation of candidate infrastructure deployments.
Figure 34: Detailed IPOC diagram

Figure 34 depicts the detailed core components of the Infrastructure Planning and Optimization Component, and how they interact with other components.

Problem Model

Before we elaborate on the core methods of the IPOC, let us assume that the service provider supplies some resource types each bearing some characteristics. Those characteristics refer to inherent resource details, e.g. location of resource, clock speed for CPU cores, etc. In addition, that service provider has certain number of sub-providers who might become its service providers.

Figure 35: Multi-domain resource provisioning

It might be the case that service providers must turn to external entities and competitors if they run out of resources (Figure 35), so that they do not lose the customer – even if that may compress their profit margins for the specific contract.

The method implemented in the IPOC focuses on three key parameters:
• Implementation costs
• Profit
• Failure probability

Looking at implementation costs and the provider’s profit, we consider the following aspects:

• We separate the whole implementation cost to be: internal cost (i.e. resources utilized internally, $C_i$) and external cost (i.e. sub-contracted resources, $C_E$).

• We assume price reductions (discounts) for bulk purchase; the more a customer buys, the larger the discount is.

• Given a standard (baseline) Quality of Service (QoS), like availability, isolation and so on, customers pay more as they deviate from this baseline, minimum QoS.

• We introduce a variable (hereafter referred to as “beta”), as a quality multiplier that affects the internal cost. It indicates the provider’s dynamic policy with regard to the additional measures to take, in order to safeguard the respective guaranteed quality of a certain specific SLA (and consequently, improve its reputation by having less SLAs failing).

Overall, it is not sensible to charge the customers for additional quality (than what they originally requested), only because it is in the provider’s best interest. Thus, the provider will have to compress its originally targeted profit, $F$, by subtracting this extra cost:

$$F^i = g^i(C_i^i, C_E^i) - \beta^i \cdot C_i^i$$

$$F = \sum_i F^i$$

**Figure 36: Final profit**

The first formula in Figure 36 indicates that the final profit for offering a specific resource $i$ is equal to the initial projected profit (as a function $g$ of the total implementation costs), minus the extra costs assumed by the provider to further safeguard the eventual QoS. The second formula simply sums up to calculate the total profit.

As regards the failure rate of the service provider (which we assume to be an indication of the probability that future SLAs may fail), we assume that this will be affected over time as the provider strives to achieve better service (perhaps by accepting some profit loss). We accept by default that a better solution is also a more expensive solution; and that a more expensive solution is at least as good as the cheaper ones. Therefore, as $\beta$ increases, the cost of implementation also increases (or, at the very least, does not decrease). If reaching the requested QoS for a resource $i$ demands –according to some model– a factor of $\beta$, then using any larger factor in the calculations would mean increased cost, but also perhaps further increased quality. In other words, it would be less probable that the requested QoS will not be met and that the SLA will be violated.
\[ P_V^i = h^i(\beta^i, T^i) \]
\[ \frac{dP_V^i}{d\beta^i} \leq 0 \]

**Figure 37: Failure rate**

From Figure 37, \( T^i \) is a measure of reputation of the subcontractors involved in the delivery of resource \( i \) for the respective SLA. \( P_V^i \) is the probability that a SLA will be violated due to \( i \) failing to deliver. So we can say, the higher \( \beta^i \), the lower the probability that a SLA will be violated due to resource \( i \). Let us suppose that we have a maximal acceptable \( P_v^i \) for a specific resource type, according to provider-specific policies. Figure 38 provides the problem definition as a cost minimization problem, given constraints for profit and failure probability.

\[ \sum_i (1 + \beta^i) \cdot N^i \cdot \sigma^i \cdot C_{B}^i + C_{E}^i \]
\[ \sum_i [g^i(C_{E}^i - \beta^i \cdot C_{B}^i)] \geq F^* \]
\[ h^i(\beta^i, T^i) \leq P_V^{i*} \]
\[ 0 \leq \beta^i, \forall i \]

**Figure 38: Problem definition**

The first formula represents total internal and external costs (\( N^i \) represents resource quantity, and \( \sigma^i \) represents a multiplier for implementation costs, depending on the additional requested QoS and the possible mass-purchase discounts).

The increase or decrease of the quality multiplier \( \beta^i \) for the resources may affect profit and failure probability in converse ways. Higher quality means higher costs, lower profits, and less chances to fail. What we need to achieve is to find the lowest possible quality multipliers according to risk management policies, the highest possible according to profitability policies, and confirm they are not excluding each other. Then, the lower values can be used to compute the cost of implementing the solution.

With regard to subcontracting, the first step to execute is to see whether there is a part (perhaps all) of the requested resources in the incoming SLA request that the provider cannot offer. If so, the main service provider has to somehow get the resources from its sub-contractors. We need to take into account the resource constraints of candidate sub-contractors. According to the SLA templates they publish, the provider can see whether they offer the resources we need, so that it can contact them with a new SLA request. We will first consider the sub-contractors who have enough requested resources and then consider their combination if none of them has enough resources. For the selection of the sub-contractors, another two important factors are taken into account, which are price of the corresponding resource and the subcontractor’s reputation (essentially provided by means of their prior failure rates).
We will employ the scalarization technique of ideal point [23], where we are measuring a point’s distance from what would be an ideal combination for cost and failure ratio (Figure 39). Clearly, that would be the point (0; 0), i.e. perfect service given for free.

**Implementation**

![UML class diagram of IPOC's algorithm](image)

As shown in Figure 40, resources can be either CPUs or memory with different properties. Each request contains all the infrastructure resources information for the incoming SLA requests, like CPU configuration and number, memory size, location, image of the virtual machine, Quality of Service (availability and isolation) and so on.

A site (provider) is represented by a request processor – that is, an instance of the algorithm’s implementation as part of an IPOC’s functionality. It handles one or more requests and interacts with the Infrastructure Service Manager (ISM) for querying and reserving the resources. Based on the information within the request, the site minimizes implementation and outsourcing costs for reasons of competitiveness, while respecting business policies for profit and risk.

An algorithm for outsourcing is designed and implemented, using cost and subcontractor reputation as selection criteria; and local resource configurations as
a constraint satisfaction problem for acceptable profit and failure risks. Thus, it becomes possible to provide educated price quotes to customers and establish safe electronic contracts automatically.

**Simulation Result**

To evaluate the model with regard to its validity, we established a simulation scenario with specific functions for increased quality costs, profit, failure probability, and so on. Resources under negotiation are CPU cores and storage. CPU cores are offered in 4 different combinations of clock speed (1GHz or 2GHz) and volatile memory (1Gb or 2Gb). Storage is offered in arbitrary quantities, in increments of 1Gb. Their negotiated qualitative characteristics are availability, measured as a percentage of time, and isolation, indicating that the customer’s virtual resources have exclusive access to the physical infrastructure that implements them (e.g. a blade server).

CPU core prices are given by Table 3 in the form of normal distributions (identified by the mean value and variance). Similarly, the price for storage is 2±0.5 per Gb. In each simulation run, each provider is assigned resource price values at random, from these distributions.

<table>
<thead>
<tr>
<th>Table 3: CPU core prices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>1Gb</td>
</tr>
<tr>
<td>2Gb</td>
</tr>
</tbody>
</table>

Price reductions are given as a stepwise function ranging between 0% and 20%, in steps of 5% at resource quantities 15, 50, 150 and 500. We do not distinguish between the types of resources; rather we apply the same function both to CPU cores and storage. The increased costs for higher quality are a 50% additional cost for isolation (which can only be true or false), and 10% for each additional unit of availability. Baseline availability is 95%, and maximum is 99%. Default value for isolation is false. We use the same function for both resource types. Profit will be the sum of 30% of internal cost and 5% of external cost. That is, the provider also makes a very small profit from outsourced resources. Therefore, price quotes to the customer are provided as internal cost plus external cost minus applicable price reductions. An initial (artificial) past failure rate is selected to be 20%. Random SLA violations are introduced, in a rate always in accordance with the site’s failure rate for each resource. Failure frequency is then further controlled by the selected values for \( \beta \); each time we choose such a value, we modify the failure rate to reflect these extra measures we take to safeguard the SLAs. The minimum profit depends on the customer. We have three customer classes; namely, Gold, Silver and Bronze (Table 4).

<table>
<thead>
<tr>
<th>Table 4: Discounts based on customer type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>discount</td>
</tr>
</tbody>
</table>
We are allowing additional resources (and hence, lower profit) even for bronze customers, as we wish to improve the reputation of the provider. Our target is to reach a failure rate equal to 5% or less, given the small gradual effect of “beta” on the future violations.

Figure 41: Experimental results

Figure 41 illustrates the most important results of our experiments. The top four plots show the available resources over time, for the main site and the three subcontractors. The utilization (preference) of subcontractors is indeed aligned with their prices and reputation. We can see that as soon as the main site’s resources become depleted, it is mostly the 3rd subcontractor that is being utilized, as it is the “best” from a price and failure rate point of view. Following, the 2nd and the 1st are utilized when the 3rd has no more available resources. The bottom two graphs illustrate the accumulated profit over time for the main site, and the values for its failure rate. We can see that the profit keeps increasing even when there are few resources available and the site has to outsource. Also, that the site’s reputation (failure rate) is improving significantly over time, starting with an artificial failure rate of 20%, but eventually reaching and surpassing the target of 5%. This decrease of failures was simulated by modifying the initial probability of a failure event, according to the values $\beta$ was taking over the simulation time.
7.3  Formalization of SLA translation

Translation is a term with different views and interpretations in different domains (e.g. networking, performance engineering), various methodologies (queuing theory, reliability engineering), topic areas (communication systems, healthcare), or underlying architecture style (e.g. Client-Server, Peer-to-Peer, Service Oriented). For example, in networking, policy translation is defined as “the transformation of a policy from a representation and/or level of abstraction, to another representation or level of abstraction”.

Theilmann et al. define "SLA translation as any form of transformation of metrics and parameters, within one layer or from one (sub)layer to another in a multi-layered SOA environment” [24]. This definition opens a broader perspective towards SLA translation problems in SOA.

However, the definition focuses on SLA Translation for layered Service Oriented Environments where SLAs are stepwise translated, either from top-level layers to bottom-level layers or the other way around. We refer to these kinds of SLA translations as vertical SLA translations. In an ideal layered architecture, layer N+1 never communicates directly with layer N-1. All information are passed through layer N. This implies that layer N+1 SLAs are translated into layer N SLAs, and never directly into layer N-1 SLAs. Of course this an ideal engineering design paradigm.

The implementation of real world layered SOAs often bypasses this design restriction due to performance issues or to decrease the coding effort. The layered dependencies are broken up and take the structure of a direct dependency graph. Likewise, the SLA dependencies form a graph structure and the stepwise vertical SLA translation becomes more challenging. Moreover, dependencies among top-level services exposed to the internet do not necessarily form a layered structure. They rather form a complex directed graph structure. We call both, the vertical as well as the horizontal dependency graph, the service dependency graph.

The research question we state is:

Is it possible to define and formalise the SLA translation problem in a generic way such that (a) a general approach to express the needs of various different domains, methodologies, areas and engineering aspects is provided, and (b) directed graph organised architectures are encompassed?

The Service Dependency Graph, the derived Property Graph and SLA Translation are discussed in the following sections.

7.3.1  Service Dependency Graph and Property Dependency Graph

The assumption here is that a service’s logic (the functionality it delivers) depends on some other service by means of composition. If we assume that a service implements an algorithm, and that an algorithm is represented by a series of operations (instructions), then one or more of those instructions are carried out by the services on which this former service depends on. These are explicit dependencies. Implicit dependencies are, typically, dependencies related to infrastructure services (without which a higher-layer service cannot operate at all), and services used for reasons of redundancy. For instance, any web server presumes that a working DNS system is available so that clients can access it. Any software relies on some hardware on which it can execute. The notion of explicit and implicit dependency itself is enough to construct a dependency graph as a first step. This dependency graph can then be elaborated in the context of
SLAs, as we will see in the upcoming sections, and function as a basis for the translation process.

We define a SLA as a set of guarantees over the consumption of a service. For instance availability could be property of a service. A threshold for this availability property in the form of a guarantee towards customers constitutes one part of a SLA. Dependency of a service upon others means that its properties are depending on properties of these other services.

Assuming the availability example for a depending service, it will be affected by the availability of its antecedent services. For example the availability of a WebService depends on the availability of the Application Server that executes the WebService. However, there might be a case where a dependent service’s property may be related to a completely different property of its antecedents. That can be best explained in the context of our example if we define availability for the web service as:

\[
\text{Availability (WebService)} := \frac{\text{Request Served}}{\text{Request Issued}}
\]

and the availability of the Application Server that hosts and executes the WebService as:

\[
\text{Availability (Application Server)} := \frac{(\text{Uptime} - \text{Downtime})}{\text{Uptime}}
\]

Due to some timeout constraints (e.g. in the TCP/IP protocol stack) a request might be discarded if the overloaded Application Server is not able to serve a request in time. In that case the WebService appears as unavailable although the Application Server itself is available (up). Hence, the Availability of WebService does not only depend on the availability of the Application Server, it also depends on the throughput property of the Application Server. The throughput property of the Application Server itself depends on the configuration of the Application Server (e.g. size of thread pool), other sub-systems properties (e.g. the Kernel TCP configuration) and computing capabilities of the host infrastructure (e.g. the CPU speed).

The Service Dependency Graph for our example is given in Figure 42. It has to be noted that this example fits into the SLA Translation Classification proposed by the project in [25], as we have Metric to Metric and Metric to Configuration dependencies, which have to be translated between services.

**Figure 42: Example Service Dependency Graph**

Due to the vast amount of service properties and potential dependencies it is clear that there cannot be a universal classification of all service properties, and their dependencies. We can only define the problem in a generic manner based on the abstraction of conditions that need to hold true. Hence, we define the Property Dependency Graph (PDG), a refinement of the Service Dependency Graph (SDG).
The Service Dependency Graph is used to capture dependencies between Services whereas the Property Dependency Graph refines a Service Dependency Graph and maps properties dependencies among Services, as depicted in Figure 43. In this example, $Sp$, $Sq$ and $Sr$ are Services whereas $p1$, $p2$ and $p3$ are properties of service $Sp$. The properties $q1$, $q2$, $q3$, $q4$ are properties of service $Sq$ and properties $r1$ and $r2$ are properties of service $Sr$.

Based on these generic data-structures we are now able to define a generic SLA Negotiation and Translation approach as described in the next section.

7.3.2  **SLA Negotiation and Translation**

We can now define *SLA translation* as the process of analyzing a SLA in relation to the PDG, applying heuristics and pre-existing knowledge, and coming up with one or more subsequent SLAs for antecedent services. These subsequent SLAs provide reasonable (but not complete) certainty that the top-level SLA will not be violated, unless some of them are violated too.

The algorithm to translate SLAs based on a PDG is depicted in Figure 44. Figure 44 also shows SLA Negotiation, which is an essential part of our framework architecture and closely related to SLA Translation.

Using the example from Figure 43, we have the following relations:

\[
\begin{align*}
p_1 &= f(q1, q3) \\
p_2 &= g(q3, r1) \\
p_3 &= h(q4, r2)
\end{align*}
\]

If an incoming SLA for service $Sp$ refers to all three properties, a SLA negotiation mechanism which includes translation functions should first find out the dependencies of these three properties based on the PDG. We assume that these dependencies are known as domain-specific expertise encoded into our system. For instance, if the property is “availability”, it will typically depend on the availability of all antecedents; this is fairly straight forward to assume. However, if we examine cost, it is not necessary that the cost of invoking service $Sp$ is related specifically to cost properties of services $Sq$ and $Sr$. On the contrary, it may be the case that there are no cost properties for these services, but rather that their providers only apply flat-rate pricing. In this case, the cost of $Sp$ may rely on properties such as Quality of Service characteristics of $Sq$ and $Sr$. 

---

**Figure 43: SDG (left) and PDG (right).**
The next action should be to find acceptable value spaces for all of \((q_1, q_3, q_4, r_1, r_2)\). By “acceptable”, we refer to values that remain within constraints set inside the templates of the two lower-level services, and at the same time satisfy the requested values in the offer for the higher-level SLA.

The last step during the translation process would be to come up with an optimal solution to the problem, according to a multi-criteria optimization algorithm, which perhaps takes into account business objectives such as profit maximization. It may well be the case that there exists no single optimal solution.

In this case, one of the solutions on the Pareto front [26] should be chosen. The Pareto front is a set of all those solutions that are considered to be optimal in multi-criteria optimization. More formally, they are solutions where none of the included criteria can be improved (accept a better value), without some other criteria in the same solution receiving a worse value.
7.3.3 Next steps

The proposed generic graph structures and the SLA Translation algorithm provide a skeleton approach that has to be adapted by each use-case in order to reflect use-case specific requirements and objectives. That is perfectly fine as it is clearly not possible to provide a universal solution for all SLA translation problems.

However, we realise that it might be possible to provide generic SLA translation solutions that can be applied to various use-cases with less (or even without) use-case specific implementation effort.

One example we are investigating in co-operation with WP A3 is to apply this work for IT Business Continuity Management. IT Business Continuity Management tries to (1) understand organisations’ business processes and associated IT services (2) identify threats and risk to IT services, IT operations and business processes and (3) determine responses to mitigate or overcome a possible disruptions. One strategy is to develop recovery plans. In case external services are used, BCM crafts Service Level Agreements that detail certain recovery times and penalties.

IT Business Continuity Management (IT-BCM) does need a holistic and comprehensive view on the complete IT service graph that supports business operations. This holistic view can be provided by the Service Dependency Graph. Furthermore IT Business Continuity Management is interested in time-bound dependencies and time-bound failure propagation of disruptions through the stack. Business Continuity Management translates BCM SLAs, such as the Maximal Tolerable Outage Time of a business process down to the Return Time Objective or Recovery Point Objective of a database. We aim to re-use the Property Dependency Graph as the underlying data-structure to express these time-bound relationships and we are investigating generic ways to analyse translate such SLAs for arbitrary service graphs. Also we aim to provide support for horizontal BCM SLA translation; that is support to detail response times and to determine penalties in case external services are utilised.

IT-BCM is a good candidate for a generic SLA translation solution as BCM applies the same BCM properties at all layers and considers service specific details in a very abstract manner.
8 Translation of SLAs into Monitoring Specifications

The monitoring of SLAs expressed in the common SLA model of SLA@SOI requires the translation of these SLAs into the specific monitoring specification languages of the different types of reasoning engines which may be plugged into the SLA@SOI framework. The general architecture of this framework supports the integration of different types of generic and/or special purpose reasoning engines, which may internally realise different monitoring approaches but externally support the same common interface. This interface enables reasoning engines to receive the guarantee terms of the SLAs that need to be monitored and report monitoring results back to the SLA@SOI framework. Due to differences in the languages that these reasoning engines may use in order to express operational monitoring specifications, it is not possible to devise a common translation scheme for all of them. Hence, in the rest of this section, we focus on the translation scheme that has been developed for EVEREST [32][33], chosen among the various alternatives as a complete framework, familiar to some of the consortium members.

8.1 Overall monitoring architecture

The purpose of the Advanced Monitoring System illustrated in Figure 45 is to complement the service manager and the SLAM, by providing the means to monitor a provisioned service instance in the context of the requested guarantee terms contained within the respective SLA.

The MonitoringManager (MM) coordinates the generation of a monitoring configuration of the system, as described in detail in Section 8.2 of [27]. It decides, for an SLA it receives, what is the most appropriate monitoring configuration according to configurable selection criteria. A monitoring configuration describes which components to configure and how their configurations can be used to obtain results for Guarantee Terms.

The Low Level Monitoring System (LLMS) is a central entity for storing and processing monitoring data. It collects raw observations, processes them, computes derived metrics, evaluates the rules, stores the history and offers all these data to other components (accessible through ServiceManager). It implements the monitoring part of ProvisioningRequest, containing constraint based rules (time and data driven evaluation) and ServiceInstance specific Sensor related configuration. It is generic by design, therefore capable of supporting infrastructure, software or other services.

The Sensor collects information from a service instance. The design and implementation of sensors is highly domain-specific. Sensors can be injected into the service instance, i.e. to achieve service instrumentation, or it can be outside the service instance intercepting service operation invocations. A sensor can send the collected information to the communication infrastructure or other components can request (query) information from it. There can be many types of sensors, depending on the kind of information that must be collected, but all of them should implement a common interface. The interface provides methods for starting, stopping, and configuring a Sensor.

The Effector is a component for configuring service instance behavior. The design and implementation of effectors is also domain-specific. An effector can be
injected into a service instance, to achieve service instrumentation, or it can provide a service configuration interface. There can be many kinds of effectors, depending on the service instance to be controlled, but all of them should implement a common interface, which should provide the methods to configure a service.

Figure 45: Overview of SLA@SOI’s monitoring system

An additional type of monitoring feature is a Reasoner Component Gateway (RCG). The RCG provides the interface for accessing a Reasoning Engine (also referred to as Reasoner); the latter performs a computation based upon a series of input data provided by the events or messages sent from a Sensor and/or Effector. An example RCG may be to retrieve the completion time, the latter accepting events from sensors detecting both request and responses to a service operation.

The subsequent sections will describe the part of the monitoring architecture of SLA@SOI which refers to the transformation and transfer of a high level SLA to
the RCG of individual reasoning engines. By processing a given SLA and a monitoring system configuration which assigns the monitoring of expressions taken from the given SLA to the related reasoning engine of the RCG, the RCG is able to generate operational monitoring specifications for the related reasoning engine.

Figure 46: Overall monitoring specification process

8.2 SLA translation for the EVEREST environment

8.2.1 Overview of the monitoring language of EVEREST

To be monitored by the EVEREST environment [32][33], SLA guarantee terms need to be expressed as monitoring rules and assumptions in the operational monitoring specification language of EVEREST, called EC-Assertion. EC-Assertion [32][33] is an XML language based on Event Calculus [31] (a first-order temporal logic language). The basic modelling constructs of Event Calculus (and EC-Assertion) are events and fluents.

An event in Event Calculus (EC henceforth) is something that occurs at a specific instance of time, is of instantaneous duration, and may cause some change in the state of the reality that is being modelled. This state is represented by fluents.

To represent the occurrence of an event, EC uses the predicate \( \text{Happens}(e, t, \text{H}(t_1, t_2)) \). This predicate represents the occurrence of an event \( e \) that occurs at some time point \( t \) within the time range \( \text{H}(t_1, t_2) \) and is of instantaneous duration. The boundaries of \( \text{H}(t_1, t_2) \) can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The EC predicate \( \text{Initiates}(e, f, t) \) signifies that a fluent \( f \) starts to
hold after the event e occurs at time t. The EC predicate \( \text{Terminates}(e, f, t) \)
signifies that a fluent \( f \) ceases to hold after the event \( e \) occurs at time \( t \). An EC formula may also use the predicates \( \text{Initially}(f) \) and \( \text{HoldsAt}(f, t) \) to signify that a fluent \( f \) holds at the start of the operation of a system and that \( f \) holds at time \( t \), respectively.

EC defines a set of axioms that can be used to determine when a fluent holds based on initiation and termination events that regard this fluent. These axioms are listed in Table 5. Axiom EC1 states that a fluent \( f \) is clipped (i.e., ceases to hold) within the time range from \( t1 \) to \( t2 \), if an event \( e \) occurs at some time point \( t \) within this range and \( e \) terminates \( f \). Axiom EC2 states that a fluent \( f \) holds at time \( t \), if it held at time 0 and has not been terminated between 0 and \( t \). Axiom EC3 states that a fluent \( f \) holds at time \( t \), if an event \( e \) has occurred at some time point \( t1 \) before \( t \), which initiated \( f \) at \( t1 \) and \( f \) has not been clipped between \( t1 \) and \( t \). Finally, axiom EC4 states that the time range in a \( \text{Happens} \) predicate includes its boundaries.

**Table 5: Axioms of Event Calculus**

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(EC1)</td>
<td>( \text{Clipped}(t1,f,t2) \iff (\exists e,t) \text{Happens}(e,t,\Re(t1,t2)) \land \text{Terminates}(e,f,t) )</td>
</tr>
<tr>
<td>(EC2)</td>
<td>( \text{HoldsAt}(f,t) \iff \text{Initially}(f) \land \neg \text{Clipped}(0,f,t) )</td>
</tr>
<tr>
<td>(EC3)</td>
<td>( \text{HoldsAt}(f,t) \iff (\exists e,t1) \text{Happens}(e,t,\Re(t1,t)) \land \text{Initiates}(e,f,t1) \land \neg \text{Clipped}(t1,f,t) )</td>
</tr>
<tr>
<td>(EC4)</td>
<td>( \text{Happens}(e,t,\Re(t1,t2)) \implies (t1 &lt; t) \land (t \leq t2) )</td>
</tr>
</tbody>
</table>

**EC-Assertion** adopts the basic representation principles of EC and its axiomatic foundation and introduces special terms to represent the types of events and conditions that are needed for runtime monitoring. More specifically, given its focus on monitoring the operation of software systems at runtime, events in EC-Assertion can be invocations of system operations, responses from such operations, or exchanges of messages between different system components. To represent these types of events, EC-Assertion defines a specific event structure that is syntactically represented by the event term

\[
event(_\text{id}, _\text{sender}, _\text{receiver}, _\text{status}, _\text{sig}, _\text{source})
\]

In this event term:
- \( _\text{id} \) is a unique identifier of the event;
- \( _\text{sender} \) is the identifier of the system component that sends the message/operation call/response;
- \( _\text{receiver} \) is the identifier of the system component that receives the message/operation call/response;
- \( _\text{status} \) is the processing status of an event (i.e., \( \text{REQ} \) if the event represents an operation invocation and \( \text{RES} \) if the event represents an operation response);
- \( _\text{sig} \) is the signature of the dispatched message or the operation invocation/response that is represented by the event, comprising the operation name and its arguments/result;
- \( _\text{source} \) is the identifier of the component where the event was captured.

Fluents are defined as relations between objects and represented as terms of the form \( \text{rel}(O_1, ..., O_n) \). In fluent terms, \( \text{rel} \) is the name of a relation which associates the objects \( O_1, ..., O_n \).
The assumptions and rules to be monitored at runtime are specified in terms of the above predicates and have the general form \( \text{body} \Rightarrow \text{head} \). The meaning of a rule is that if its \( \text{body} \) evaluates to \( \text{True} \), its \( \text{head} \) must also evaluate to \( \text{True} \). The \( \text{Happens} \) predicates in a rule with no constraints for their lower and upper time boundaries are what we call "unconstrained" predicates. During the monitoring process, rules are activated by events that can be unified with the unconstrained \( \text{Happens} \) predicates in them. When this unification is possible, the monitor generates a rule instance to represent the partially unified rule and keeps this instance active until all the other predicates in it have been successfully unified with events and fluents of appropriate types or it is deduced that no further unifications are possible. In the latter case, the rule instance is deleted. When a rule instance is fully unified, the monitor checks if the particular instantiation that it expresses is satisfied.

To enable the specification of the different types of SLA guarantee terms that have been identified in SLA@SOI, \textit{EC-Assertion} has been extended to offer support for the following:

- Support for aggregate functions as fluents arguments. The \textit{EVEREST} implementation supports a generic type of fluent whose arguments can be arbitrarily constants, variables and/or functions.
- Constants and time variables are now supported as Function arguments. Constants are translated to variables with respect to constant types, while time variables are translated to long variables.
- Constants, variables and time variables are supported as relational predicates operands
- \( \text{HoldsAt, Initiates} \) and \( \text{Terminates} \) predicates support time expressions instead of plain time variable.

### 8.2.2 \textit{EVEREST} templates

This section will describe the scheme for specifying \textit{EVEREST} monitoring templates, i.e. sets of parametric monitoring EC formulas; and will give examples of templates for SLA guarantee terms in the abstract syntax model.

For each QoS Term that \textit{EVEREST} sensors support, there is a parametric \textit{EVEREST} monitoring template which includes a set of assumptions, used for the QoS Term computation and monitoring. At the time being, \textit{EVEREST} supports the following QoS Terms: \textit{Throughput}, \textit{MTTR}, \textit{MTTF}, and \textit{Reliability}. In the following, we present the parametric \textit{EVEREST} monitoring template for \textit{Throughput}.

\textit{Throughput} is the number of consecutive service operation calls that are served before this call cannot be served anymore, i.e. it is dropped. For keeping track of the served operation calls of a service, the fluent \( \text{Served} \) is specified having as parameters the id of the throughput period—referring to the period between two consecutive dropped operation calls of the monitored service \( \_P \)—, the unique id of the monitored service \( \_SrvId \), the number of consecutive service operation calls that are served during this period \( \_N \), and the starting time point of the aforementioned period \( \_t \). By processing the formulas in Figure 46, \textit{EVEREST} manages (by means of initiation and termination) instances of the fluent \( \text{Served} \) whenever an operation call of the monitored service is served or dropped.

To reason whether the operation calls of the monitored service are served or dropped, \textit{EVEREST} should receive primitive call and response events from the sensor that has been selected by the Monitoring Manager to provide such events for the particular service. Therefore, a call to the monitored service that occurred
at $t$ is considered served if a corresponding response occurs within a predefined time range between $t$ and $t+d$.

Finally, EVEREST generates and instantiates the fluent $\text{Throughput}(\text{ThroughputValue})$, which is specified having as a parameter the variable $\text{ThroughputValue}$, whenever an operation call of the monitored service is dropped. It should be noted that the value of $\text{ThroughputValue}$ variable equals to the number of consecutive service operation calls that are served within the current throughput period ($_N$). The assumption formulas that are used for monitoring the Throughput of a monitored service are included in Appendix D.

The EVEREST monitoring specifications included in Appendix D are described in an ad-hoc language which is a combination of the EC-Assertion language used for the monitoring rules and assumptions, and of an informal language that introduces the template parameters. Examples of the latter are the monitored service id $<_\text{SrvId}>$ or the unique case id $<_\text{CaseId}>$ that the template is selected for and which refers to the unique ids of the SLA and SLA objects that were assigned to EVEREST. The automation of the process of EVEREST template selection and instantiation makes it necessary to describe the templates in a formal language and, for this purpose, we re-specify the supported QoS terms templates using a language that is a combination of EC and the Formal Template Language (FTL) [29][22]. FTL is used to express the monitoring templates, as well as to formalise the generation of monitoring specifications from the templates given an instantiation. By “instantiation” we refer to a mapping of the template parameters to specific modelling elements, which in our case are defined by the SLA and SLA Template Abstract Syntax (see Section 4.1.2).

A brief introduction of FTL, as well as a detailed description of the EVEREST monitoring specifications generation process using EC aware FTL templates is provided in [30]. It should be noted that the only FTL constructs that are used for re-specifying the informal parametric EC monitoring templates to EC aware FTL templates are placeholders and template definitions, as presented in Section 3.2.1 in [30].

**Figure 47: EVEREST RCG Translator using EC-aware FTL templates**
In Appendix E the definition of EC aware FTL template for Throughput is shown. Having introduced the EVEREST monitoring specifications generation process using EC-aware FTL templates, an overview of the EVEREST RCG translation component is presented in Figure 47.

In particular, the EVEREST RCG Translator consists of the following components: the Parser, the ASTTranslator, and the Instantiator. The Parser transforms the monitorable SLA expressions, which are assigned to EVEREST by the Monitoring Manager and are specified as Java objects (representing the SLA@SOI instance of the abstract syntax) in monitoring system configuration specifications, to abstract syntax trees (ASTs). The ASTTranslator processes the parsed ASTs and selects the appropriate FTL templates from the corresponding repository (FTL Templates Repository). Finally, by taking into account the monitored SLA, as well as, the monitorable SLA expressions that are assigned to EVEREST, the Instantiator generates operational EVEREST monitoring specifications from the selected FTL templates.

8.2.3 SLA Parsing

This section will describe the mechanism for parsing the SLA (encoded as a Java object) that is passed to the RCG of the EVEREST reasoning engine, into an abstract syntax tree which is then used to generate the actual EVEREST monitoring rules and assumptions. The latter constitute the EVEREST monitoring specification.

The algorithm Parse included in Appendix F takes as input any java object that refers to the following constructs of the SLA & SLA Template Model: AgreementTerm, GuaranteedState, and ConstraintExpr. The output of algorithm Parse is an abstract syntax tree (AST henceforth) that is used for the selection and instantiation of the parametric EC aware FTL templates.

Regarding the parsing of the aforementioned SLA constructs, for each AgreementTerm a node labelled as IMPLIES is created. An IMPLIES node is a binary node, i.e., it has two children. It should be noted that algorithm Parse defines the relation between the Precondition and the Guaranteed States of an AgreementTerm as implication, while the relation between Guaranteed States of a term is defined conjunctively. As a result, if the Precondition of a term holds, then all Guaranteed States should be met. Therefore, the left child represents the Precondition of the corresponding AgreementTerm, if any. The right child is a node labelled as AND node. The right child represents the conjunction of the Guaranteed States of the “parent” AgreementTerm.

Similarly, for representing a GuaranteedState, the algorithm generates a binary node labelled as IMPLIES. The algorithm defines the relation between the Precondition and the State of a GuaranteedState as implication. As a result, if the Precondition of a GuaranteedState holds, then the State should be met. Therefore, the left child represents the Precondition of the corresponding GuaranteedState, if any, while, the right child represents the actual State of the GuaranteedState.

According to [29], Preconditions and States are defined as ConstraintExpr. The algorithm Parse takes into account whether a ConstraintExpr is a CompoundConstraintExpr, which is used for defining logical sub expressions, or a TypeConstraintExpr, which is used for defining comparison expressions, and generates the corresponding sub trees. It should be noted that the sub trees of a ConstraintExpr node end up in leaf nodes that represent QoS Terms or Constants. To further illustrate the parsing output of the Parse algorithm, assume that the following agreement term is assigned to EVEREST by the Monitoring Manager:
AT1:

\[ \text{AT1.Precondition: Throughput} \geq T \]

\[ \text{AT1.Guaranteed.States} = \{\text{MTTR} < M\} \]

Given the corresponding Java object that specifies the agreement term AT1 to the algorithm Parse, the tree illustrated in Figure 48 is generated.

![Figure 48: AST for agreement term at1](image)

### 8.2.4 Selection of parametric templates and generation of operational monitoring specifications

This section describes the mechanisms for selecting and producing the EVEREST formulas required for generating an operational monitoring specification from an abstract syntax tree.

The algorithm Translate presented in Appendix G is used for the selection of the parametric FTL templates that must be used for the monitoring of the assigned SLA expressions. The algorithm Translate takes as input the AST of the parsed SLA expression that is assigned toEVEREST. By traversing recursively the given AST, the algorithm compiles a set of parametric FTL templates (ECFormulasList). The compiled list contains templates for each QoS Term that the assigned SLA expression includes, as well as, a parametric EC-Assertion monitoring rule (monitoringRule) that is necessary for the runtime check of the assigned SLA expression as a whole. It should be noted that the EC monitoring rule which is built up on the fly, as well as all the formulas that EVEREST checks, are specified as body $\Rightarrow$ head, where body and head are conjunctions of EC-Assertion predicates. Therefore, the monitoring rule construction is undertaken by
analyzing the visited nodes and adding corresponding predicates in the conjunctions of monitoring rule body and head.

Appendix G provides a complete account of the algorithm and an example of its application (for a Throughput guarantee) in full technical detail.

8.3 Limitations and Future Work

EVEREST RCG currently supports the transformation of a subset of SLA and SLA(T) abstract syntax constructs into operational EVEREST monitoring specifications. The EVEREST RCG implementation is still under development, and therefore there currently exist some limitations which will be addressed in Y3:

- Guaranteed Actions, i.e., a set of actions that one of the parties to the SLA is obligated to perform, are not supported by the translation process.
- The periodic event class, which is used in SLA(T)s to denote the "trigger" conditions in time-series functions and guaranteed actions, is currently not supported neither by the translation process nor by the EVEREST reasoning engine.

Future work will focus on the support of an extended specification for SLA models, supporting periodic events and Guaranteed Actions translation.
9 **Provisioning and Adjustment**

SLAs provide the service consumer with some level of confidence that the services delivered by the provider will operate within acceptable bounds, particularly regarding the non-functional properties and QoS attributes. Aside from the Service Provider being able to deliver highly available, reliable systems, just being able to deliver the service in the promised conditions is key to success and is increasingly becoming a competitive requirement. Moreover, most targets set in a Service Level Agreement are subject to direct financial penalties or indirect financial repercussions if not met. And apart from the direct economical impact, it is worth mentioning the influence of the SLA fulfilment on improved user perception (quality of experience).

For these reasons, the provision and adjustment of services becomes a fundamental part in a transparent SLA management system. SLA Enforcement ensures continual identification, monitoring and reviewing of the optimally agreed levels of services as required by the business, and ensures that given SLAs are fulfilled such that previous estimates on performance should be equal to the final performance of the running services. To this end, the system dynamically readjusts itself in cases that service levels cannot be met.

Because of the possible penalties when a malfunctioning of a service occurs, it is critical for this management process to flag when service levels are projected to be violated in order for the service provider to take actions to address the issue. To this extent, the adjustment module must detect any deviation from the expected standard operation of a service that causes, or may cause an interruption to the normal operation, or a reduction in the quality of the service. The objective is to provide continuity by restoring the service in the quickest way possible.

### 9.1 Requirements

This section briefly summarizes the (implicit, and explicit) requirements for the Provisioning and Adjustment functionality. Implicit requirements derive from the formal description of provisioning and adjustment in the DoW, while explicit requirements have been documented by the various use cases.

Following the description from the DoW, the Provisioning functionality must:

- Be dynamic and automated;
- Be SLA-aware;
- Support highly dynamic and scalable service consumption; and
- Be well-coordinated as it affects software and infrastructure levels, and only the whole set will deliver the intended business functionality.

From the same text, we deduce that the Adjustment component must:

- Facilitate the fulfilment or enforcement of the SLA properties;
- Be able to react to available monitoring data and the output of the service provisioning;
- Be able to adjust software service operations, based on the collection of SOA monitoring data;
- Be able to ultimately enforce SLAs or at least automatically adjust them at the lowest possible level, namely the physical infrastructure;
• Support SLA enforcement at the infrastructure layer, by triggering resource management policies (resource migration, landscape reconfiguration), and by signalling violations to the application service level;

• Decide on which level an exception should be handled, and escalate decisions to higher SLA levels. These processes may be domain and application specific: while some issues can be resolved on the lower levels (e.g. by forcing dynamic resource reallocation on the SOI, substituting a service or increasing the service level), other issues might include global re-planning; and

• Use policies as a guiding principle for influencing decisions and actions.

Conversely, the use cases documented their requirements, which later on have been mapped to features from the A-line work-packages:

• Service Providers must be selectable on the basis of both functional and QoS parameters;

• A Service Consumer in concertation with a Service Provider must be able to design orchestration/composition rules;

• It must be possible to dynamically provision and manage physical and virtual machines as well as resource bundles / data storage and software components;

• It must be possible to start SLA enactment immediately after negotiation or at a future time;

• Infrastructures resources must be re-allocated in response to load variance;

• Dynamic / unforeseen re-allocations should be possible;

• The decisions about who reacts to SLA violations and how they do so, must be configurable;

• Dynamic and automatic selection of (component) service (registered) providers must be supported;

• Service parameters (at the very least response time and execution time) must be automatically adjusted to restore SLA compliance;

• Resources for services must be automatically adapted based on SLA constraints; and

• The framework must autonomously detect SLA status (value of observables/SLOs and penalty/rewards).

### 9.2 PAC within SLA@SOI Architecture

In the actual architecture of SLA@SOI, the tasks described in previous subsections rely heavily on the Provisioning and Adjustment Component (PAC). The PAC, being part of the SLAM, is responsible for executing the provisioning plans supplied by the Planning and Optimization Component (POC) as described in Chapter 7. The actual execution of plans can be realized by posting requests to the Service Manager, where requests include provisioning and decommissioning of services (both software instances and infrastructure resources).

The PAC subscribes to an event bus (Monitoring Event Channel) in order to receive two type of events: first, at provisioning time, events from the Manageability Agents governing the deployed services, informing about the actual status of provisioning; second, events from the monitoring system indicating that
a SLA has been violated; and finally, events from the prediction components, alerting that a SLA will soon be violated unless appropriate actions are taken. Upon the receipt of these events, the PAC may behave differently: it will take some decision about provisioning if the message comes from a Manageability Agent; and in case of monitoring and prediction events, it may take an action on its own given sufficient certainty about the problem and its solution or it may request the POC to create a new plan, or even to re-negotiate an existing SLA.

To this end, the Provisioning and Adjustment Component offers the <<plan>>, <<control>> and <<query>> interactions, and uses <<query>> interface of the SLA registry, <<subscribe_to_event>> of the Monitoring Event Channel, and the <<manage_T_services>> interface offered by the Service Managers. The later is by definition domain specific, and derives in <<manage_software_services>> at software level and <<manage_infrastructure_services>> at the infrastructure layer.

The <<plan>> interaction subsumes the functionality that is necessary to request the execution of a plan, as well as to inform about the status of the execution of the plan back to the POC. The <<plan>> interaction is partitioned among the following interfaces:

**IPlanExecution**

This interface sends information about a plan, and an associated action: whether to execute it, or to ignore the monitoring events related to a previous plan. It is composed by the following methods:

- **executePlan** triggers the execution of a plan. The plan is provided as an argument, and it contains a number of tasks to be executed by the PAC.
- **cancelExecution** cancels the execution of a plan. The already provisioned resources will be released.
- **ongoingAction** is used to inform the PAC that a corrective action is already being taken at a lower level. The lower level component that is in charge of solving the problem sends a notification to higher level components through the iReporting interface of the POC. Therefore, the higher-level POC must inform the PAC, and instructs it to ignore the violation events corresponding to the same SLA.

**IQueryPlanStatus**

The IQueryPlanStatus interface is composed of one method:

- **getPlanStatus**, returns the status of the execution of a given plan

The <<control>> interaction has been designed to take business level criteria into account at the lower levels. For instance, the service provider can decide that it is better to accept violations in some SLAs to give priority to others based on business impact. It is even possible that the service provider decides to prevent breaches for some SLAs or specific QoS metrics while applying reaction upon violations for others. To this end, the <<control>> interaction allows the Business Manager to retrieve the current adjustment policies, as well as to set a new list of policies. The corresponding interface is:

**IControl**

The Provisioning and Adjustment Component exposes the “IControl” interface with the following methods:

- **setPolicies**: Set the list of policies for the adjustment functionality.
• **getPolicies**: Fetch and return the list of policies.

Finally, as the PAC is the proxy between the SLA core and the Monitoring System, the **<<query>>** interaction allows an external entity (namely the Business Manager) to retrieve information about SLA violations as well as the historical monitoring information. In the second case, the PAC acts as a proxy to retrieve the monitoring data from the Low Level Monitoring System (LLMS) database. This interaction is implemented through the interface:

**IQuery**

- **querySLAViolation**: Queries the SLA Violation registry.
- **queryMonitoring**: Retrieves monitoring information from the LLMS database.

### 9.2.1 External interfaces

Here we present a summary of the components and the interfaces being invoked by the Provisioning and Adjustment Component. Further details can be found in the corresponding deliverables.

The **SLARegistry** needs to be accessed to retrieve the SLAs and their dependencies (**<<query>>** interaction) for adjustment purposes, and also at provisioning time, in order to update the SLA with the access point of the provisioned service (**<<register>>** interaction).

For the SLAM to trigger the provisioning and manipulation of services, the Service Manager must be invoked. **Infrastructure- and Software Service Managers** control the provisioned service instances and encapsulate all service-specific details, acting as mediators between the SLAMs and the service instances. At the software level, the **<<manage software service>>** interaction subsumes all functionality related to the provisioning and management of a software service: provisioning, decommissioning, (re-)configuration of a given service and of the monitoring system. At the infrastructure layer, the **<<manage_infrastructure_service>>** interaction supports all interfaces required to provision requested infrastructure, and to manipulate it once provisioned. Further details can be found in Deliverable D.A3a for the Software Service Manager and in D.A4a for the infrastructure layer.

At runtime, the PAC will receive events through a messaging bus. The **MonitoringEventChannel** serves as a flexible communication infrastructure that allows the collection of information, namely monitoring events (e.g. SLO violations) about the service instance status. The PAC uses the **<<subscribe_to_event>>** interaction stereotype, which allows the connection/disconnection to an event bus, and the subscription/unsubscription to a given channel.

Apart of sending events informing of SLA violations, the **Low Level Monitoring System (LLMS)** provides the **IMonitoringDataAndHistory** interface to retrieve the (last reported) metric values and their history.

Once the provisioning is finished, the PAC should inform the **POC** about the status of the execution of the plan. For this, the **<<plan>>** interaction includes the **IPlanStatus** interface. Furthermore, when a problem appears that cannot be solved locally by the PAC, a re-planning can be triggered using the **IReplan** interface.
Finally, the Business Manager needs to know the exact status and performance of each SLA. This information can flow through the ITrack interface, which provides a callback mechanism by which SLAMs can inform the Business Manager of violations, monitoring details, and any other information required to perform a business-level evaluation of the behaviour of the contracted services.

9.3 Behavioural view

In the previous sections the position of the Provisioning and Adjustment Component inside the SLA Manager has been introduced, together with a description of the main functionalities of the PAC.

At this point, a sequence diagram showing the interactions for the provisioning and adjustment procedures will help to further clarify the mission of the component. In order to keep the description in a domain-independent language, generic names (e.g. <<manage_T_services>>) will be used.

Figure 49 shows a basic and generic scenario of provisioning and adjustment at one of the lower levels (software or infrastructure). The diagram starts when the provisioning request arrives to the POC. First, the POC queries the Service Manager about the possible implementations of a given service, including the capabilities to monitor such a service. Then a request is sent to the Monitoring Manager (MM), which decides the most convenient monitoring configuration according to configurable selecting criteria. The POC then makes a reservation in the Service Manager of one of the selected implementations. Once the negotiation process finishes, the POC makes a plan with the instructions for provisioning, and sends it to the Provision and Adjustment component. The PAC transforms the plan into a set of requests to the Service Manager, taking care of the parallelization and synchronization of tasks. The Service Manager starts the chosen service, and configures the monitoring system, which in turn means the configuration of the Sensors and Effectors. The PAC subscribes to the appropriate channels in the Monitoring Event Bus to receive events from the service layer. The Manageability Agents will publish event as soon as the service is up and running, so the PAC can proceed with the subsequent steps in provisioning. And at runtime, PAC reacts to violations/warnings received by the messaging bus and uses a reasoning engine to carry out operations. Only when a local solution cannot be found, the problem is escalated to the POC.

9.4 Generic PAC Architecture

The Provisioning and Adjustment component can be seen as a two faced module. On one hand, it is the window to command the Service Manager provisioning requests, so it must able to speak and understand the language of the SMs. And it receives events alerting of problems in the deployed services, so only a deep knowledge of the service layer can allow PAC to react upon the violations or warnings. In short, the PAC is by definition domain-specific.
On the other hand, the PAC lives inside the SLA Manager, and one of its main duties is to guarantee the fulfillment of SLAs. Therefore, it must also understand...

Figure 49: Provisioning and Adjustment sequence diagram
the SLA lifecycle and concepts. This part is domain-independent, since it is common to all the layers in the framework and to the different domains.

Given the statements before, it seems infeasible to build a fully generic Provisioning and Adjustment Component. At the same time, when designing the architecture, one of the main objectives was to avoid the obligation for the use cases to write a new instance of the component from scratch.

Thus it became necessary to keep the domain-independent aspects separate to the domain-specific ones, and to make the resulting system easily customizable. Task TA3.5 of WP A3 has designed an autonomic architecture that is appropriate to fulfil the requirements described above. They provide the implementation of a generic agent, in charge of managing a so-called generic Managed Element, which represents a component with direct access to resources. One or more tasks can be assigned to each agent, and each task can rely on a rule engine mechanism to take decisions and trigger actions. Some common tasks are provided: EventBusHandlingTask, receiving events from a messaging bus; TranslationTask, translating or parsing events; AnalysisTask, that relies on a reasoning engine to analyze the incoming events. A persistency mechanism is also available, based on hibernate and SQL.

In the autonomic architecture, configuration of agents and tasks is encapsulated in an XML configuration file. New tasks can be defined just extending the abstract ITask class, and assigned to an agent through the configuration file.

This architecture is very convenient for the implementation of the PAC, since it can be extended and customized without much difficulty. An agent or set of agents can be deployed where needed: in the infrastructure SLAM, in the software SLAM and in the Business Manager, all of which contain a Provisioning and Adjustment Component. In this case, the role of Managed Elements is played by the Service Managers, being then controlled by the agents.

Figure 50 shows a class diagram of the autonomic architecture extended with the classes specifically implemented for the Provisioning and Adjustment Component. Only main classes are shown in order to keep the diagram readable. Three main tasks have been added to cope with the execution of the plans provided by the POC. PlanCoordinationTask keeps track of all the plans currently being executed, adding a control on the number of plans that can be run in parallel. This prevents congestion in the Service Manager as well as keeps the number of running threads under control. PlanExecutionTask coordinates the execution of all the elements that compose a plan. In particular, it ensures that tasks that can run in parallel are triggered at the same time, while those that depend on previous tasks are not launched until the execution of precedent tasks has been completed. Each task of the plan is represented by the ActionExecutionTask, that is domain-dependent and should be extended at software and infrastructure levels to be compatible with the interfaces of the corresponding Service Managers.

The format of messages that need translation (Monitoring and Prediction Messages) and therefore their corresponding translators is also domain-specific, and the exact definition corresponds to the domain where they are used. Nevertheless, the ManageabilityAgentMessage, the one sent when the provision has finished, has been agreed to share a common format.

Apart from the main elements of the architecture, and the classes that extend it, there is a datatype of principal importance: the Plan class. A Plan encapsulates a set of atomic tasks that need to be executed during the provisioning of an SLA. After the negotiation process, a provisioning plan is created by the POC and then sent to the PAC, which triggers the execution. The structure to represent the Plan must allow the expression of parallelization (for those tasks that can be triggered
simultaneously) as well as synchronization semantics (for those tasks with dependencies).

A Directed Acyclic Graph (DAG) [34] fulfils the requirements above. A Directed Acyclic Graph or Acyclic digraph is a directed graph with no cycles in it. Each node of the DAG represents a specific task to be executed, and the hierarchy represents dependencies: a child node can not be executed unless the execution of the all its parent nodes finishes successfully, while children at the same depth can be parallelized.

![Figure 50: PAC Class Diagram](image)

Using the proposed architecture, the SLA provisioning and enforcement approach does not become a monolithic domain-specific platform, but a flexible layer able to manage the SLAs along their lifecycle. This layer could be integrated seamlessly with the pre-existing provider infrastructure when that is based upon service orientation principles.

### 9.5 Deployment architecture

In the previous sections a generic architecture for the Provisioning and Adjustment Component has been described, and a basic implementation has been introduced. The agent-based architecture makes it possible to use the generic
PAC as part of different components, just configuring and deploying the agent properly.

Following the SLA@SOI architecture described in Deliverable D.A1a, a PAC component will be placed in the SLA Managers of the three layers (business, software and infrastructure).

The PACs at software and infrastructure levels are in charge of triggering the provisioning of the service and, at runtime, they receive monitoring information (SLA violations and warnings), and analyze it with the objective of fulfilling the SLAs at that level. If needed, adjustment can trigger some action on the resources. These PACs will start an Agent, implementing the tasks connected to Provision (understood as Plan Execution) and Adjustment.

Conversely, the Business Manager would need to have its own adjustment functionality. To this end, the BM will configure and deploy an Agent, executing the tasks directly connected to Adjustment (not to Provisioning).

The following sections show how the generic PAC can be extended to deal with the specific provisioning and adjustment functionalities at the software and infrastructure layer, for the Open Reference Case (ORC). The corresponding adaptations for the Business Manager are described in Deliverable D.A2a.

### 9.6 Software

This section describes the necessary extensions and adaptations to the generic PAC described above, so to build a reference Provisioning and Adjustment Component for the ORC integration scenarios.

First, in what concerns the provisioning part, a specific `ActionExecutionTask` has been implemented, able to understand the Service Construction Model (SCM) and to invoke the interfaces of the Software Service Manager. The SSM offers the `<<manage_software_services>>` interaction, implemented through the interface `IManageSoftwareService`. The latter subsumes all functionality related to the provisioning and management of a software service:

- Provisioning of an already booked instance, release of a provisioned service instance;
- Configuration and reconfiguration of a service;
- Configuration and reconfiguration of the monitoring system;
- Modification of the implementation or dependencies of an instantiated service; and
- Query possible change actions.

This task needs to be added to an agent, and this is done by including the classname in the agent configuration file, as Deliverable D.A3a explains. When the system is started, the agent configures itself based on this XML file, instantiates, and starts the tasks using Reflection.

Once the system is provisioned, the PAC starts listening to the event bus, in order to receive messages informing about the status of the service. These messages usually are domain-specific, so the format and a parser to translate the XML message to Java classes should be provided and added to the configuration file. Translators can be easily implemented using the Xstream library [36].

At the software level, three types of messages will be received through the event bus. First, Manageability Agents will send an event to inform that the service is effectively running. These messages have been defined in such a way that the
same format can be used in the different layers, encapsulating domain-specific information into a generic hash list.

Second, at runtime the sensors will monitor the performance of the service and will send two types of events: information about the violations of SLA terms, and information about the terms that are close to be violated. These are XML messages that need JAXB to be de-serialized, thus they also need a specific translator to be parsed.

Based on the received events, some adjustment actions can be taken. These actions can be either corrective (to solve a problem, namely a violation in one of the Guarantees of the SLA), either preventive (take into account prediction information, and try to prevent the occurrence of an SLA violation). In both cases, an AnalysisTask (as described in D.A3a) has been used to analyze the incoming events and, using the Drools rule engine to decide the most appropriate action to trigger.

Specific rules have been written for the software services adjustment. For illustration purposes, a very basic rule is shown here:

```java
//Detects a violation indicating that the response time of a given service does not fulfill the SLA, and reconfigures the BPEL it.
rule "Violation on Response time"
when
  $event: Event(type == EventType.MonitoringEventMessage, $sla : event.eventPayload.monitoringResultEvent.monitoredSLA);
  $violation : SLAViolationEvent( type == "response time", $processId : processId, $responsetime : value);
  $ssm : SoftwareServiceManager();
then
  $ssm.executeAction(removePartnerRoleRule($processId));
  $ssm.executeAction(addPartnerRoleRule($processId));
  retract($violation);
end
```

This example shows a rule that acts upon the reception of a violation on the guaranteed response time of an SLA. In that case, the BPEL is reconfigured and the service behaviour is observed to check whether this action has solved the problem.

Rules have been provided that analyze the different received events and trigger different actions: renegotiation, reconfiguration and allocation of extra-resources. These actions have been provided as requirements by the different B-line work packages.

Concerning the use cases, usually they will be sharing with the ORC the interface to the Software Service Manager, as well as the format of the incoming messages. Therefore, typically they should only provide their specific adjustment rules, in case they are different to the ones implemented for the ORC.

### 9.7 Infrastructure

Similar to the previous subsection, here we describe the necessary extensions and adaptations to the generic PAC, with the goal of building a reference Provisioning and Adjustment Component for the ORC infrastructure integration scenarios.
Regarding the provisioning scenario, a specific ActionExecutionTask has been implemented, able to invoke the interfaces of the Infrastructure Service Manager. The ISM offers the \(<\text{manage\_infrastructure\_services}>\) interaction, implemented through the interface IManageInfrastructureService. This interface supports all interactions required to provision requested infrastructure, and to manage it once provisioned. In more detail:

- Given an identifier of an active reservation, or complete details of a new request, commits to this reservation and schedules the infrastructure for provisioning;
- Returns the details of an infrastructure request;
- Given an existing provisioning request, and an updated provisioning request, replaces the former with the later if possible;
- Given a particular resource, allows that resource to be started or stopped; and
- Given a particular provisioning request, terminates the entire provisioning request, tearing down any live VMs and removing any resource commitments previously scheduled.

Similarly to what has been described in the software section, this task needs to be added to an agent using the configurability feature of the system. Then, when the system is started, the agent configures itself, instantiates and starts the specified tasks.

Once the system is provisioned, the PAC starts listening to the event bus, in order to receive messages providing information about the status of the service. These messages are usually domain-specific, so the format and a parser to translate the XML messages to Java classes should be provided and added to the configuration file. Translators are easily implemented using the Xstream library.

At the infrastructure level, two types of messages will be received through the event bus: the ones sent by the Manageability Agents to inform that the service is effectively running, and monitoring events. While the former share the format with the events at the software level, the later may present some differences.

The Infrastructure Service Manager includes a prediction component (as described in Deliverable D.A6a) that provides a framework for predicting the behaviour of basic infrastructural metrics of resource consumption; such as CPU usage, load, memory or disk input/output. This component is offered as a service; that is, events are not automatically sent to the messaging bus, but the prediction service offers its last results through a query interface. To this end, a specific task, PollingTask, has been implemented to allow a periodic query to the infrastructure prediction service.

Both prediction and monitoring events are fed into the drools rule engine managed by an AnalysisTask. For the infrastructure layer, specific rules have been written. For illustration purposes, two basic rules are shown here. Note that for the sake of simplicity, the use of constants has been avoided in these examples, being substituted by numerical examples:
The first example shows a rule that acts upon the reception of a violation on the guaranteed availability of an SLA. In that case, the virtual machine is restarted and its behaviour is observed to see if the reset has improved its performance.

The second example presents a preventive action: after a given set of predictive events (warnings) alerting on high memory consumption for the same infrastructure element arrive in a given time window, then the system triggers a re-provisioning, modifying the amount of memory allocated for the virtual machine.

Rules have been provided that analyze the different received events and trigger different actions: re-provisioning, re-start of VMs or allocation of extra-resources. These actions have been requested by the ORC and the other B-line work packages.

Concerning the use cases, and similarly to the software adaptations, they usually will be sharing with the ORC the interface to the Infrastructure Service Manager, as well as the format of the incoming messages. Therefore, typically they should only extend the ORC PAC by providing their specific adjustment rules, in case they are different to the ones already implemented.
10 SLA Management Console

While the Eclipse platform is designed to serve as an open tools platform, it is architected so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform. By using Eclipse RCP, the developer doesn’t need to take care of the low-level interactions among different graphical components, rather only about the logic design part for each component.

![Figure 51: Structure of Eclipse RCP application]

The SLA management console is an Eclipse RCP-based graphical application. It is used for querying and displaying the SLA information from the SLA registry of the SLAMs within a framework instance.

10.1 Structure and Functions

10.1.1 Structure of the Console

The console includes three main views:

- SLA Query configuration view, on which the user is able to set all the configurations and execute the query operation;
- SLA List view, on which the results of queries (SLA lists) are displayed; and
- SLA information view, on which the detailed information of a specific SLA can be displayed when user selects one item out of SLA list view.

10.1.2 Functions

The SLA management console is a GUI (graphical user interface) that provides an overview of the current state of affairs as regards existing SLAs from the SLA registry.
At beginning, the user has to fill in the configuration information; the very first step is to set the required state(s) of SLA. Currently we support four types of states, namely “Warn”, “Observed”, “Expired” and “Violated”. At least one of them should be selected, thus the query operation will search the SLA list according to the states that user selects (The red area of Figure 52).

Additionally, it can provide information about past SLAs (the yellow area of Figure 52). In this part the user should select “True” if he/she wishes to query only the SLAs that are currently active (i.e. have not expired).

The green area of Figure 52 represents the kind of display that the user would like to see. Either “Full Info” or “Minimum Info” can be chosen. “Full Info” means that all the detailed information about each SLA on the SLA list should be displayed, whereas “Minimum Info” will display only the following information:

- The time that the SLA is agreed between customer and provider;
- The time point, from which the SLA is effective;
- The time point, until which the SLA is effective;
- The template ID of the SLA; and
- The two parties of the SLA.

Furthermore, if a user wishes to query the full information of a SLA list, a dependency of each SLA could be displayed (Blue area of Figure 52). The dependency means that when the UUID of a SLA is provided, the system returns a complete list of dependencies of this SLA onto other SLAs. Apart from that, the user has an extra option “upward”, which means that given the UUID of a SLA, the system returns a complete list of SLAs that depend on the terms of the selected SLA.

The purple area of Figure 52 contains two buttons, “Submit” and “Restore Defaults”. “Submit” initiates the query operation, during which a bar in the
orange area indicates progress. “Restore Defaults” reverts to the default settings of the SLA Query Configuration view.

When the user submits a query, the corresponding result will be shown in the SLA list view, and the general information of the SLA list will be displayed in the SLA information view. Furthermore, when the user selects an item from the SLA list, the detailed information of that SLA will be displayed. In the detailed information window, a user may navigate to previous and next pages.

The application is compatible with several operating systems (Linux, Windows, Mac and so on), all being supported by RCP.

10.2 Implementation

After the user sets the configuration information and presses the submit button, the console will query the SLA registry/-ies and then display the result list in the view "SLA List" and "SLA Information".

In the example Figure 53 there are 4 SLAs retrieved from the SLA registry. After the user selects one item from the SLA list, the detailed information about that SLA will be displayed in the view "SLA Information", as illustrated in Figure 54.

Overall, this work is necessarily considered as a first approach to the problem. It will be refined in the future, when we wish to:

- Refine the architectural structure of SLA management console according to the updates from other components, like SLA model and SLA registry;
- Refine the look and feel of the application;
- Include graphical representations of statistical figures that can provide useful management information to framework administrators.
Figure 54: Detailed SLA information view
11 Conclusions

11.1 Summary
This deliverable presents a summary of the achievements of WP A5 during the 2nd project year, mapping results onto both tasks and the SLA lifecycle. We illustrate developments and novelties; such as the SLA model abstract syntax, interoperable multi-round negotiation, SLA-template-based service discovery, business-driven planning and outsourcing methodologies, autonomic SLA adjustment, etc.

During Y2, there was a lot of progress towards achieving the project objectives. In particular, there is now the SLA Model, and an architectural artefact – the SLA Manager (SLAM) — targeting full SLA lifecycle management. It is important that the SLAM is customizable; it can fit any use case, by appropriate factoring of domain-specific components for which only interfaces have been defined. Reference implementations for infrastructure services, and for the ORC software services, have been implemented and will be made available as implementation examples.

11.2 Outlook on Future Work
Having achieved coherent integration with the other components of the new project architecture (as described in D.A1a), the main goal for WP A5 in the 3rd project year is to focus further on the scientific aspects of SLA management, resulting in further novelty and added project value. We wish to continue with the implementation of template-matching algorithms, used both for service discovery and for filtering of service advertisements; introduce more than 2 parties in multi-round negotiation and implement different complex protocols; extend planning for s/w and infrastructure services with additional novel algorithms and approaches, fitting different scenarios; and enhance the current autonomic SLA adjustment with even more innovative methods (particularly exploring Artificial Intelligence techniques that may be applied to this area).
12 References


**Appendix A: Glossary**

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agreement Initiator</td>
<td>An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.</td>
</tr>
<tr>
<td>Agreement Offer</td>
<td>An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.</td>
</tr>
<tr>
<td>Agreement Responder</td>
<td>The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.</td>
</tr>
<tr>
<td>Agreement Template</td>
<td>An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.</td>
</tr>
<tr>
<td>Agreement Term</td>
<td>Agreement terms define the content of a service level agreement.</td>
</tr>
<tr>
<td>Business Service</td>
<td>A business service is exposed/invoked via at least some non IT elements.</td>
</tr>
<tr>
<td>Business Manager</td>
<td>A specialization of service provider: person that defines the SLATs of products and joins available services in a product.</td>
</tr>
<tr>
<td>External Service</td>
<td>External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.</td>
</tr>
<tr>
<td>Framework Administrator</td>
<td>A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.</td>
</tr>
<tr>
<td>Guarantee Term</td>
<td>Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.</td>
</tr>
<tr>
<td>Hybrid Service</td>
<td>A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).</td>
</tr>
<tr>
<td>Infrastructure Manager</td>
<td>A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.</td>
</tr>
<tr>
<td>Infrastructure Provider</td>
<td>A specific kind of service provider that focuses on the provisioning of infrastructure services.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Infrastructure Service</td>
<td>An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.</td>
</tr>
<tr>
<td>Internal Service</td>
<td>Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.</td>
</tr>
<tr>
<td>IT Service</td>
<td>An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.</td>
</tr>
<tr>
<td>Offered Service</td>
<td>An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.</td>
</tr>
<tr>
<td>Operation Level Agreements</td>
<td>A specification of the conditions under which an internal service or a component is to be used by its “customer”.</td>
</tr>
<tr>
<td>Service</td>
<td>A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. See also service interface type, service concreteness, service exposure</td>
</tr>
<tr>
<td>Service Concreteness</td>
<td>The stage a service reaches over time from a fully abstract type to actually instantiated.</td>
</tr>
<tr>
<td>Service Consumer</td>
<td>Person(s) who actually consume/use the provided services. Typically they belong to the service customer.</td>
</tr>
<tr>
<td>Service Customer</td>
<td>Someone (person or group) who orders/buys services and defines and agrees the service level targets.</td>
</tr>
<tr>
<td>Service Description Term</td>
<td>Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.</td>
</tr>
<tr>
<td>Service Exposure</td>
<td>Services can be exposed either internally (within the same administrative domain) or externally.</td>
</tr>
<tr>
<td>Service Implementation</td>
<td>A service implementation is a possible concrete realization of a given service type.</td>
</tr>
<tr>
<td>Service Instance</td>
<td>A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.</td>
</tr>
<tr>
<td>Service Interface Type</td>
<td>Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface.</td>
</tr>
<tr>
<td>Service Level Consequence</td>
<td>An action that takes place in the event that a service level objective is not met.</td>
</tr>
<tr>
<td>Service Level Agreement</td>
<td>An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or</td>
</tr>
</tbody>
</table>
availability. Entities can dynamically establish and manage agreements via Web service interfaces.

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

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

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

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

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

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

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

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

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

---

**Appendix B: Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOM</td>
<td>Aspect Oriented Modeling</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>AsML</td>
<td>Aspect Markup Language</td>
</tr>
<tr>
<td>BM</td>
<td>Business Manager</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>BSLAM</td>
<td>Business SLA Manager (also: B-SLAM)</td>
</tr>
<tr>
<td>DoW</td>
<td>Description of Work</td>
</tr>
<tr>
<td>EC</td>
<td>Event Calculus</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>GSLAM</td>
<td>Generic SLA Manager</td>
</tr>
<tr>
<td>IE</td>
<td>Interaction Event</td>
</tr>
<tr>
<td>FCR</td>
<td>Finite capacity regions</td>
</tr>
<tr>
<td>IPOC</td>
<td>Infrastructure Planning and Optimization Component</td>
</tr>
<tr>
<td>ISLAM</td>
<td>Infrastructure SLA Manager (also: Infr-SLAM)</td>
</tr>
<tr>
<td>ISM</td>
<td>Infrastructure Service Manager (also: Infr-SM)</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>LHS</td>
<td>Left Hand Side</td>
</tr>
<tr>
<td>LLMS</td>
<td>Low Level Monitoring System</td>
</tr>
</tbody>
</table>
Appendix C: Standard Terms

This appendix presents the SLA@SOI vocabulary of standard terms. For terms which denote function/event operators, we also indicate the required function/event signature\(^{15}\) (i.e. the expected parameters & return-types). Please note that the terms presented here do not include business-level specific terms (such as; pricing, penalties).

\(^{15}\) For present purposes we use only an informal typing scheme. For more formal details, see: http://wiki.sla-at-soi.eu/index.php?title=Workspace/Workpackages/Action_Line_A/A5_SLA_Management_&%26_Foundations/TA5.1_SLA_Model/Standard_Terms
Namespace Conventions

Standard term identifiers are defined as URIs with following namespaces:

- sla = http://www.slaatsoi.org/slamodel#
- qos = http://www.slaatsoi.org/commonTerms#
- core = http://www.slaatsoi.org/coremodel#
- units = http://www.slaatsoi.org/coremodel/units#
- infra = http://www.slaatsoi.org/resources#

SLA Roles, Protocols & Policies

Agreement Roles

\texttt{sla:provider} indicating the primary provider of, or broker/aggregator for, the services governed by a SLA. There can only be one provider per SLA.

\texttt{sla:customer} indicating the customer of the services governed by a SLA. There can only be one customer per SLA.

\texttt{sla:third\_party\_provider} in the case that the primary provider is a broker/aggregator, indicates a 3rd party provider of the "brokered" services governed by the SLA.

\texttt{sla:auxiliary\_party} indicating an external party.

Endpoint Protocols

\texttt{sla:telephone} communication by voice telephony.


\texttt{sla:email} communication by e-mail.

\texttt{sla:SMS} Short Message Service.

\texttt{sla:REST} a ‘REST’ style communication interface (implementation details to be specified using annotations).

\texttt{sla:XMPP} Extensible Messaging and Presence Protocol

\texttt{sla:HTTP} Hypertext Transfer Protocol

Action Policies

\texttt{sla:mandatory} the action \textbf{must} be performed

\texttt{sla:optional} the action \textbf{may} be performed (the action is optional when specified preconditions hold, but is otherwise forbidden)

\texttt{sla:forbidden} the action \textbf{must not} be performed

Default Interface Operation “Related” Properties

All interface operations are associated with a default set of "related" properties (i.e. properties whose values it is assumed can be readily accessed for each invocation of the operation). The standard \textbf{identifiers} for the default interface operation "related" properties are listed below:
endpoint denotes the endpoint (type ENDPOINT) at which the operation was invoked

consumer denotes the consumer (type AGENT) who invoked the operation (i.e. the invocation message sender).

request_time denotes the point in time (type TIME_STAMP) at which the operation was invoked (i.e. invocation message received)

reply_time denotes the point in time (type TIME_STAMP) at which the operation was completed (i.e. posted a reply)

* note that these terms are defined as simple identifiers, not URIs.

**Built-In Operators & Functions**

**Logical Operators**
The legal operators for compound constraint & domain expressions ...

- core:and (arity 2+)
- core:or (arity 2+)
- core:not (arity 1)

**Comparison Operators**
The legal comparison operators for domain expressions ...

- core:identical_to
  - core:equals
  - core:not_equals
  - core:less_than
  - core:less_than_or_equals
  - core:greater_than
  - core:greater_than_or_equals
  - core:member_of
  - core:subset_of
  - core:proper_subset_of
  - core:superset_of
  - core:proper_subset_of
  - core:matches
    - core:isa tests the "type" of the LHS, e.g. "X isa xsd:integer"

**Arithmetic Functions**

- core:add expression:
  - add( a : <number>, b : <number> ) : <number>
  - definition: addition of a and b.
core:subtract  expression:
subtract( a : <number>, b : <number> ) : <number>
definition: subtraction of b from a.

core:multiply  expression:
multiply( a : <number>, b : <number> ) : <number>
definition: multiplication of a by b.

core:divide  expression:
divide( a : <number>, b : <number> ) : <number>
definition: division of a by b.

core:modulo  expression:
modulo( a : <number>, b : <number> ) : <number>
definition: modulo division of a by b.

core:round  expression:
round( a : <number>, b : <count> ) : <number>
definition: rounds a floating point value a to the degrees of precision specified by integer b

Aggregate (Set) Functions

core:sum  expression:
sum( x : <array-of-number>) : <number>
definition: summation over the values in x.

core:std  expression:
std( x : <array-of-number>) : <number>
definition: standard-deviation of the values in x.

core:mean  expression:
mean( x : <array-of-number>) : <number>
definition: mean of the values in x.

core:median  expression:
median( x : <array-of-number>) : <number>
definition: median of the values in x.

core:mode  expression:
mode( x : <array-of-number>) : <number>
definition: mode of the values in x.

core:max  expression:
max( x : <array-of-number>) : <number>
definition: maximum value in x.

core:min  expression:
min( x : <array-of-number>) : <number>
definition: minimum value in x.
**core:count** expression:

count( x : <array-of-any> ) : <quantity>

\[ \text{count( x : <array-of-any>, d : <domain> ) : <quantity>} \]

*definition*: number of values in \( x \) (which lie in the domain \( d \)).

**core:percent** expression:

percent( x : <array-of-any>, d : <domain> ) : <ratio>

\[ \text{percent( x : <array-of-any>, d : <domain> ) : <ratio>} \]

*definition*: percentage of values in \( x \) which lie in the domain \( d \).

**Time-Series**

**core:series** expression:

series( f : <function>, e : <event> ) : <array-of-number>

\[ \text{series( f : <function>, e : <event> , c : <count> ) : <array-of-number>} \]

*definition*: a time-series collated by successive measurements of function \( f \) triggered by the class of events \( e \) with a maximum capacity \( c \) (or unlimited capacity if not specified).

**core:value** expression:

value( f : <function>, e : <event> ) : <number>

\[ \text{value( f : <function>, e : <event> ) : <number>} \]

*definition*: the latest measurement of metric \( f \) triggered by the class of events \( e \).

**Access to Context**

**core:time_is** expression:

time_is( ) : <time-stamp>

*definition*: denotes the time of day

**core:day_is** expression:

day_is( ) : <day>

*definition*: denotes the day of the week. Default values are; MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

**core:month_is** expression:

month_is( ) : <month>

*definition*: denotes the month of the year. Default values are; JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER

**core:year_is** expression:

year_is( ) : <year>

*definition*: denotes the year (as a four digit integer; YYYY)

**Miscellaneous**

**core:interval** expression:

interval( s : <time-stamp>, e : <time-stamp>)
definition: denotes a specific interval of time from start 
s to end e.

**sla:custom_location**  standard term for unknown endpoint locations in SLA Templates (i.e. denoting "SLA specific locations").

core:param denotes an "unspecified parameter", used only in 'iterators' (see below).

core:iterator expression:

```
iterator( f : <metric> , s : <array-of-any-type> ) : <array-of-any-type>
```

**definition:** similar to a time-series, but providing a sequential evaluation of the METRIC 'f' by iterating over the parameters supplied in 's'. e.g. iterator( add(?,?), [[1,2],[3,4]]) , where '?' are "unspecified parameters" (see above), evaluates to the sequence [3,7].

core:transpose expression:

```
transpose( mtx : <array-of-any-type> ) : <array-of-any-type>
```

**definition:** denotes a matrix-transpose operation, e.g.
```
transpose( [[a,b,c],[1,2,3]] ) evaluates to [[a,1],[b,2],[c,3]].
```
The result is undefined for inputs which are not 2-dimensional.

**Events**

Event expressions specify "classes of events", which are used in SLA(T)s to denote the 'trigger' conditions in time-series functions and guaranteed actions. Standard event terms are listed below:

core:specialisation expression:

```
specialisation[ e : <event>, c : <constraint> ] : <event>
```

**definition:** the specialisation (sub-class) of event e by the membership constraint c.

core:intersection expression:

```
intersection[ e1 : <event>, e2 : <event> ] : <event>
```

**definition:** the intersection of event expressions e1 & e2.

core:union expression:

```
union[ e1 : <event>, e2 : <event> ] : <event>
```

**definition:** the union of event expressions e1 & e2.

core:periodic expression:

```
periodic[ f : <duration> ] : <time-signal>
```

**definition:** periodic time signals at regular intervals
with frequency $f$ (measured in time units, e.g. every 1hr, 5s, 3 days, etc.), recurring indefinitely.

**core:** bounded_periodic expression:

bounded_periodic[ $f : <duration>$, $i : <interval>$ ] : <time-signal>

**definition:** as for periodic[ $f$ ], but bounded by the interval $i$.

**core:** schedule expression:


**definition:** a sequence $ts$ of specific, successive fixed points in time.

**core:** time expression:

time[ $t : <time-stamp>$ ] : <time-signal>

**definition:** a specific point $t$ in time.

**core:** fault expression:


**definition:** the throwing of an exception (fault) during the invocation of an interface operation.

**core:** violated expression:

violated[ $c : <constraint>$ ] : <sla-state-change>

**definition:** the detection of a violation of a constraint $c$.

**core:** warned expression:


**definition:** the generation of a warning that a constraint $c$ is likely to be violated with probability (or reliability) $p$.

**core:** recovered expression:

recovered[ $c : <constraint>$ ] : <sla-state-change>

**definition:** the detection of a recovery from a previous violation of constraint $c$.

**Quality of Service (QoS) Terms**

The standard Quality of Service (QoS) terms defined in SLA@SOI are listed below:

**Numeric Measures (Metrics)**

The following metrics all take a parameter, $S$ - denoting a 'service', of which there are 2 basic types:

- `<message-type-service>` : defined as an arbitrary collection of *interface operations*. In particular, *wrt* the SLA(T) Model, $S$ may be the ID of a
ServiceRef, an InterfaceDeclr or an interface Operation.

- `<resource-type-service>` : denotes an infrastructure ‘resource’ made available as a ‘service’. In particular, wrt the SLA(T) Model, S must be the ID of a ResourceType entity.

Different QoS definitions may apply according to service type. In addition, for `<message-type-services>`, QoS definitions also need to cover the case that functional interfaces do not explicitly specify any operations. In such cases, we assume the following:

  - that a communications protocol is explicitly specified (e.g. HTTP, SOAP, etc),
  - that each invocation via the protocol is interpreted as a distinct operation call.

Many of the definitions assume some ‘monitoring schedule’ operating over a temporal interval $T = T_2 - T_1$, where:

- $T_1$ : is the point in time at which monitoring of the metric begins
- $T_2$ : is the point in time at which the metric is evaluated
- $T_2 > T_1$

The values $T_1$ and $T_2$ are specified by enclosing the metric ‘function’ in a Time-Series.

All numerical functions are classes as type `<function>`, i.e. they can be employed in any expression where the type `<function>` is allowed.

The definitions given below apply only to services of type `<message-type-service>` for which explicit operations have been defined. Definitions for other cases (e.g. `<resource-type-service>`) have not yet been finalised.

$qos:availability$ expression:

availability( s : <service> ) : <ratio>

the probability that a service, $S$, is up and running.

definition: the probability that a service, $S$, is up and running.

formal definition:

- given $b_t$ as a time at which none of the operations in $S$ could be invoked (by consumers of $S$), due to reasons other than network connectivity, where: $T_1 \leq b_t \leq T_2$
- given $e_t$ as the moment when one or more operations in $S$ became invocable again, where: $b_t < e_t$ and $T_1 \leq e_t \leq T_2$
- such that, for the monitoring period $T$, the total duration, $d$, of 'downtime' is given by: $d = \Sigma( e_t - b_t )$
- the availability, $A$, for $S$ is defined as: $A = (T - d) / T$

$qos:accessibility$ expression:

accessibility( s : <service> ) : <ratio>

definition: the probability that a service, $S$, is
accessible. 'Accessibility' refers to the ability to complete an invocation request, leading either to a state change or a return message (independent of its kind, e.g. success, failure/error, information, warning, etc.).

**Formal Definition:**

- Given $I_a$ as the set of all invocations of operations in $S$ during the monitoring period $T$,
- Given $I_d \subseteq I_a$ as the set comprising only those invocations which were not served (i.e. which were dropped),
- $N_a$ and $N_d$ are the number of elements in $I_a$ and $I_d$ respectively,
- The accessibility, $A$, for $S$ is defined as: $A = (N_a - N_d) / N_a$

**QoS: Arrival Rate Expression**

\[
\text{arrival\_rate}(s : \text{<service>}) : \text{<transaction\_rate>}
\]

**Definition:** denotes the invocation rate for operations in service $S$.

**Formal Definition:**

- Given $t$ as some standard time unit (i.e. a STND term denoting 'second', 'minute', 'hour' etc.),
- $R = n/t$ is the request arrival rate, where $n$ is the number of invocation requests received for operations in $S$ per time unit $t$.

**QoS: Data Volume Expression**

\[
\text{data\_volume}(s : \text{<service>}) : \text{<data\_size>}
\]

**Definition:** denotes the data volume that is processed per service request.

**Formal Definition:**

- Given a request $r_0$ for with parameters $P_1, ..., P_n$ to some operation $O$ of $S$,
- The data volume of $O$ is defined as the sum of the sizes of all its parameters.

**QoS: Throughput Expression**

\[
\text{throughput}(s : \text{<service>}) : \text{<transaction\_rate>}
\]

**Definition:** denotes the maximum arrival rate for operations in service $S$, before invocations start being dropped.

**Formal Definition:**

- Given arrival rate $R$
- $R_1$ is that value of $R$ where requests are
starting to be dropped (i.e., the point where $A$ becomes marginally smaller than 1)

- dependent on the accessibility, $A$, for $S$:
  - if $A = 1$ then $R = R_1$,
  - if $A < 1$ then $R > R_1$,
- the throughput, $Q$, for operations in $S$ is defined as: $Q = R_1$

$qos$: completion_time expression:

$\text{completion\_time}( s : <\text{message\_type\_service}> ) : <\text{duration}>$

definition: the completion time for operations in service $S$; i.e. the time between receiving an invocation message (on the service end) for some operation, $O$, in $S$, and completing the construction of a response (as measured in some domain/application specific time unit).

formal definition:

- given a request (invocation) message $M_Q$ for $O$ received in full at time $t_Q$,
- given a response message $M_R$ for $O$ that was put on the wire in full at time $t_R$,
- the completion time, $t_{CT}$, of $O$ is defined as: $t_{CT} = t_R - t_Q$.

$qos$: mttr expression:

$\text{mttr}( s : <\text{service}> ) : <\text{duration}>$

definition: "mean time to repair", a time value indicating how much time it takes, on average, from the moment a service, $S$, becomes unavailable to the moment it is available again [see definition of 'availability' above].

formal definition:

- given a moment in time, $t_b$, at which $S$ becomes unavailable,
- given the respective moment in time, $t_e$, that $S$ becomes available again; $t_b < t_e$,
- $t = t_e - t_b$ is the respective period (duration) of unavailability,
- $T = \{ t_1, t_2, ..., t_n \}$ is a series of $n$ periods of unavailability,
- the mean-time-to-repair, $T_{mttr}$, of $S$ is defined as: $T_{mttr} = \Sigma( T ) / n$.

$qos$: mttf expression:

$\text{mttf}( s : <\text{service}> ) : <\text{duration}>$

definition: "mean time to failure", a time value
indicating how much time it takes, on average, from one failure of service $S$ to the next.

**formal definition:**

- given a moment in time, $t_{b}$, at which a restoration following a failure of $S$ occurs,
- given a consecutive failure of $S$ starting at time $t_{a}$, $t_{b} < t_{a}$,
- $t = t_{a} - t_{b}$ is the respective period (duration) of availability,
- $T = \{ t_{1}, t_{2}, ..., t_{n} \}$ is a series of $n$ periods of availability,
- the mean-time-to-failure, $T_{mtt}$, of $S$ is defined as: $T_{mtt} = \sum(T) / n$.

**qos:reliability expression:**

reliability( $s : <service>$ ) : $<$ratio$>

**definition:** the probability that a service, $S$, is accessible and completes without failure, i.e., the result of service execution is not a message / return value / signal indicating that a failure has occurred. We assume that failures may arise due to software bugs in service components, as well as failures of the execution environment of a service (e.g., application servers, operating systems, underlying hardware resources).

**formal definition:**

- given $I_{a}$ as the set of all invocations of operations in $S$ during the monitoring period $T$,
- given $I_{f} \subseteq I_{a}$ as the set comprising only those invocations which were not served (due to inaccessibility) or completed with failure,
- $N_{a}$ and $N_{f}$ are the number of elements in $I_{a}$ and $I_{f}$ respectively,
- the reliability, $R$, for $S$ is defined as: $R = (N_{a} - N_{f}) / N_{a}$

**qos:isolation expression:**

isolation( $s : <service>$ ) : $<$boolean$>

**definition:** indication whether it is possible to have "provider agents" for service $S$ executed on different physical (or virtual) machines for different consumers of $S$.

**formal definition:**

- given a set, $C = \{ C_{1}, C_{2}, ..., C_{n} \}$, of consumers of $S$,
- a physical/virtual machine, $M$,
• the predicate \( \text{executes}(C, M) \) representing the fact that "provider agents" (processes) for \( S \), for one or more consumers in set \( C \), execute on machine \( M \),

• then, service \( S \) is "isolated" for some consumer, \( C_i \in C \), iff:

• \( \text{executes}( \{ C_i \}, M ) \land \neg \text{executes}( C - \{ C_i \}, M ) \)

**qos: accuracy** expression:
\[
\text{accuracy}(s : \text{<service>}) : \text{<count>}
\]
definition: the number of significant digits to the right of the decimal point.

formal definition:
• for each operation, \( O \), in \( S \),
• \( O \) returns a result \( V \), where \( V \in C, V = a + bi \),
• for \( a = a - \lfloor a \rfloor \) and \( \beta = b - \lfloor b \rfloor \),

\[
( a \in \mathbb{R} ) \land ( a \cdot 10^n \in \mathbb{Z}), n \in \mathbb{N},
\]
\[
( \beta \in \mathbb{R} ) \land ( \beta \cdot 10^m \in \mathbb{Z}), m \in \mathbb{N},
\]
• the accuracy, \( U \), of operation \( O \) is defined as: \( U = \max(n, m) \)

**Conformance to Standards**

**qos: non_repudiation** expression:
\[
\text{non_repudiation}(s : \text{<service>}) : \text{<array-of-standards>}
\]
definition: indication whether it is possible for a service provider to deny use of the service \( S \) by a customer (and any consumers on whose behalf the customer acts).

formal definition:
• the "non-repudiation indicator" of service \( S \) is defined as a set \( M = \{ M_1, M_2, .., M_n \} \) of well-defined, commonly understood (i.e. by both customers & providers) methodologies for certifying the authenticity of signatures and/or the integrity of signed data.

**qos: supported_standards** expression:
\[
\text{supported_standards}(s : \text{<service>}) : \text{<array-of-standards>}
\]
definition: denotes the set of standard specifications, \( Q \), supported by service \( S \).

formal definition:
• assuming a commonly understood set, \( L \), of standard specifications relevant to the domain at hand: \( L = \{ L_1, L_2, \ldots, L_n \} \), where; \( L_{1 sisn} \) is a fully qualifying name, e.g. "ISO8859-7",

• then, \( Q \subseteq L = \{ Q_1, Q_2, \ldots, Q_m \} \), \( m \leq n \), is the subset of \( L \) for which \( Q_i \) is fully implemented by \( S \forall j \in \{1, \ldots, m\} \).

**qos:** regulatory expression:

\[ \text{regulatory}( s : <service> ) : <array-of-standards> \]

**definition:** denotes the set of regulatory specifications, \( V \), supported by service \( S \).

**formal definition:**

• assuming a commonly understood set, \( L \), of regulatory specifications that apply in the domain at hand:

\[ L = \{ L_1, L_2, \ldots, L_n \} \], where; \( L_{1 sisn} \) is a fully qualifying name, e.g. "31994Y0702" or "OJ C 181, 2.7.1994, p. 1-2",

• then, \( V \subseteq L = \{ V_1, V_2, \ldots, V_m \} \), \( m \leq n \), is the subset of \( L \) for which \( V_j \) is fully respected by \( S \forall j \in \{1, \ldots, m\} \).

**qos:** integrity expression:

\[ \text{integrity}( s : <service> ) : <array-of-standards> \]

**definition:** represents the ability of a service, \( S \), to preserve data integrity.

**formal definition:**

• integrity is synonymous with "supported_standards" (above), where the relevant standards are service transaction protocols.

**Security**

**qos:** authentication expression:

\[ \text{authentication}( s : <service> ) : <untyped> \]

**definition:** indication of proper authentication method(s) being used by service \( S \).

**formal definition:**

• given uniquely identifying information \( ID \) of type \( T_i \) for a service consumer, \( C_i \), either of the specific service \( S_j \) or some broader domain of services (possibly all available services through unique world-wide identification),

• a well-defined methodology, \( M_i \), which accepts input of type \( T_i \) and certifies it as correct (acceptable/true credentials) or
incorrect (unacceptable/false credentials),

- a set, \( P = \{ <M_1, T_1>, <M_2, T_2>, \ldots, <M_n, T_n> \} \) of all possible methodology/input-type pairs,

- then the subset \( N \subseteq P \) with \( k \leq n \) elements, is defined as the "authentication indicator" of \( S \), if \( S \) offers all methods \( M_i \) of \( N \) for the identification of consumers.

**qos: auditability**

**expression:**

\[
\text{auditability( s : <service> ) : <untyped>}
\]

**definition:** indication of auditable logs being maintained for service \( S \).

**formal definition:**

- assuming a set \( I = \{ I_1, I_2, \ldots, I_n \} \) of \( n \) auditable pieces of information (e.g. originating IP address, time of request, etc.),

- a set, \( K_i \) of preconditions which confirm the validity of auditable information (e.g. NTP synchronisation, read-only user access to auditable data, etc.),

- \( D \subseteq I = \{ D_1, D_2, \ldots, D_m \}, m \leq n \), and \( D_i \) logged by \( S \) \( \forall i \in \{1, \ldots, m\} \),

- then the auditability of \( S \) is defined as the pair: \( <D, K> \)

**qos: authorisation**

**expression:**

\[
\text{authorisation( s : <service> ) : <untyped>}
\]

**definition:** indication of proper authorisation framework being used by service \( S \).

**formal definition:**

- for a set, \( C = \{ C_1, C_2, \ldots, C_n \} \), of consumers of \( S \),

- and a well-defined set, \( R = \{ R_1, R_2, \ldots, R_m \} \), of resources exposed by \( S \),

- the "authorisation indicator" of \( S \) is defined as a mapping function: \( Z : C \times R \rightarrow \{ \text{true, false} \} \) which is specified according to a well-defined set of rules (possibly undisclosed to consumers) and which evaluates to \( \text{true} \) iff consumer \( C_{1\leq i\leq n} \) is authorised to access resource \( R_{1\leq j\leq n} \).

**qos: data_encryption**

**expression:**

\[
\text{data_encryption( s : <service> ) : <untyped>}
\]

**definition:** indicates encryption capabilities for data
management (transfer, storage or other).

formal definition:

- given a well-defined methodology, $M_i$, which accepts input of type $T_i$ (and produces encrypted output of some type $T_j$ that can then be decrypted using a well-defined methodology, $M_j$),
- a set, $P = \{ <M_1,T_1>, <M_2,T_2>, \ldots, <M_n,T_n> \}$ of all possible methodology/input-type pairs,
- a subset $N \subseteq P$ with $k \leq n$ elements,
- a set $R = \{ R_1, R_2, \ldots, R_m \}$, of resources (e.g. network stack, storage) exposed/consumed by $S$
- the "data-encryption capabilities", $E$, of $S$ are then defined as: $E = N \times R$

Infrastructure

The following expressions are specific to infrastructure 'resources' (i.e. services of <resource-type-service>):

\[ \text{infra:CPU\_Cores} \]
expression:

\[ \text{CPU\_Cores}( r : \text{<resource-type-service>}) : \text{<count}> \]

\[ \text{definition: denotes the Number of Cores of } r \]

\[ \text{infra:CPU\_Speed} \]
expression:

\[ \text{CPU\_Speed}( r : \text{<resource-type-service>}) : \text{<frequency}> \]

\[ \text{definition: denotes the processing speed of } r \]

\[ \text{infra:Memory} \]
expression:

\[ \text{Memory}( r : \text{<resource-type-service>}) : \text{<data-size}> \]

\[ \text{definition: denotes the amount of available memory of } r \]

Metric Units

The standard terms for metric units defined in SLA@SOI are listed below:

Area Units   \[ \text{units:mm}^2 \]
\[ \text{units:um}^2 \]

Data Rate Units   \[ \text{units:b\_per\_s} \]
\[ \text{units:Kb\_per\_s} \]
\[ \text{units:Mb\_per\_s} \]

Data Size Units   \[ \text{units:bit} \]
\[ \text{units:Byte} \]
\[ \text{units:KB} \]
units: MB
units: GB

Energy Units
units: J
units: KJ
units: Wh
units: KWh
units: mWh

Length Units
units: m
units: cm
units: mm

Units of Duration
units: s
units: tick
units: ms
units: us
units: min
units: hrs
units: day
units: week
units: month
units: year

Frequency Units
units: Hz
units: KHz
units: MHz
units: GHz
units: rpm

Ratio Units
units: percentage

Power Units
units: W
units: mW
units: kW

Transaction-Rate Units
units: tx_per_s
units: tx_per_m
units: tx_per_h

Weight Units
units: tx_per_s
units: tx_per_m
units: tx_per_h

Currency Units
units: EUR
units: USD
**Appendix D: EC-Assertion formulas for Throughput**

**Set of EC-Assertion formulas for Throughput**

A0.Throughput.<CaseId>:
Initially(Served(0,<_SrvId>,0,0))

A1.Throughput.<CaseId>:
Initially(<CaseId>.Throughput(0))

A2.Throughput.<CaseId>:
Happens(e(_id1, _Snd, <_SrvId>, Call(_O), <_SensorId>), t1, [t1,t1])∧
Happens(e(_id2,<_SrvId>, _Snd,Response(_O), <_SensorId>), t2, [t1,t1+<CaseId.d>])∧
(∃ t3: HoldsAt(Served(_P, <_SrvId>, _N1, t3), t1)∧(t3 ≤ t1))∧
(¬∃ t4, _P1, _N2: (_P1≠_P2)∧(t4 ≥ t3)∧(t4 < t1+<CaseId.d>)∧HoldsAt(Served(_P2, <_SrvId>, _N2, t4), t1+<CaseId.d>))⇒
Terminates(e(_id1, <_SrvId>, _Snd, Response(_O), <_SensorId>), Served(_P1, <_SrvId>, _N1, t3), t1)∧

A3.Throughput.<CaseId>:
Happens(e(_id1, _Snd, <_SrvId>, Call(_O), <_SensorId>), t1, [t1,t1])∧
¬Happens(e(_id2,<_SrvId>, _Snd,Response(_O), <_SensorId>), t2, [t1,t1+<CaseId.d>])∧
(∃ t3: HoldsAt(Served(_P, <_SrvId>, _N, t3), t1)∧(t3 ≤ t1))∧
(¬∃ t4, _P1, _N1: (_P1!=_P)∧(t4 ≥ t3)∧(t4 < t1+<CaseId.d>)∧HoldsAt(Served(_P1, <_SrvId>, _N1, t4), t1+<CaseId.d>))⇒
Initiates(e(_id1, <_SrvId>, _Snd, Response(_O), <_SensorId>), Served( _P1+1, <_SrvId>, _N1+1, 0, t1), t1)

A4.Throughput.<CaseId>:
Happens(e(_id1, _Snd, <_SrvId>, Call(_O), <_SensorId>), t1, [t1,t1])∧
¬Happens(e(_id2,<_SrvId>, _Snd,Response(_O), <_SensorId>), t2, [t1,t1+<CaseId.d>])∧
(∃ t3: HoldsAt(Served(_P, <_SrvId>, _N, t3), t1)∧(t3 ≤ t1))∧
(¬∃ t4, _P1, _N1: (_P1!=_P)∧(t4 ≥ t3)∧(t4 < t1+<CaseId.d>)∧HoldsAt(Served(_P1, <_SrvId>, _N1, t4), t1+<CaseId.d>))⇒
Initiates(e(_id1, <_SrvId>, _Snd, Response(_O), <_SensorId>), <CaseId>.Throughput(
The assumption formulas A0 and A1 initiate the Served and \(<\text{CaseId}.\text{Throughput}\) fluents for the first time. The assumption A2 increases the number of served calls for a given throughput measurement period, after an operation call has been served. The assumption formula A3 starts a new throughput period, after an operation call has been dropped. Finally, A5 initiates the Throughput fluent, after an operation call of the monitored service has been dropped.

**Appendix E:** EC-aware FTL template for Throughput

**EC aware FTL template for Throughput**

\[
(\forall \text{case} : (\text{CaseId}); \text{srvId} : (\_\text{SrvId}); \ t : \text{Time})
\]

\(A0.\text{Throughput}.\text{case}:
\]

Initially(Served(0,srvId,0,0))

(\(\forall \text{case} : (\text{CaseId}); \text{srvId} : (\_\text{SrvId})\))

\(A1.\text{Throughput}.\text{case}:
\]

Initially(case.\text{Throughput}(0))

(\(\forall \text{case} : (\text{CaseId}); \text{srvId} : (\_\text{SrvId}); \text{sensorId} : (\_\text{SensorId}); \ d : (\text{CaseId}.d); \ t1, t2, t3, t4 : \text{Time}\))

\(A2.\text{Throughput}.\text{case}:
\]

Happens(e(\_\text{id1}, \_\text{Snd}, srvId, Call(\_O), sensorId), t1, [t1,t1]) \land

Happens(e(\_\text{id2}, srvId, \_\text{Snd}, Response(\_O), sensorId), t2, [t1, t1+t4]) \land

(\exists t3: HoldsAt(Served(_P, srvId, _N1, t3), t1) \land (t3 \leq t1)) \land

(\neg \exists t4, _P2, _N2: (_P1 \neq _P2) \land (t4 < t1+d) \land HoldsAt(Served(_P2, srvId, _N2, t4), t1+t4)) \Rightarrow

Terminates(e(\_\text{id1}, srvId, \_\text{Snd}, Response(\_O), sensorId), Served(_P1, srvId, _N1+1, _t3), t1) \land

Initiates(e(\_\text{id1}, srvId, _Snd, Response(\_O), sensorId), Served(_P1, srvId, _N1+1, _t3), t1)

(\forall \text{case} : (\text{CaseId}); \text{srvId} : (\_\text{SrvId}); \text{sensorId} : (\_\text{SensorId}); \ d : (\text{CaseId}.d); \ t1, t2, t3, t4 : \text{Time})

\(A3.\text{Throughput}.\text{case}:
\]

Happens(e(\_\text{id1}, \_\text{Snd}, srvId, Call(\_O), sensorId), t1, [t1,t1]) \land

\neg Happens(e(\_\text{id2}, srvId, \_\text{Snd}, Response(\_O), sensorId), t2, [t1, t1+t4]) \land
(\exists t_3: \text{HoldsAt}(\text{Served}(_P, \text{srvId}, _N1, t_3), t_1) \land (t_3 \leq t_1)) \land
(-\exists t_4, _P2, _N2: (\text{P1} \neq _P2) \land (t_4 \geq t_3) \land (t_4 < t_1+d) \land \text{HoldsAt}(\text{Served}(_P2, \text{srvId}, _N2, t_4), t_1+d)) \Rightarrow
\text{Initiates}(e(_id1, \text{srvId}, _\text{Snd}, \text{Response}(_O), \text{sensorId}), \text{Served}(_P1+1, \text{srvId}, 0, _t3), t_1)

(\forall \text{ case} : \langle \text{CaseId} \rangle; \text{srvId:} \langle _\text{SrvId} \rangle; \text{sensorId:} \langle _\text{SensorId} \rangle; d: \langle \text{CaseId.d} \rangle; t_1, t_2, t_3, t_4 : \text{Time}) \quad (4)

A4.Throughput.case:
\text{Happens}(e(_id1, _\text{Snd}, \text{srvId, Call}(_O), \text{sensorId}), t_1, [t_1, t_1]) \land
\neg \text{Happens}(e(_id2, _\text{srvId, Snd, Response}(_O), \text{sensorId}), t_2, [t_1, t_1+d]) \land
(\exists t_3: \text{HoldsAt}(\text{Served}(_P, \text{srvId}, _N1, t_3), t_1) \land (t_3 \leq t_1)) \land
(-\exists t_4, _P2, _N2: (\text{P1} \neq _P2) \land (t_4 \geq t_3) \land (t_4 < t_1+d) \land \text{HoldsAt}(\text{Served}(_P2, \text{srvId}, _N2, t_4), t_1+d)) \Rightarrow
\text{Initiates}(e(_id1, \text{srvId}, _\text{Snd}, \text{Response}(_O), \text{sensorId}), \text{case.Throughput}(_N), t_1)

\textbf{Appendix F : SLAs into Abstract Syntax Trees algorithm}

\textit{Algorithm for parsing a subset of SLA objects into abstract syntax trees}

Parse(inputObject)
1. node : ASTNode
2. node = null
3. node.Object = inputObject
4. IF node.Object is an AgreementTerm THEN
5. node.Label = IMPLIES
6. IF node.Object.Precondition is NOT empty THEN
7. bodyNode = parse(inputObject.Precondition)
8. node.addLeftChild(bodyNode)
9. ELSE
10. node.addLeftChild(null)
11. END IF
12. headNode.Object = LogicalOperator.AND
13. headNode.Label = AND
14. FOR each guaranteedState in input.Object.Guaranteed.States[] DO
15. guaranteeStateNode = parse(guaranteedState)
16. headNode.addChild(guaranteeStateNode)
17. END FOR
18. node.addRightChild(headNode)
19. ELSE IF node.Object is a GuaranteedState THEN
20. node.Label = IMPLIES
21. IF node.Object.Precondition is NOT empty THEN
22. bodyNode = parse(node.Object.Precondition)
23. node.addLeftChild(bodyNode)
24. ELSE
25. node.addLeftChild(null)
26. END IF
27. headNode = parse(node.Object.State)
28. node.addRightChild(headNode)
29. ELSE IF node.Object is a ConstraintExpr THEN
30. IF node.Object is a CompoundConstraintExpr THEN
31. nodeLabel = inputObject.LogicalOperator.STND.Value
32. FOR each subExpression in inputObject.SubExpressions DO
33. subExpressionNode = parse(subExpression)
34. node.addChild(subExpressionNode)
35. END FOR
36. ELSE IF node.Object is a TypeConstraintExpr THEN
37. IF node.Object.DomainExpr is a SimpleDomainExpr THEN
40. node.addLeftChild(LHSNode)
42. node.addRightChild(RHSNode)
43. ELSE IF node.Object.DomainExpr is a CompoundDomainExpr THEN
44. node.Name = inputObject.DomainExpr.LogicalOperator.STND.Value
45. FOR each subExpression in node.Object.DomainExpr.SubExpressions DO
46. subExpressionNode = parse(subExpression)
47. node.addChild(subExpressionNode)
48. END FOR
49. END IF
50. END IF
51. END IF
52. return node
END Parse

Appendix G: Selecting Parametric Templates

Algorithm for selecting parametric templates with respect to given abstract syntax tree

Translate(node, initialNode, ParametricECFormulasDB, ParametricECPredicatesDB, QoSTermECCConditionsDB, ECFormulasList, monitoringRule)
1. IF node.Object is an AgreementTerm THEN
2. body = node.LeftChild
3. IF body is NOT null THEN
4. Translate(body, initialNode, ParametricECFormulasDB, ParametricECPredicatesDB, QoSTermECCConditionsDB, ECFormulasList, monitoringRule)
5. END IF
6. head = node.RightChild
7. Translate(head, initialNode, ParametricECFormulasDB, ParametricECPredicatesDB, QoSTermECCConditionsDB, ECFormulasList, monitoringRule)
8. ELSE IF node is a LogicalOperator THEN
9. FOR each child in node.Children DO
10. Translate(child, initialNode, ParametricECFormulasDB, ParametricECPredicatesDB, QoSTermECCConditionsDB, ECFormulasList, monitoringRule)
11. END FOR
12. ELSE IF node is a ComparisonOperator THEN
13. relationalPredicate = ParametricECPredicatesDB.getPredicate(Relational, node)
14. FOR each child in node.Children
15. IF child.Object is a ValueExpr THEN
16. IF child.Object.Value is QoSTerm THEN
17. childTemplates = ParametricECFormulasDB.getTemplates(child.Object.Value)
18. appendAll(ECFormulasList, childTemplates)
19. QoSTermHoldsAtPredicate = ParametricECPredicatesDB.getPredicate(HoldsAt, child.Object.Value)
20. IF child is node.LeftChild
21. relationalPredicate.LHSOperand = QoSTermHoldsAtPredicate.QoSTermVariable
22. ELSE
23. relationalPredicate.RHSOperand = QoSTermHoldsAtPredicate.QoSTermVariable
24. END IF
25. IF initialNode is an AgreementTerm THEN
26. IF child is at initialNode.LeftSubtree THEN
27. QoSTermECConditions = QoSTermECConditionsDB.getPredicates(child.Object.Value)
28. FOR each predicate in QoSTermECConditions DO
29. IF predicate is NOT contained in monitoringRule.Body THEN
30. append(monitoringRule.Body, predicate)
31. ELSE IF
32. END FOR
33. END IF
34. IF child is at initialNode.LeftSubtree THEN
35. append(monitoringRule.Body, relationalPredicate)
36. ELSE
37. append(monitoringRule.Head, relationalPredicate)
38. END IF
39. ELSE
40. IF monitoringRule.Body does NOT contain any predicate THEN
41. QoSTermECConditions = QoSTermECConditionsDB.getPredicates(child.Object.Value)
42. appendAll(monitoringRule.Body, QoSTermECConditions)
43. END IF
44. append(monitoringRule.Head, QoSTermHoldsAtPredicate)
45. END IF
46. END IF
47. ELSE IF child.Object.Value is a CONST THEN
48. IF child is node.LeftChild
49. relationalPredicate.LHSOperand = child.Object.Value
50. ELSE
51. relationalPredicate.RHSOperand = child.Object.Value
52. END IF
53. END IF
54. END FOR
55. IF initialNode is an AgreementTerm THEN
56. IF child is at initialNode.LeftSubtree THEN
57. append(monitoringRule.Body, relationalPredicate)
58. ELSE
59. append(monitoringRule.Head, relationalPredicate)
60. END IF
61. ELSE
62. append(monitoringRule.Head, relationalPredicate)
63. END IF
64. END IF
65. append(ECFormulasList, monitoringRule)
66. return ECFormulasList
67. END Translate

For each visited node \( n_i \), the algorithm checks whether \( n_i \) represents an agreement term, or a logical operator, or a comparison operator. In case that \( n_i \) represents an agreement term, the algorithm processes recursively the precondition, if any, and its guaranteed states, by visiting the left and right child nodes of \( n_i \) respectively (lines 1–7). In case that \( n_i \) represents a logical operator, the algorithm again processes recursively the children of \( n_i \) (lines 8–11).

For each AST node \( n_i \) that represents a comparison operator (lines 12–13), the algorithm generates an EC-Assertion relational predicate (\( \text{relationPredicate} \)) of the same type as the comparison operator in \( n_i \) by retrieving a parametric EC-Assertion relational predicate from the corresponding repository (\( \text{ParametricECPredicatesDB} \)). The operands of the relational predicate are specified in accordance to the left and right child nodes of \( n_i \). The algorithm considers that both left and right child of \( n_i \) can be either a QoS term (lines 15–16) or a constant (line 47).

Whenever the algorithm visits a \( n_i \) child node that represents a QoS term (lines 16–64), the algorithm retrieves the parametric EC aware FTL templates, which are necessary for the QoS term monitoring, from the corresponding repository (\( \text{ParametricECFormulasDBs} \)) and adds them to the \( \text{ECFormulaList} \). Moreover, for each QoS Term node, the algorithm creates a HoldsAt predicate (\( \text{QoSTermHoldsAtPredicate} \)) that should contain a fluent for the translated QoS term by retrieving a parametric EC Assertion HoldsAt predicate from \( \text{ParametricECPredicatesDB} \) (line 19). It should be noted that the QoS term fluent is initiated by the formulas that are specified in the corresponding parametric FTL templates for the translated QoS term. The algorithm uses the QoS term fluent variable to set the corresponding operand of the \( \text{relationPredicate} \) depending on the relative position of the QoS term node (lines 20–24). More specifically, the algorithm decides whether to set the left (LHS) or right (RHS) operand with respect to the relative position of the QoS term node. In case that the QoS term node is the left child of \( n_i \), then the LHS operand is set. The RHS operand is set otherwise.

To further resume the \( \text{monitoringRule} \) build up, the algorithm checks whether the \( \text{initialNode} \) is an AgreementTerm (line 25). In case that the \( \text{initialNode} \) is an AgreementTerm and the QoS term node is located in the left sub tree of the \( \text{initialNode} \), the algorithm resumes by adding the parametric EC-Assertion predicates \( \text{QoSTermECCConditions} \) that are compulsory for triggering the QoS term monitoring in the \( \text{monitoringRule} \) (lines 26–33). It should be noted that, for each QoS Term, which is supported by EVEREST, there is a predefined set of parametric EC-Assertion predicates, which represent the necessary conditions for the QoS term computation. For instance this predefined set for Throughput is as follows:

\[
\forall \text{case : (CaseId); srvId: (_SrvId); sensorId: (_SensorId); d: (CaseId.d); t1, t2, t3, t4 : Time}
\]

\[
\text{ThroughputECCconditions} = \{ \text{Happens(e(_id1, _Snd, srvId, Call(_O), sensorId), t1, [t1,t1])} , \\
- \text{Happens(e(_id2, srvId, _Snd, Response(_O), sensorId), t2, [t1, t1+d])}, \\
\}
\]
The predefined condition predicate sets are stored in the repository `QoSTermECCConditionsDB`. Thus, the algorithm retrieves the QoS term condition predicates from `QoSTermECCConditionsDB`. For each parametric condition predicate \( p \), the algorithm checks whether the body of `monitoringRule` contains \( p \). In case \( p \) is not included, the algorithm adds \( p \) in the conjunctive list of `monitoringRule` body. Regarding the generated `QoSTermHoldsAtPredicate`, the algorithm adds the predicate to the `monitoringRule` body in case that the `initialNode` is an AgreementTerm and the current node \( n_i \) is located in the `initialNode` left subtree of `initialNode`, or to the `monitoringRule` head otherwise (lines 34–38).

In case that a child of the comparison operator node \( n_i \) represents a constant, the algorithm uses the constant value to set the appropriate operand of the `relationaPredicate` that is generated because of \( n_i \) (lines 47–52). Again, the algorithm decides whether to set the left (LHS) or right (RHS) operand with respect to the relative position of the constant node. In case that the constant node is the left child of \( n_i \) then the LHS operand is set. Otherwise, the algorithm sets the RHS operand. Finally, when all `initialNode` child nodes are visited, `monitoringRule` is appended to the final output of the algorithm, i.e. to the `ECFormulaList` (lines 45–46).

Table 6: Instantiation look-up table

<table>
<thead>
<tr>
<th>FTL EC template Placeholders</th>
<th>SLA@SOI SLA &amp; SLA Template Abstract Syntax Equivalent Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists t_3: \text{HoldsAt} \text{(Served}(P, \text{srvId, } N_1, t_3), t_1) \land (t_3 \leq t_1)) )</td>
<td>( \text{HoldsAt}(\text{Served}(P_2, \text{srvId, } N_2, t_4), t_1+d) )</td>
</tr>
<tr>
<td>( \neg \exists t_4, P_2, N_2: (P_1 \neq P_2) \land (t_4 \geq t_3) \land (t_4 &lt; t_1+d) \land \text{HoldsAt} \text{(Served}(P_2, \text{srvId, } N_2, t_4), t_1+d) )</td>
<td></td>
</tr>
</tbody>
</table>

To give an illustrated example of monitoring rules that `Translate` algorithm generates, assume the AST in Figure 48. The given AST is translated into:

\[
(\forall \text{case : } \langle \text{CaseId} \rangle; \text{srvId: } \langle \text{SrvId} \rangle; \text{sensorId: } \langle \text{SensorId} \rangle; d: \langle \text{CaseId}.d \rangle; t_1, t_2, t_3, t_4 : \text{Time})
\]

**Rule.case:**

\[
\begin{align*}
\text{Happens}(e(_id1, _Snd, \text{srvId, Call}(O), \text{sensorId}), t_1, [t_1, t_1]) \land \\
\neg \text{Happens}(e(_id2, \text{srvId, } _Snd, \text{Response}(O), \text{sensorId}), t_2, [t_1, t_1+d]) \land \\
(\exists t_3: \text{HoldsAt}(\text{Served}(P, \text{srvId, } _N_1, t_3), t_1) \land (t_3 \leq t_1)) \land \\
(\neg \exists t_4, P_2, N_2: (P_1 \neq P_2) \land (t_4 \geq t_3) \land (t_4 < t_1+d) \land \text{HoldsAt}(\text{Served}(P_2, \text{srvId, } _N_2, t_4), t_1+d)) \land \\
\text{HoldsAt}(\text{case}.\text{Throughput}(\text{ThroughputValue}), t_1) \land \\
\text{ThroughputValue} \geq T \Rightarrow \\
\text{HoldsAt}(\text{case}.\text{MTTR}(\text{MTTRValue}), t_1) \land \\
\text{MTTRValue} < M
\end{align*}
\]

Once the `ECFormulaList` is compiled, the `Instantiator` of the EVEREST RCG Translation component processes the parametric `ECFormulaList` templates and transforms them to operational EVEREST monitoring specifications. Besides the compiled `ECFormulaList`, the `Instantiator` component takes as input the java SLA object containing the SLA@SOI object that is assigned to EVEREST for monitoring by the Monitoring Manager, as well as, the monitoring configuration generated by the Monitoring Manager. The parametric templates contained in `ECFormulaList` are instantiated to operational EVEREST monitoring specifications by making the appropriate substitutions according to the following look up table.
CaseId                    SLA.UUID+AssignedSLAObject.Id
_SrvId                    InterfaceRef.UUID → service URL
_SensorId                 The id of the sensor id that provides the primitive request and response events of the monitored service

Caseid.d                  Constant configurable per SLA

More specifically the following substitutions are made:

- Each appearance of CaseId in a selected parametric template is substituted by a string, which contains the unique id of the monitored SLA plus the unique id of the java SLA@SOI object that is assigned to EVEREST for monitoring by the Monitoring Manager.

- Each appearance of _SrvId is substituted by the unique id of the interface reference of the monitored service. It should be noted that the substitution string should be identical to the value of the receiver and sender parameters of the request and response SLA@SOI primitive events that are generated by the monitored service, as defined in [30].

- Each appearance of _SensorId is substituted by the unique id of the sensor that provides the request and response SLA@SOI primitive events that are generated by the monitored service. It should be noted that substitution string should be again identical to the source parameter of the aforementioned primitive events, as defined in [30].

- Each appearance of CaseId.d is substituted by the constant that represents the time period within which a request to the monitored service is considered served or dropped upon the occurrence or non occurrence of a response from the service respectively.