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 – Evaluated Framework

Keywords:
Service Level Agreement, Service-Oriented Infrastructure, Planning, Negotiation

Due date of deliverable: 31st July 2011
Actual submission to EC date: 31st July 2011
Start date of project: 1st June 2008
Duration: 36 months

Lead contractor for this deliverable: UDO
Revision: V.0.8 (31th July 2011)

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

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>Public</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Date</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>14.6.2011</td>
</tr>
<tr>
<td>15.6.2011</td>
</tr>
<tr>
<td>28.6.2011</td>
</tr>
<tr>
<td>11.07.2011</td>
</tr>
<tr>
<td>21.07.2011</td>
</tr>
<tr>
<td>25.07.2011</td>
</tr>
</tbody>
</table>
Executive Summary

This document provides an update to the work done in work package A5 after Y1 and Y2. During the initial period mostly conceptual work and initial implementations work have been conducted. This work serves as a basis for the progress presented in this deliverable.

The progress in Y3 mostly encompasses extensions of the feature set and an improvement on already available features. The work conducted regards most of the work package’s tasks and also most of the implementation from Y2. The technical aspects of this work will be presented in this document including technical details. Conceptual work from the previous years has been kept stable, with the exception of the SLA model, which has been improved significantly and re-implemented. Furthermore the negotiation framework has been extended conceptually and is backwards compatible to the state in Y2. In Y3 we introduced an implementation of SLA translation, which so far has been available only as preliminary conceptual work. We have also extended the reasoning gateway of the EVEREST monitoring framework to increase the coverage of the SLA model. Another premier feature is the SLA editor which allows for natural editing of SLAs and SLA templates, which can be used during the adoption phase of the framework and during negotiations. Further extensions and improvements regard SLA monitoring, planning and optimization and the SLA registry.

Conclusively we have been able to successfully extend A5’s feature set in Y3 in alignment with the requirements by the use cases and the targeted objectives as stated by the DoW. Recommendations of the past reviews have been taken into account in Y3 and are referenced throughout the document. Many of our provided features provide results, which go beyond the state of the art, as will be explained individually. In addition based on our current progress we are able to provide an outlook on future directions for SLA management in general.

This document is structured to reflect the progress in Y3 according to the tasks in A5 according to the DoW. First, A5’s contributions will be presented from three different high level views. Then, the impacts on the overall framework architecture will be discussed. The main part of the document deals with our contributions in Y3 from a technical point of view. Finally, we provide a conclusion as well as bibliographical references in the appendix of this document.

1 Throughout the document the project periods will be referred to as Y1, Y2 and Y3.
# Table of Contents

1  Introduction ............................................................................................................ 10
   1.1  Context and Scope ............................................................................................ 10
   1.2  Document Overview ......................................................................................... 12

2  Contributions Overview .......................................................................................... 14
   2.1  Key Innovations .............................................................................................. 14
       2.1.1  SLA Model .............................................................................................. 14
       2.1.2  SLA Translation ..................................................................................... 15
       2.1.3  Customisable Negotiations ..................................................................... 16
       2.1.4  Negotiation Profiling .............................................................................. 16
       2.1.5  Planning for IaaS Using Computational Geometry .............................. 16
       2.1.6  Graphical Editing of SLAs ................................................................. 17
       2.1.7  Skeleton SLAM ....................................................................................... 17
   2.2  Framework Contributions ............................................................................... 17
   2.3  Task Level Activities ...................................................................................... 18

3  Architecture ............................................................................................................. 20

4  SLA Modelling ......................................................................................................... 21
   4.1  Introduction ..................................................................................................... 21
   4.2  Objectives, Concepts and Approach ............................................................. 21
       4.2.1  The Notion of “Service” is Ambiguous .................................................. 22
       4.2.2  An SLA Represents an Agreement ......................................................... 23
       4.2.3  An SLA(T) Describes a Service ................................................................. 24
   4.3  Implementation Progress ................................................................................. 25
       4.3.1  Y1 : Proof of Concept .............................................................................. 25
       4.3.2  Proof of Concept ..................................................................................... 25
       4.3.3  Prototype .................................................................................................. 26
       4.3.4  Final Implementation .............................................................................. 28
   4.4  Beyond SLA@SOI ............................................................................................ 30

5  SLA Registry ............................................................................................................ 32
   5.1  Data structures and Classes .............................................................................. 32
       5.1.1  Functional Interface .............................................................................. 33
       5.1.2  IRegister interface ................................................................................. 33
       5.1.3  IQuery interface ...................................................................................... 34
       5.1.4  Detailed Specification .............................................................................. 35
       5.1.5  IRegister .................................................................................................. 35
       5.1.6  IQuery ...................................................................................................... 35
       5.1.7  IAdmin .................................................................................................... 36
   5.2  New Features in Y3 .......................................................................................... 36
   5.3  Management Console ....................................................................................... 37

6  SLA Translation ........................................................................................................ 38
   6.1  Motivation ......................................................................................................... 38
       6.1.1  Problem Statement .................................................................................. 38
   6.2  Dependency Modelling ..................................................................................... 39
       6.2.1  Dependency Graphs ............................................................................... 39
   6.3  Assumptions for an Implementation ................................................................ 39
   6.4  Related Work .................................................................................................. 41
       6.4.1  Term Rewriting Systems ...................................................................... 41
       6.4.2  Semantic Technology ............................................................................. 41
   6.5  A Rule-based Approach ................................................................................... 41
       6.5.1  Covering complex translation scenarios ................................................. 41
       6.5.2  Performance analysis .............................................................................. 41
   6.6  Future Directions .............................................................................................. 45
6.7 Conclusion ................................................................................................................. 45
7 SLA Negotiation ........................................................................................................... 47
  7.1 Observations from State of the Art .......................................................................... 47
  7.2 Customizable Negotiations ..................................................................................... 51
  7.3 Profiling for Negotiations ....................................................................................... 53
8 Planning and Optimization ........................................................................................... 54
  8.1 Infrastructure as a Service ...................................................................................... 54
  8.1.1 Problem Statement ............................................................................................ 54
  8.1.2 Problem Modelling ............................................................................................ 55
  8.2 Software as a Service ............................................................................................... 62
  8.2.1 Translation Problem & Solution Approach ...................................................... 64
9 SLA Monitoring ............................................................................................................. 65
  9.1 Overall architecture of the SLA@SOI monitoring infrastructure ......................... 65
  9.1.1 Overview of EC-Assertion .................................................................................. 66
  9.1.2 Parametric templates for basic QoS terms ....................................................... 67
  9.2 Enhancement of EVEREST and EVEREST RCG ................................................ 68
  9.3 Testing and Validation ............................................................................................ 70
  9.4 Future Work ............................................................................................................ 71
10 Editing of SLAs and SLA Templates .......................................................................... 72
  10.1.1 Domain-specific Extensions ............................................................................ 73
  10.1.2 Creation and Editing Process ........................................................................... 74
  10.1.3 Negotiation process .......................................................................................... 74
11 Skeleton SLAM & SLAM Wizard ................................................................................ 76
12 Performance Evaluation .............................................................................................. 78
  12.1 Simulated Components ......................................................................................... 78
  12.2 Simulation Scenario ............................................................................................... 79
  12.3 Simulation Results ................................................................................................ 79
  12.4 Profiling ................................................................................................................ 80
13 Conclusions ................................................................................................................. 82
  13.1 Summary ............................................................................................................... 82
  13.2 Lessons Learned ................................................................................................... 82
  13.2.1 It is infeasible to provide truly generic solutions ............................................ 82
  13.2.2 Provided results need to be easily adoptable .................................................. 82
  13.2.3 A trade-off between requirements and realization needs to be found ............ 83
  13.2.4 Many scientific and engineering approaches need to be integrated for good results 83
  13.2.5 The choice of integration technology is a prime strategic concern ............... 83
  13.3 Outlook on Future Work ....................................................................................... 83
14 References ................................................................................................................... 85
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Relationship between the SLA model and human operators</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Historical Time-Line for SLA(T) Model and SLA Template Registry Implementations</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Screen-Shot of (Y3) SLA Template Registry GUI</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>Accessing an SLA registry</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Extending the dependency model of the SLA model</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Registration of a new SLA in the SLA-registry</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>Querying an SLA in the SLA-registry</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>A common 3-tiered service-oriented architecture</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>A dependency graph depicting relations among QoS parameters</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>The SLA translation components general architecture</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>A layered architecture for SLA translation</td>
<td>43</td>
</tr>
<tr>
<td>12</td>
<td>Generalized State Machine</td>
<td>51</td>
</tr>
<tr>
<td>13</td>
<td>Sequence Diagram showing Customized Negotiation</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>Computational geometry representation</td>
<td>56</td>
</tr>
<tr>
<td>15</td>
<td>Computational geometry representation of different plans</td>
<td>57</td>
</tr>
<tr>
<td>16</td>
<td>Class diagram for the implementation of computational geometry</td>
<td>58</td>
</tr>
<tr>
<td>17</td>
<td>Moving the request point to find alternative representations</td>
<td>59</td>
</tr>
<tr>
<td>18</td>
<td>Fragment statistics</td>
<td>62</td>
</tr>
<tr>
<td>19</td>
<td>Resource utilization of all servers</td>
<td>62</td>
</tr>
<tr>
<td>20</td>
<td>Hierarchical negotiation with the SW-SLAM in the middle</td>
<td>63</td>
</tr>
<tr>
<td>21</td>
<td>Organisation of EVEREST and EVEREST RCG</td>
<td>66</td>
</tr>
<tr>
<td>22</td>
<td>Partial Results of Testing with Use case Elements</td>
<td>71</td>
</tr>
<tr>
<td>23</td>
<td>The SLA editor showing a syntax-error</td>
<td>74</td>
</tr>
<tr>
<td>24</td>
<td>The SLA editor showing code-completion</td>
<td>74</td>
</tr>
<tr>
<td>25</td>
<td>The structure of the skeleton SLAM</td>
<td>76</td>
</tr>
<tr>
<td>26</td>
<td>Using the maven-plugin to generate a SLAM</td>
<td>77</td>
</tr>
<tr>
<td>27</td>
<td>The resulting B3-SLAM</td>
<td>77</td>
</tr>
<tr>
<td>28</td>
<td>Architecture of the SLAM-simulation</td>
<td>78</td>
</tr>
<tr>
<td>29</td>
<td>The setup for profiling SLAMs using a simulation environment</td>
<td>80</td>
</tr>
<tr>
<td>30</td>
<td>Profiling the simulation of the SLAM</td>
<td>81</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Achievements of A5 relating to DoW objectives ........................................ 12
Table 2: Relation of tasks to A5’s Y3 innovations...................................................... 18
Table 3: Performance Measurements of the SLA translation component ........... 45
Table 4: Server capacities ...................................................................................... 61
Table 5: Satisfaction ratio of requests with different shifting steps................... 61
Table 6: Terms supported by SLA monitoring......................................................... 70
Table 7: The simulation scenario applied for performance evaluation of SLAMs . 79
1 Introduction

1.1 Context and Scope

This deliverable presents the main achievements of work package A5 during Y3. Furthermore, it relates these achievements to the work done during the previous project periods. Finally, this document provides a recap of the overall accomplishment of work package A5 during the project’s whole duration.

In the Y1 of the project we have developed a conceptual SLA model as a focal point, around which we have explored the field of SLA management. The result of project Y1 was an ad hoc framework with major contributions from work package A5. Further details on the work done in Y1 are presented in the corresponding deliverable D.A5a.

In Y2, a project wide architecture was developed, containing the SLA managers as central components. The SLA managers, abbreviated SLAMs, are an output of the work of A5 and A1 and have been realized in a first version during project Y2. The A5 and A1 deliverables for Y2 introduce the SLAM architecture and discuss its functionality in detail.

This document reflects the innovations that have been incorporated during Y3 into previous results from A5. As such it provides an update on grounds of the Y2 deliverable in terms of functionality and architecture. Further changes encompass the extension of monitoring capabilities through the reasoning-component gateway and a new implementation of the SLA model. Since this deliverable focuses on the innovations in Y3, a sound understanding of the concepts presented in the preceding deliverables (D.A5a) helps in comprehending our newest achievements.

As in the previous project years the outputs of A5 need to be seen in the context of the overall architecture, which provides the big picture and lets the reader understand better the concept of the SLAMs, SLA monitoring and the SLA model itself. For the current results with regards to the whole framework architecture please refer to the deliverables from work package A1. In addition the results presented in this document can be best understood in the context of their application in the ORC, which is described in detail in B2’s deliverable. The adoption of the partially abstract concepts in A5 can be found in the SLA@SOI adoption guide and the SLA@SOI tutorial documents. Further we have contributed work that relates both to work packages A4 and A5 regarding the IS-SLAM and I-PAC. Similarly the negotiation framework has been advanced in a combined effort by work packages A2 and A5. Though the mentioned deliverables can be a great help in understanding this document, we intend this to be a largely self-contained document.

During the whole project duration the type of output from work package A5 has shifted from conceptual results to concrete implementations. In Y1 the results were mostly of conceptual nature, reflecting the experimental character of the conducted work and the prepared state of the art analysis. In Y1 we have developed concepts for managing SLA hierarchies by reflecting these hierarchies in an architectural way. To cater for the needs of the project’s use cases, we have developed a conceptual SLA model, which he have integrated directly with WS-Agreement. Based on this model we have defined an SLA lifecycle including template management, SLA negotiation, SLA provisioning, SLA monitoring and service adjustment. The implementation of this mostly conceptual work has been realized as part of the ad hoc framework as demonstrated at the end of project Y1.
In Y2 we have built upon our acquired understanding of the concepts which are required for SLA management by working out an holistic methodology for SLA management. This methodology can be seen clearly in the resulting component architecture of the SLAMs, which reflects a separation of concerns for the various parts of the SLA lifecycle. In Y2 we have implemented the core functionality of the SLAMs for the generic components. Further we have provided exemplary implementations for domain specific components within the SLAMs, which can be directly used in the context of the ORC. A major achievement in Y2 was the implementation of the SLA model, which supported Java- and XML-based representations. On top of this, we have implemented the template registry, which provides querying mechanisms on the basis of the SLA model. The distribution of templates is achieved using a custom publish-subscribe system. As a basis for negotiations we have researched and developed the protocol engine as a generic negotiation-protocol component, which supports a wide variety of negotiation protocols and which can be extended by framework adoptees. To provide a high degree of interoperability we further have developed the syntax converter as a conversion component between our SLA model and various other representations including WS-Agreement. We have further provided generic as well as domain specific components, which plan, provision, optimize and adjust services during an SLA’s life-cycle. As an additional feature, one that is situated outside the core-framework we have developed an SLA management console to provide intuitive access to a SLAM’s registered SLAs. Conclusively it can be stated that in Y2 we have laid the foundation for SLA management in form of implemented components, which have been improved and extended with new features during Y3.

Based on our achievements in Y2, we started out in Y3 with an already integrated platform. Now it was time to extend the already implemented features with pending innovations. To accomplish this, also further effort has been invested into preparing a comprehensive state of the art analysis. The SLAMs, as a basis of a lot of functionality in A5, have been improved mostly in terms of stability, but also in usability. A key innovation here is the advent of the skeleton SLAM, which is the scaffold of the G-SLAM, stripped from all domain-dependent implementation. As such the skeleton SLAM provides the basis for further adoptions of the SLAMs by the use cases and future industrial adoptees. This effort relates to the work on the SLA@SOI studio, which provides an integrated suite for the SLA@SOI framework. SLA@SOI studio is using the skeleton SLAM as the basis for generating a domain-specific version of a SLAM using a graphical user-interface. The SLA model, which was mostly conceptual in Y1 and an ad-hoc Java-based implementation in Y2 has been re-implemented to allow more flexibility, better querying mechanisms and most of all an easier adoption. SLA negotiation has been extended in Y3 by customizable negotiations, profiling mechanisms and a publish-subscribe system for negotiations. SLA translation has been researched and realized as a layered rule-based approach, which makes it easy for adoptees to realize a domain-specific SLA translation. The domain-specific implementations for the IaaS- and SaaS-layers of the planning-and-optimization components have been extended based on the work in Y2, with optimization schemes that specifically target those domains. SLA monitoring has been extended by a tighter integration of EVEREST with the SLA model, which now covers a broader range of the SLA model and its standard QoS terms. Finally a graphical editor for SLAs and SLA templates based on our SLA model has been developed. Its graphical user interface is expected to make it easier to adopt the whole SLA@SOI framework, since it integrates many previously separate tasks for validating an SLA.

Table 3 lists the objectives as stated in the DoW and how these have been addressed by work package A5 during the whole project.
**Table 1:** Achievements of A5 relating to DoW objectives

<table>
<thead>
<tr>
<th>Objective as per DoW</th>
<th>Achievement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical specification of SLAs</td>
<td>The technical specification has been achieved in an elaborate way by the conceptual work and implementation of the SLA model.</td>
</tr>
<tr>
<td>Service/SLA Template registry and discovery</td>
<td>The SLA registry and the SLA template registry both provide storing mechanisms. The template registry additionally provides a querying mechanism, based on the SLA model.</td>
</tr>
<tr>
<td>Translation of SLAs within or between layers within the architectures</td>
<td>A layered implementation for SLA translation has been realized using an event-action approach.</td>
</tr>
<tr>
<td>SLA negotiation service that links a service requester or consumer with a service provider or mediator</td>
<td>SLA negotiation has been realized in form of the SLAMs, which act as consumer, provider and mediator. Concretely the syntax converter and the protocol engine act respectively as access point and orchestrator for negotiations.</td>
</tr>
<tr>
<td>SLA optimization/scheduling/planning</td>
<td>The planning-and-optimization components have been developed as generic components with partial domain-specific implementations for reference scenarios.</td>
</tr>
<tr>
<td>SLA monitoring infrastructure</td>
<td>An SLA monitoring infrastructure, which deploys the event-calculus based monitoring framework of EVEREST has been developed, supporting a broad set of QoS terms.</td>
</tr>
<tr>
<td>SLA provisioning/activation/execution control</td>
<td>SLA provisioning and adjustment is available via the provisioning-and-adjustment component, which covers provisioning, adjustment and delayed execution of services.</td>
</tr>
</tbody>
</table>

**1.2 Document Overview**

Table 2 list the changes that have been incorporated into this document against the corresponding Y2 deliverable.
Table 2: Changes against the Y2 deliverable

The rest of this document is structured as follows. Section 2 provides an overview on the innovations in Y3. Section 3 gives an overview on the architectural impacts of our work in Y3. The final version of the SLA model is presented in Section 4. Section 5 provides an overview on updates to the SLA registry in Y3. Section 6 introduces the layered architecture for SLA translation. Section 7 provides information on new features for SLA negotiation. In Section 8 extensions to the previously available optimization routines for IaaS and SaaS are introduced. Section 9 discusses the extensions to the support for SLA monitoring by the EVEREST monitoring framework in Y3. The SLA editor is introduced in Section 10. Section 11 describes the skeleton SLAM and how it can be used to generate domain-specific SLAMs. Sections 12 and 13 finally provide a conclusion to the deliverable and bibliographical references, respectively.
2 Contributions Overview

This chapter provides an overview on the contributions by A5 during Y3 from three different perspectives. In Section 2.1 all key innovations are listed by themselves and briefly explained. In Section 2.1.7 the contributions are explained from the framework perspective. Finally Section 2.3 relates the presented innovations to the tasks in work package A5.

2.1 Key Innovations

The key innovations in Y3 target most of all better adoptability, extension of the feature set and an improvement of already available features. Better adoptability is required so that A5’s outputs can be adopted by other A-line work packages as well as the use cases. Furthermore the improved adoptability shall benefit later adoptees from industry. An improved adoptability has been reached through the following efforts: the skeleton SLAM, the reworked SLA model, SLA translation, extensions to SLA monitoring and the SLA editor. Furthermore we have continued on our research path and improved as well as extended the feature sets of most of our components beyond the state of the art. Feature improvements and extensions have been conducted in the following areas: SLA modelling, negotiation framework, SLA translation, SLA planning and SLA optimisation.

Over the whole project duration we have provided various advancements to the state of the art, which target key objectives of the scientific and technical goals of this project as stated by the DoW in Section B1.1.4. A5’s outputs provide the basis for consistent SLA management in terms of the SLAM components, which is embedded in A1’s framework architecture. Furthermore we have provided an end user platform for negotiations with the protocol engine and the syntax converter as key components. The latter is capable of handling representations of our own SLA model as well as a rendering which uses WS-Agreement as an adopted standard. The SLA model itself provides a mechanism to specify agreement terms across all layers for a service-oriented architecture using domain-specific extensions. The translation of SLAs and their respective agreement terms between layers is managed using our SLA translation component, which is introduced for the first time in this document. In A5 we provide generic interfaces and partially generic components for planning and optimization of SLAs and in addition, with cooperation of work packages A4, B2 and B4 we provide custom implementations as exemplary solutions for the mentioned components. We also contribute to SLA monitoring in terms of the monitoring manager and extensions of the EVEREST framework. Finally the infrastructure SLAM, which has been developed in cooperation with work package A4 allows for a quick adoption to changes in a virtualized execution environment through a tight integration with A4’s ISMs.

The remainder of this section will describe the individual key innovations briefly.

2.1.1 SLA Model

The SLA model is a central element to the project and it is used by most of the components of the SLA@SOI framework. Providing a high degree of flexibility and adoptability is imperative to its success within the project and beyond.

In Y2 a version of the SLA model was made available, which had a lot of hard-coded information. This meant that extensions to the SLA model also had to be
hard-coded, which decreases the model’s flexibility and its adoptability. Furthermore there were structural imperfections, which made it hard to implement elaborate querying mechanisms.

To resolve these limitations we have revised the SLA models concept and re-implemented it based on this revision. The new version of the SLA model does not incorporate extensions a priori, but reads them during run time from external documents. Furthermore our conceptual improvements to the SLA model allow for a more elaborate implementation of querying mechanisms.

The final SLA model as implemented in Y3 provides the desired querying mechanisms and it manages extensions in the form of external XML documents, which can be loaded at run time. This allows for a high dynamicity of extensions and decouples the core model from domain-specific definitions. In the course of re-implementing the SLA model, we have incorporated advanced validation mechanisms. The new version of the SLA model provides elaborate error information including type and line of errors in an SLA. For the validation of the Y3 SLA model, a tool with a graphical user interface has been made available.

Finally with regards to the SLA model it needs to be mentioned that we have finished the implementation of the SLA model in Y3 which has started with a conceptual model in Y1. During Y3 there has been a discussion within the consortium regarding the adoption of the final Y3 model, which represents a completely new implementation providing all envisioned features. The existing code base at beginning of Y3 however built on the Y2 ad hoc implementation of the SLA model. As an outcome the consortium has decided to continue the development of the SLA model, but not to adopt it in the framework at the current time. This decision is related to considerations regarding risk mitigation. As such the Y3 SLA model and the SLA template registry are not going to be used by most use cases. B6 is an exception in this regard due to special requirements regarding the SLA model. A final integration of the Y3 SLA model with the whole framework will be left to the open source community and future adoptees.

### 2.1.2 SLA Translation

The translation of SLAs is an inherent problem of multi-tier architectures. Since establishing SLAs makes most sense between different tiers and across domains, it goes hand in hand with SLA translation. SLA translation provides a way to convert one or more source SLAs to one or more target SLAs.

Until the beginning of Y3 mostly conceptual work on the topic of SLA translation had been carried out. As such we did not have an explicit implementation of an SLA translation component. Instead SLA translation between the software and infrastructure layer has been carried out implicitly in the SW-POC.

In Y3 we have researched additional approaches to solve the SLA translation problem. We have found that a rule-based approach, which embeds domain knowledge from system experts, is best suited. On this basis we have developed a layered architecture and a rule-based component for SLA translation, which can be used by the POCs to enhance planning and optimization, even during the negotiation phase.

The resulting SLA translation implementation is capable of capturing complex translation scenarios. Performance tests have shown that our method is also capable of handling large amounts of translation rules in a short period of time, which makes it applicable for negotiation time SLA translation.
2.1.3 Customisable Negotiations

The framework’s overall goal is to achieve many high quality SLAs as an outcome of SLA negotiation. The result of each negotiation depends on the suitability of the negotiation methodology to the negotiation’s circumstances and environment. Thus it is of benefit to develop concepts for customisable negotiations.

Currently the project’s negotiation framework depends on a protocol description, which is provided a priori. This means that the SLAMs might deploy a protocol which is not well suited for the present situation.

In Y3 we have introduced a pre-negotiation phase, which establishes the grounds for further negotiations. The negotiation protocols themselves are described as rules, which contain customisable parts. These customisable parts can be negotiated in the pre-negotiation phase as well as influenced by other framework components such as those in the business tier.

With our newest contributions it is possible to set up a custom negotiation protocol at run time, for each individual negotiation. The negotiation protocols can be customised by the negotiation agents themselves or by external policies.

2.1.4 Negotiation Profiling

As mentioned above, it is a major objective of this project to achieve a high number of quality SLAs through the use of our negotiation framework. In real world applications providers and customers do not always adhere to established SLAs. Thus it is important to adapt to historical data about negotiations and SLAs with regards to individual negotiation parties.

At the end of Y2 no mechanism which would allow taking into account past experiences in negotiations has been available in our project.

In Y3 we have developed a profiling mechanism, which takes account of the success of negotiations with other parties. Based on this the negotiation framework establishes custom protocols and policies based on historical negotiation data for individual parties.

Our approach works well for classifying negotiation parties and adapting the effort which is being put into establishing new SLAs with those parties. Thus the negotiation framework focuses on fruitful negotiations and omits parties with low potential.

2.1.5 Planning for IaaS Using Computational Geometry

Planning for IaaS involves the generation of advance reservation plans that take many considerations into account. The computation of those plans needs to happen quickly since it is carried out during negotiation time. In addition the reservation plans need to be assessed quickly for feasibility.

At the end of Y2 we have not used an optimized representation, which has led to a mixing of optimisation algorithms with representations, encoding implicit assumptions. This has led to a problem while implementing new algorithms and targeting new optimisation objectives due to the emerging complexity.

In Y3 we have employed an efficient representation for advance reservation plans which allows us to incorporate many dimensional criteria, without increasing the implementation complexity significantly.
The result is a flexible approach to model advance reservation criteria incorporating time and price decisions, which also allows for overbooking scenarios.

2.1.6 Graphical Editing of SLAs

A crucial task during the adoption of the SLA@SOI framework is to edit valid SLA templates. Furthermore template-editing is also of interest for semi-automatic negotiations.

During Y2’s use case demonstrator-phase we have experienced that creating valid SLA templates entails a lot of steps, which are not transparent to the user. The various validation steps were disparate and information on problems was sparse or not easy to understand. In addition SLAs were only XML-based, which is hard to read in case of complex SLAs.

In Y3 we have developed a graphical editor that integrates the various validation steps and allows for vocabulary extensions. Furthermore the editor uses a JSON-like representation which is easy to read and write even for non-technical personnel.

The resulting editor significantly decreases the effort required to create valid SLA templates. It can also be used for template-editing during negotiations via a provided API. The SLA editor has been developed so that it can be integrated as part of the SLA@SOI studio.

2.1.7 Skeleton SLAM

At the end of Y2 we have delivered a working version of the G-SLAM which does not contain any domain-specific implementation. This component is complex in itself and used throughout all use cases as a central component for negotiations and SLA management.

The preparation by the use cases for their respective Y2 demonstrators has shown, that the adoption and creation of custom implementations of SLAMs is one of the major hurdles to overcome. At end of Y2 the G-SLAM was available including documentation of its usage. The limitation to this approach is that it takes significant time to create a runnable, albeit mostly empty, custom SLAM.

During Y3 we have provided additional information on how to adopt the G-SLAM and we have developed the skeleton SLAM. The skeleton SLAM is an extension of a G-SLAM, which incorporates template-components, which can readily be used for domain-specific implementation. The skeleton SLAM is combined with a maven-plugin, which generates a domain-specific SLAM from the skeleton SLAM which can be further developed as its own project. The resulting SLAM can also be started immediately in the OSGi-integration environment and as such it can be embedded in the running platform.

The skeleton SLAM leads to a better adoptability of the SLAMs, which in turn allows the adoptee to focus efforts on functional implementations. Furthermore the skeleton SLAM is being used as part of the SLA@SOI studio, to generate SLAMs using a graphical configuration wizard.

2.2 Framework Contributions

Due to the central theme of work package A5, our developments throughout the project as well as in Y3 specifically have an impact on a large portion of the overall framework. We have focused our work mostly on the SLAMs themselves
as well as on the SLAMs’ internal components. The SLAMs are deployed as central components in all use cases and thus have an impact in all layers of an service-oriented architecture. Further our work has an impact on many components through the SLA model. Finally results from work package A5 are part of the SLA@SOI studio.

Our work on the SLAMs entails the skeleton SLAM and component updates to the planning-and-optimisation component, provisioning and adjustment component, protocol engine, syntax converter, SLA registry, SLA template-registry, monitoring manager and publish-subscribe system.

In addition we have contributed updates to the SLA model, which are being used in the SLAMs, the low-level monitoring system and the business-management components.

The generation of skeleton SLAMs and of skeleton components for the POC and PAC are part of SLA@SOI studio. Furthermore the SLA editor has been developed to work on its own, as well as to integrate well with SLA@SOI studio.

The development for planning and service adjustment are shared with work package A4, since they find application in A4’s domain. The SLA editor, as well as the skeleton SLAM is being used for the SLA@SOI studio, which is part of work package A1. The innovations to the negotiation framework are being shared partially with work package A2.

Thus the results of work package A5 find a wide adoption throughout the framework and beyond through the SLA@SOI studio. The detailed contributions to the mentioned components can be found in the subsequent sections of this deliverable.

### 2.3 Task Level Activities

This section provides an overview on the activities in work package A5 from the task level perspective. The organisation of the document is aligned with the task level view as can be seen in Table 3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA5.1: SLA Modelling</td>
<td>Section 4</td>
</tr>
<tr>
<td>TA5.2: Service/SLA Template Registry and Discovery</td>
<td>Section 4 &amp; Section 5</td>
</tr>
<tr>
<td>TA5.3: SLA Translation</td>
<td>Section 6</td>
</tr>
<tr>
<td>TA5.4: SLA Negotiation</td>
<td>Section 7</td>
</tr>
<tr>
<td>TA5.5: SLA Planning and Optimisation</td>
<td>Section 8</td>
</tr>
<tr>
<td>TA5.6: SLA Monitoring</td>
<td>Section 9</td>
</tr>
<tr>
<td>TA5.8: SLA Interface Access Layer</td>
<td>Sections 10 &amp; 11</td>
</tr>
</tbody>
</table>

**Table 3: Relation of tasks to A5’s Y3 innovations**

All tasks with the exception of TA5.7 (SLA Provisioning and Adjustment) have progressed during Y3. A summary of the task level progress follows.
1. The SLA model has been improved beyond the state of Y2 conceptually and it has been re-implemented. Furthermore additional tooling for validating SLAs has been provided.

2. The SLA-template registry has been re-implemented on the basis of the final SLA model to allow for complex template queries.

3. A concept for a layered architecture for SLA translation has been devised. The concept has been implemented and tested for performance.

4. Customisation of negotiations has been made available in the protocol engine; Profiling of negotiations has also been researched and integrated in the negotiation framework.

5. Advanced concepts for IaaS reservation using computational geometry have been developed and implemented.

6. The set of the supported SLA standard QoS and core terms covered by the EVEREST monitoring framework has been extended through the enhancement of the reasoning gateway of EVEREST which now supports the translation of most SLA terms.

7. Task TA5.7 has not been advanced, since a workable state has been achieved in Y2. This is in alignment with the requirements, as stated by the use cases for Y3 features.

8. The SLA editor has been developed from scratch to improve the usability of the SLA model, by allowing a simplified creation of SLAs and SLA templates.
3 Architecture

The architecture as developed in collaboration with A1 during Y2 has been implemented in A5 already in Y2. This work mostly concerns the SLAM architecture in A5, including interfaces and interactions to other components in the framework, which rely on the SLAMs. All interactions have been realized in the course of Y2 with final contributions in Y3.

The architectural definition of A5 components has been found well suited to also support the work in A5 during the final project period. As such no major changes to the architecture regarding A5 have been conducted. A potential change would have been to provide a new, separate component for the SLA translation component, which has been discarded. The SLA translation component is only used by the POCs and it is not required by all SLAMs. As such it does not constitute a generic feature. Furthermore there has been an extension to the negotiation interaction between SLAMs, which target partially A2’s B-SLAM. The interaction has been widened to allow for customization and profiling of negotiations. Details of this change in the negotiation interface will be provided in Section 7.

The fact that few changes are required reflects the stable nature of the architecture. During the development in Y3 it became apparent that from the perspective of A5 most required features can be implemented without any changes to the architecture or even modifications on the interface-level.

Due to the stability of the architecture and the envisioned future directions of individual components in A5 we do not foresee any substantial changes necessary to the architecture.
4 SLA Modelling

4.1 Introduction

This section describes the SLA@SOI approach to modelling SLAs and SLA Templates, and the evolution of the SLA(T) model from initial prototype to final (Y3) implementation. The complete technical details of the final SLA(T) model are complex and require more space than available in the present document. As such, we present here only a brief and high-level overview. The complete technical details can be found instead in [16], [15] and [14].

This section is structured as follows. Section 4.2 first recaps the main objectives, concepts and approach informing the SLA(T) modelling efforts. Section 4.3 then charts the realisation of the SLA(T) model over the course of the project, in order to highlight and explain the key developments in Y3. Section 4.4 concludes with a brief look at the continued development of the model beyond SLA@SOI.

4.2 Objectives, Concepts and Approach

The primary objectives, underlying concepts and basic approach of the SLA(T) modelling activity have been explained in detail in previous versions of this deliverable, in particular: D.A5.a (M12) Sections 2 (Essential Concepts) & 4.1 (Template Registry), and D.A5.a (M26) Sections 4.1 (SLA Model High-Level Overview) & 5.2 (Template Registry). For present purposes, we provide only a brief summary of the most important aspects.

The primary objective of the SLA(T) modelling task is:

- to specify a model for SLA(T)s which is sufficiently specific and detailed to support those aspects of SLA(T) management which are common to all domains (specifically: those encapsulated by the functionality of the Generic-SLAM), yet at the same time sufficiently generic and extensible to support customisation to domain-specific requirements (as encapsulated by different use case requirements).

To achieve this objective we adopted an extensible modular approach to modelling. The main effort in A5 has been to develop a core set of model modules – referred to as “vocabularies” – encapsulating the following information:

- a vocabulary model: specifying the basic structure of a vocabulary (module), and serving as the foundation for all other modules,
- a customisable, generic constraint language: providing a common mechanism for the description of constraints on service delivery (e.g. for describing service level objectives).
- interface descriptions: corresponding to the traditional functional “service description”.
- SLA and SLA Templates: encapsulating details of an agreement between service provider and customer, the terms of which specify guarantees on the functional and qualitative behaviour of services and/or actors.

---

2 The acronym “SLA(T)” is used to denote “SLA and/or SLA Template”.

• Common quality of service (QoS) metrics: including: service availability, mean-time-to-failure, completion, adopted standards, repudiation, and others.

Individual use case WPs are then responsible for developing additional vocabularies as extensions to the A5 core, in order to support their own domain-specific use case requirements.

A secondary objective is that the SLA(T) model should be specified in a language and technology independent manner: the rationale being that different problem domains would be best served by different dedicated syntactic formats and tools. For example:

• XML would be likely to be employed for the exchange of documents between components, or for simple configuration tools,
• Semantic-based search & discovery tools would employ semantically annotated descriptions (e.g. OWL-S for service descriptions),
• Rule-based representations (e.g. Prolog, Drools) would be employed by various reasoning mechanisms (e.g. for planning, adjustment and monitoring).

Accordingly, the SLA(T) model was conceived as an abstract syntax: essentially a universal reference model of SLA(T) document content. In practical application, the abstract syntax must be instantiated, in whole or in part, by an appropriate concrete syntactic format – e.g. XML, OWL, Drools, human-readable formats – and/or by different coding languages – e.g. Java, Objective-C, Prolog.

Finally, there are a handful of concepts, or assumptions, which are fundamental to the SLA(T) model but which, although obvious to some, have proven surprising or even confusing to others. For the sake of clarity, therefore, we dedicate the following sub-sections to explanations of these key assumptions, namely:

• the notion of “Service” is ambiguous,
• an SLA represents an agreement,
• an SLA(T) describes a service.

4.2.1 The Notion of “Service” is Ambiguous

One of the most surprising results of the conceptual analysis underlying the SLA(T) model, is that there is no clear, common notion of what a “Service” is. This question has been discussed at length throughout the SLA(T) modelling process, but no real consensus has ever been reached. It seems there are as many valid interpretations of “service” as there are partners in SLA@SOI. To illustrate, the term “service” variously denotes:

i) a functional type – i.e. services are distinguished just by their functional capabilities, independent of who provides the service, the endpoints at which it can be accessed, or how many times service operations are invoked.

ii) a distinct execution instance – i.e. the execution of a specific service operation at a specific endpoint in response to a specific invocation.

iii) anything in between the two extremes identified by ‘i’ and ‘ii’.

iv) a running BPEL process instance.

v) the fulfilment of a specific SLA – i.e. each SLA, agreed between provider & customer, describes the provision of a service (see also Section 4.2.3 below).
Along with the conceptual differences, come differences in how services are described. In at least two use case scenarios, for example, it was required to describe services for which no (explicit) service operations could be identified – the only explicitly represented fact about the service being its name.

To cope with these differences, the SLA(T) model simply leaves the concept of “service” undefined. Instead it provides a mechanism by which different interface, operation, invocation & endpoint instances (or classes of instances) can be identified and combined, in arbitrary ways, to effectively specify custom “service” concepts on a per SLA(T) basis.

4.2.2 An SLA Represents an Agreement

The assertion that “an SLA represents an agreement” may appear trivial - given that “SLA” is an acronym for “Service Level Agreement” - but it is never-the-less important to be precise about what the assertion implies. The point to appreciate here is just that the term “agreement” is an intentional term: it denotes a state of mind rather than any tangible, physical object. An agreement is an understanding that exists between two (or more) parties. It is not an object that you can point to.

This intentional notion of agreement is significant because it has strong consequences for negotiation, and in particular the notion of automated agreement negotiation. Assuming that only humans have minds, and hence only humans are capable of entering into agreement, then the negotiation process must necessarily entail human actors in a decisive role.

For modelling purposes, however, we are not concerned with intentionality, but with representation. We state only that “an SLA represents an agreement”, not that “an SLA is an agreement”. That is; we are only concerned with the physical entities which serve to evidence, or represent, an agreement. Specifically, the SLA(T) model defines an SLA(T) as a physical document. The model itself is therefore a document model: as illustrated in Figure 1, it specifies the physical content of SLA(T) documents.

![Figure 1: Relationship between the SLA model and human operators](image)

To be precise, the model actually distinguishes the following three kinds of SLA(T) document:

---

3 EPR Hosting and Infrastructure
i) a signed SLA: representing an actual agreement, i.e. one that has been agreed between a specific provider and specific customer,

ii) an unsigned SLA: representing a potential, or proposed, agreement, between a specific provider and specific customer,

iii) an SLA Template: similar to an unsigned SLA but with no specific customer identified, representing a set of potential agreements with different customers. In addition, the SLA Template may contain unbound variables (or slots), supporting customisation.

The lifecycle relations between these different forms of SLA(T) are detailed in Deliverable D.A5.a (M12), Section 2.4 SLA Lifecycle.

4.2.3 An SLA(T) Describes a Service

In the traditional notion of a “service description” (cf. WSDL) a service is essentially described by its purely functional characteristics, which are typically captured in the form of function signatures or workflows. SLA(T)s, in addition to functional characteristics, also capture non-functional, or qualitative, aspects of service delivery, which are typically referred to as quality-of-service (QoS) characteristics. Example QoS terms include:

- **temporal properties** – e.g.
  - service availability,
  - maximum completion-time of operations,
  - mean-time-to-recovery (following a fault in service delivery).

- **usage constraints** – e.g.
  - maximum number of transactions per day,
  - maximum storage capacity (for data-storage services).

- **fiscal properties** – e.g.
  - unit cost per invocation,
  - penalties in case of SLA violation.

It should be apparent that, while QoS terms do not describe the service in the traditional functional sense of a “service description”, they do never-the-less say something about the nature of the service. QoS terms are non-functional service descriptions. Accordingly, we assert that an SLA(T) does not merely include (or refer to) a “service description”, an SLA(T) is a service description in its own right: it describes both the functional and non-functional characteristics of a service. To avoid ambiguity, the SLA(T) model refers to a traditional functional service description (cf. WSDL) as an “interface”.

The main impact of this assertion, beyond terminology, is that it changes the way we approach service discovery. The SLA(T) model provides a rich description of service characteristics and we wish to exploit this richness for discovery. Instead of simply searching for a data-storage service, a potential customer should be able to search for the cheapest, the fastest, or the most secure data-storage service.

As described in detail in Deliverable D.A5.a (M12), the focus for this enriched model of service discovery in SLA@SOI is the SLA Template Registry. SLA Templates serve as both the means by which a provider describes the services they offer, and the means by which customers can describe their requirements. In order to support this “template-based” discovery, the development of the
SLA(T) model has necessarily been *tightly coupled* to the development of the SLA Template Registry. In the following section we briefly chart the history of this co-development.

### 4.3 Implementation Progress

This section presents a brief history of the realisation of the SLA(T) model. As noted in the previous section, the development of the model is tightly coupled to the development of the SLA Template Registry. Indeed the SLA(T) model and SLA Template Registry are essentially two aspects of a single system: registry functions are specified in terms of the model, and the model is designed to facilitate registry functions.

The following sections briefly trace the development of the SLA(T) model and SLA Template Registry from initial conception in Y1 to final implementation in Y3. Figure 2 presents a rough timeline as a guide (note that all work on the model/registry was suspended for several months from M19 due to effort being diverted to WP A1 for an overhaul of the framework architecture).

![Figure 2: Historical Time-Line for SLA(T) Model and SLA Template Registry Implementations](image)

#### 4.3.2 Proof of Concept

The bulk of activity in Y1 was devoted to conceptual modelling of SLA(T)s, the results of which are reported in detail in Deliverable D.A5.a (M12), and are not repeated here (Section 4.2 above reviews the most significant aspects).

In parallel to the conceptual modelling, however, a first ‘proof of concept’ implementation of the SLA Template Registry query mechanism was also developed: the goal being just to explore the logical aspects of query semantics and their implications for SLA(T) content. To this end, the registry & query mechanism were implemented entirely in Prolog, with SLA Templates encoded in predicate form in distinct Prolog modules. The basic approach can be summarised by the following Prolog rule:

```
query(SLAT_ID, REQUIRED_QOS) :-
    SLAT_ID:guarantee(OFFERED_QOS),
```
more_stringent(OFFERED_QOS, REQUIRED_QOS).

Briefly, both queries and SLA Templates comprise a set of QoS constraints: REQUIRED_QOS & OFFERED_QOS (resp.). A template satisfies a query just if the constraints expressed in the template (OFFERED_QOS) are more stringent than those expressed in the query (REQUIRED_QOS). A query for the QoS constraint completion_time(S) < 5s (where S denotes some service), for example, would be satisfied by an SLA Template offering a completion_time(S) < 4s, but not by one offering a completion_time(S) < 6s.

Complication arises in the query process from various factors - such as:

- arbitrary logical combinations of constraints (i.e. the logical operators: AND, OR, NOT),
- relations between comparison operators - e.g. ‘<’ is more stringent than ‘<=’,
- metric unit conversion (equivalent constraints expressed in different units).

In principle, however, we can state that query resolution is essentially a problem of constraint satisfaction, and that the efficacy of the template-based discovery approach rests primarily on the manner in which constraints are expressed and represented in SLA(T)s. Similar arguments can also be made for other key functional components of the Generic SLAM in particular for planning & optimisation, and provisioning & adjustment. The initial ‘proof of concept’ implementation served just to formulate requirements for a common constraint language, as a basis for the representation of QoS guarantees in SLA(T)s, and in support of Generic-SLAM operations.

4.3.3 Prototype

The goal of the next phase of implementation (roughly the first half of Y2) was to move from the conceptual and ‘proof of concept’ stage (above), to a concrete Java realisation of the SLA(T) model and SLA Template Registry. Development of this Java API was hampered by two factors:

i) strict time limits: the Java API for the SLA(T) model had to be completed before development of the SLA@SOI framework components could begin, since the functional capabilities of these components depend in large part on the SLA(T) model.

ii) insufficient use case requirements: in particular, details of the specific QoS guarantees to be employed in use case SLA(T)s were missing. Various attempts were made to elicit this information from use cases prior to implementation of the Java API, but very little concrete information was received.

Because of these constraints, it was possible to implement only a rudimentary – or prototype - version of the SLA(T) model, focusing only on the "core" features required for the Generic-SLAM, and ignoring many aspects relating to

---

4 As described in section4.2.3, the SLA(T) model per se is an abstract syntax which is language & technology independent. For clarity: the Java API is just one possible interpretation of the model into a specific language & technology. In particular, the API makes various assumptions (such as URI formatting for universal unique identifiers) which are not strictly required by the model.
'vocabularies' and domain-specific use case extensions (see ‘primary objective’ in Section 4.2 above). This core model comprised:

- a small set of data primitives,
- a generic constraint language,
- a model for “interface” documents (i.e. functional service descriptions),
- a model for SLA(T) documents (combining the preceding elements to define QoS guarantees), and
- a handful of common “QoS metrics” : generic terms such as “availability”, “accessibility”, “completion-time”, which were considered to be useful for all use cases.

An XML Schema version of the model was then derived from the API (for XML (de)serialisation), and a first version of the SLA Template Registry was constructed, providing basic mechanisms for the persistent storage and retrieval of SLA(T)s. Template-based querying, as an advanced (i.e. complex and time-consuming) feature of the registry, was not considered an immediate priority, and hence was neither implemented nor indeed fully supported by the model API.

The prototype API mirrors the SLA(T) model closely at a coarse level. At a fine level, however, various simplifications were made for the sake of programming convenience. The model, for example, states that identifiers, path expressions, external references, constants and operators are all distinct types of expression, while the Java API glosses over these differences, treating them all as opaque strings - i.e. the API is weakly typed with respect to the model. These same simplifications carry over to the XML Schema. Accordingly a validation tool was also developed: to control the validity, with respect to the SLA(T) model, of SLA(T) documents constructed with the API, or in accordance with the XML Schema. This validation tool was incorporated into the SLA Template Registry to ensure that only valid SLA(T) documents could be registered.

In sum, the prototype comprised a Java API (encapsulating a reduced version of the SLA(T) model), an XML Schema (derived from the API), a validation tool, and the SLA Template Registry.

It was only after this prototype was completed and released that concrete use case requirements begin to appear. To collect and manage these requirements, a wiki page was created: providing specifications of all QoS related terms, and open to contributions from all partners. These terms were then incorporated into the core model API in a largely ad-hoc fashion, employing a very primitive hard-wired “extension” mechanism.

The main issues raised by use cases, however, concerned usability. In particular, the weak typing system provided little guidance for novice SLA(T) developers unfamiliar with the details of the SLA(T) model. The documentation for the model, although complete, was never-the-less

5 The XML Schema and (de)serialisation mechanisms were developed as part of the Syntax Converter component (described elsewhere in this deliverable).

6 The use-case requirements were typically expressed as “how can we use the API/XML-Schema to state that …” or “feature … is missing”. So we can only suggest that the use-case developers needed something concrete to criticise before they could formulate their needs.
terse, complex and technical, and thus intimidating. Finally, the validation tool, as a prototype, provided only minimal, and not very user-friendly, information about errors. The learning curve, in short, was steep and high. As a result, a significant amount of effort in WP A5 was devoted to hands-on tutoring of use case developers. These usability issues were addressed in the final (Y3) implementation, described in the next section.

4.3.4 Final Implementation

The final phase (roughly M23 onwards) in development of the SLA(T) model and SLA Template Registry was devoted to meeting in full the task objectives (described in Section 4.2), and to tackling the various issues raised by experience with the Y2 prototype. The work proceeded in four phases:

i) formal specification of the vocabulary model. As discussed in Section 4.2, the SLA(T) model is intended to be modular, and we refer to modules as “vocabularies”. The “vocabulary model” specifies a “vocabulary”, and can be considered as a kind of meta-model for different components of the SLA(T) model.

ii) (re-)formalisation of the SLA(T) model in terms of the vocabulary model. The opportunity was also taken to fix various issues with the existing (Y2) model. In particular, the Y3 version has clearer semantics, a much stronger typing system (supporting improved validation), and increased support for metric unit conversions and automated constraint satisfaction.

iii) complete overhaul of the Java API, XML-Schema and validation mechanism.

iv) implementation of the SLA Template Registry template-based query mechanism.

The most significant aspect of the Y3 implementation was the introduction of the vocabulary model. In essence, the vocabulary model is a generic model of document content: it defines the basic notion of a “document”, and provides various constructs for specifying the structured content of (the whole or part of) a document. It can be thought of as playing a role similar to the XML-Schema specification\(^7\), except that it is not specific to XML or any concrete syntax. The vocabulary model defines a vocabulary as a document, and the model is itself, recursively, a vocabulary specifying the content of vocabularies (cf. the XML-Schema for XML-Schema).

Vocabularies can be specified from scratch, or they can build on, extend or incorporate elements from other vocabularies. The reformulated SLA(T) model builds on the vocabulary model, and comprises the same ‘core’ as described in Section 4.3.2. The complete technical specification of the Y3 SLA(T) model can be found at [14]. This specification replaces all previous versions of the SLA(T) model.

In terms of the Java API, the impact of the vocabulary model was two-fold. First, a new API was developed encapsulating the vocabulary model. Since the vocabulary model is a document model, this API has the character of a document model.

---

\(^7\) The XML-Schema specification (see: [http://www.w3.org/standards/xml/schema](http://www.w3.org/standards/xml/schema)) specifies the content of XML-Schemas, instances of which, in turn, specify the content of XML documents. The “vocabulary model” specifies the content of “vocabularies”, instances of which, in turn, specify the content of documents.
object model (DOM): documents are modelled as node hierarchies, and navigation is performed by traversing the arcs between nodes. The upshot is that, in purely functional terms, everything that could be done with the prototype (Y2) Java API can now be achieved using only the (Y3) API for the vocabulary model. For backwards compatibility, however, the prototype API implementation was retained (with modifications to accommodate model improvements) but re-engineered in the form of ‘proxies’ (i.e. ‘wrappers’, operating over the vocabulary model API).

Second, and of more significance with regards the modelling objectives, is that the ad-hoc “extension” mechanism hard-coded into the prototype API to support use case specific requirements, has been completely discarded. Instead, all the model extensions required by use cases can now be encoded entirely in the form of vocabulary documents: with no Java coding, or API changes required.

Use case developers were also supported by the following:

- a much simplified XML-Schema realisation of the SLA(T) model,
- decoupling of the serialisation mechanisms from the Syntax Converter. The (Y2) prototype required a running instance of the SLA@SOI framework just to parse an XML formatted SLA(T). The serialisation mechanisms in Y3 are provided as stand-alone tools.
- greatly improved error reporting in the validation mechanism (for an illustration: see the “Validation Report ...” window in the screenshot, Figure 3),
- a graphical user-interface for the validation tool,
- on-line examples and tutorials,
- a tool, similar to javadoc, for the automatic generation of html documentation for vocabularies: offering a consistent, navigable and easily maintained source of model specifications (much of the documentation at [14] was generated with this tool).

All the work just described was completed and released to the project in M30. Shortly afterwards, however, an executive project decision was taken not to adopt the finished SLA(T) model in the SLA@SOI framework. This decision was based on two key considerations:

i) the adoption of the Y3 SLA(T) model and modified API was considered high risk given the short time remaining for development,
ii) use cases reported that their requirements could be fulfilled without adoption of the new model.

As a result, the SLA@SOI framework continues to use the incomplete, prototype SLA(T) model from Y2. The SLA(T) model as officially delivered by WP A5, however, is the final, complete (Y3) version described in this Section.

As a result of this decision, the remaining effort in Y3 – devoted to the finalisation of the SLA Template Registry and template-based service discovery – has been conducted “offline” with respect to the SLA@SOI framework. Briefly, the SLA Template Registry has been redesigned and rebuilt from the ground up as a stand alone application, fully exploiting the features of the final SLA(T) model. The registry incorporates all the tools mentioned above (e.g. multi-format serialisation, validation, auto-generation of vocabulary documentation, tutorials) and is now also furnished with a graphical administrator interface (Figure 3). In keeping with the design principles of the SLA(T) model, the registry is also constructed in an open modular fashion, with plug-in extensibility for novel serialisation formats, back-end persistence, automated metadata generation and custom query formats/resolvers.
Since the final versions of the SLA(T) model and SLA Template Registry are not officially part of the SLA@SOI framework, their respective APIs have not been included in the project’s managed Sourceforge code repository. Instead, the source code for both the model and registry can be accessed from: https://sourceforge.net/apps/trac/sla-at-soi/wiki/SlaModel. At the time of writing, the registry implementation remains a work in progress, and hence the registry source code is subject to change.

4.4 Beyond SLA@SOI

The historical development of the SLA(T) model described in the previous Section can be best described as explorative. Existing standards, in particular WS-Agreement [4] and WSLA [35], were studied and provided significant input, but on the whole the modelling task was approached with very few preconceptions about the final outcome. Instead, it was decided that model development should be driven entirely by use case requirements and feedback.

As described in Section 4.3.2, however, it proved almost impossible to elicit use case requirements until well into the implementation phase. In the absence of specific goals the SLA(T) model defaulted to generality, extensibility and customisability. Even the notion of “service”, as we saw in Section 4.2.1, remains technically undefined in the SLA(T) model. If five different use cases scenarios each employ a different notion of “service”, then from the perspective of A5, the concept of “service” is necessarily a domain-specific concept – however surprising this may be.
The upshot is that the hard, detailed SLA(T) modelling has, in the end, been conducted by the use cases themselves: since it is only in the use case scenarios that the semantics of the SLA(T) model abstractions could be fleshed out in terms of hard technological artefacts. The logical next step in the development of the SLA(T) model, therefore - looking beyond SLA@SOI - is a reassessment of the model, and in particular the model semantics, in the light of use case experiences. To this end, we will be continuing to work on the SLA(T) model within the Future Internet PPP project, FI-WARE [1].
5 SLA Registry

The SLARegistry is a persistent store for SLAs, and historical SLA-state information. It is maintained directly by the ProtocolEngine and can be queried by all internal SLAM components. It also exposes query web services via the SyntaxConverter.

Architecturally, there is one registry for customer-facing SLAs, and another for provider-facing SLAs. As well as the SLAs themselves, the registry also maintains historical SLA status information (for auditing). The registries serve as an archive for completed SLAs as well as those which are currently in effect.

The SLARegistry is defined by two interfaces which provide access to the implementation of the <<query>> and <<register>> stereotype interactions. The register interaction stores a SLA in the registry, with pointers to dependent/depending SLAs. The second interaction, query, retrieves a SLA, its status, its status history, and the dependent/depending SLAs. For administrative access, the interface <<admin>> allows the internal management of this registry. Operations such as cleaning and flushing are supported.

Each SLAM contains an instance of SLA registry and can be accessed as depicted in Figure 4.

![Figure 4: Accessing an SLA registry.](image)

The line 149 shows how to get the SLA Registry reference from the SLAManager context. This context is available for all SLAMs based on the generic SLAM and which is injected automatically by the framework during initialization of the domain-specific SLAM. Lines 151 to 153 show how to get the services IQuery, IRegister and IAdmin of that SLA Registry.

5.1 Data structures and Classes

In addition to the concepts described in the SLA model, here we introduce the concept of a Dependency:
The Dependency class is defined as an association between two different SLAs. Its properties are

- an ID of the agreement term of the 'depending' SLA
- a list of IDs of agreement terms of the 'antecedent' SLA, on which the respective term of the 'depending' SLA depends.

### 5.1.1 Functional Interface

The SLARegistry is defined by the following interfaces which provide access to the implementation of the <<query>> and <<register>> stereotype interactions:

- **<<register>>**: Stores a SLA in the registry, with pointers to dependent/depending SLAs.
- **<<query>>**: Retrieves a SLA, its status, its status history, and the dependent/depending SLAs.

### 5.1.2 IRegister interface

The diagram illustrates the two candidate invocations of the IRegister interface; the first inserts a SLA document in the registry, possible accompanied by its any dependencies, and the current state of the SLA (typically something to indicate establishment but not provisioning yet). The second updates this document with newer information re: the very content, the dependencies and the state.
5.1.3 **IQuery interface**

This interface:

- Retrieves the document that describes an agreement (i.e. the SLA)
- Retrieves status history for a SLA (possibly only the latest status update)
- and retrieves dependencies for a SLA.

![Diagram of SLA registry operations](image)

**Figure 6:** Registration of a new SLA in the SLA registry

**Figure 7:** Querying an SLA in the SLA registry

The four allowed operations to query the registry may return a complete SLA document that corresponds to a given ID, the dependencies for a SLA, the state history (possibly only the current state depending on the value of the 'Current' flag), and also a list of SLAs that correspond to a specific state. A typical querying interaction is shown in Figure 7.
5.1.4 Detailed Specification

The Dependency class is defined as an association between two different SLAs. Its properties are

- an ID of the agreement term of the `depending' SLA
- a list of IDs of agreement terms of the antecedent' SLA, on which the respective term of the depending’ SLA depends.

Exceptions

- RegistrationFailureException: The attempt to register a SLA failed. Reason should be provided, e.g. lack of storage space, ill-formed SLA, invalid reference to dependencies, invalid initial status, etc.
- UpdateFailureException: As with RegistrationFailureException.
- InvalidUUIDException: A query request passed on an invalid SLA ID.
- InvalidStateException: The argument to getSLAByStatus is erroneous.

5.1.5 IRegister

This interface is used to insert SLAs in the registry, and update them.

- UUID register(agreement: SLA, dep: Dependency[], state: SLAState) throws RegistrationFailureException
  
  Inserts 'SLA' to the registry, with a list of dependencies of its terms on terms of other agreements, and its initial state.

- UUID update(id: UUID, agreement: SLA, dep: Dependency[], state: SLAState) throws UpdateFailureException

  Updates a previously inserted SLA, identified by UUID. If the content (i.e. "agreement") is modified, then a new UUID is returned. If the content remains the same, but either dependencies ("dep") or state ("state") is modified, then the same UUID is returned.

5.1.6 IQuery

This interface is used to retrieve information from the SLA registry

- SLA[] getSLA(id: UUID[]) throws InvalidUUIDException

  Retrieves a list of SLAs, given a list of UUIDs. The ordering in the list should be the same.

- Dependency[] getDependencies(id: UUID) throws InvalidUUIDException

  Given the UUID of a SLA, it returned a complete list of dependencies of its terms, on the terms of other SLAs.

- SLAState[] getStateHistory(id: UUID, current: boolean) throws InvalidUUIDException, InvalidStateException?

  Given a SLA ID, this operation returns the state history if "current" is False. If "current" is True, then only the current state is returned.

- UUID[] getSLAsByState(state: SLAState[], boolean inclusive) throws InvalidStateException?
Returns a list of IDs for SLAs that are currently in state "state". For discarding SLAs in state 'state' inclusive flag should be 'false'.

- **Dependency[] getUpwardDependencies(id: UUID) throws InvalidUUIDException**
  Given the UUID of a SLA, it returned a complete list of dependencies, that depend on the terms of this SLA.

- **SLA[] getMinimumSLAInfo(id: UUID[]) throws InvalidUUIDException**
  Given a list of UUID related to SLAs, it returns a list with minimum information about those SLAs. The 'minimum information' includes: agreedAt, effectiveFrom, effectiveUntil, templateID and the involved parties.

### 5.1.7 IAdmin

This interface is used to manage existing information from the SLA registry. Its main goal is to provide administrative access to this registry.

- **void cleanAll() throws Exception**
  Deletes all existing SLAs in this SLA registry.

- **void clean ( UUID id ) throws Exception**
  Removes from the SLA registry the specified SLA. All dependencies will not be removed.

### 5.2 New Features in Y3

In Y3 the SLA Registry provides following new features:

- Notification via email when a SLA is established
  - Each SLA Registry sends emails to the email account (address customized on the domain-specific SLAM) whenever a SLA is established.

- Support of automatic and semi-automatic negotiations (see 'Protocol Engine' section.
  - By automatic negotiations the SLA registry notifies the domain-specific POC whenever a SLA is established.
  - For semi-automatic negotiations a manual provisioning is required after the SLA is negotiated.

- New service in the IRegister interface called 'sign'. This allows a semi-automatic negotiation. This service updates the SLA status to OBSERVED

- Two new SLA status supported: UNSIGNED and RENEGOTIATED

- Administrative access for cleaning SLA registry
  - A new IAdmin interface is provided

- Enhancement accessing to DB engine
  - Minor enhancements beside the DB access
5.3 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 plugins 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, but instead can focus on the logic design part for each component. In Y3 this management console was synchronized to the latest functionality of the SLARegistry and the embedded dependencies were improved allowing the faster initialization of this console.

A short list of the requirements implemented for this console is described as follows:

1. A GUI (graphical user interface) that provides an overview of the current state of affairs as regards existing SLAs from the SLA registry. Additionally, it can provide information about past SLAs.
2. The dependency among different SLAs should also be defined somewhere and displayed in the console in a useful manner.
3. In the GUI, the user should be able to set the query configuration information, based on whichever query operations can be executed.
4. When the user submits the query request, the results will be displayed in a list where only general information of SLAs is displayed.
5. When the user selects an item on the list, the detailed information of the SLA will be displayed.
6  SLA Translation

6.1  Motivation

In real-world use cases for SLA management systems, an architecture is deployed which consists of multiple tiers. A standard example is a setup of infrastructure as a service for the lowest, followed by layers for platform as a service and software as a service. In general each layer builds on the layers below, as our example will show.

6.1.1  Problem Statement

The infrastructure layer provides physical or virtualized infrastructure resources as a service. These resources usually cover compute, network and storage resources. The platform layer provides a development and run-time environment for software services. This includes specific libraries as well as development tools. The platform layer as such abstracts away the infrastructure into a coherent façade for specifically targeted application scenarios. As such the platform layer depends on the infrastructure supplied by the infrastructure layer, and cannot provide its service without the underlying infrastructure resources. The software layer finally provides software services, which in turn leverage the services offered by the platform layer by implementing concrete software services using the environment as provided by the platform layer. Furthermore the software as a service layer may also rely on other software services, which results in the case of service composition.

In such a 3-tiered service oriented architecture as shown in Figure 8 each layer exposes services which in turn can be backed up by SLAs. An integral part of these SLAs are guarantees regarding non-functional properties for the targeted service. The non-functional properties depend on the layer in which the service resides. For example a metric for the number of guaranteed database queries makes sense only for a database service in the platform layer, but not for virtual-machine service in the infrastructure layer. As such there is a diverse set of non-functional properties, which depends on the layer. Even though some metrics may have the same designator (e.g. availability), their semantics depend heavily on the layer and domain of application.

![Diagram of a 3-tiered service-oriented architecture](image)

**Figure 8:** A common 3-tiered service-oriented architecture
6.2 Dependency Modelling

6.2.1 Dependency Graphs

If one wants to establish guarantees, optimize their own services and assess the delivered quality of service from a dependent service, it is important to have some kind of translation mechanism between non-functional metrics of two layers. Building on our previous work we define a dependency between non-functional properties as follows:

*A non-functional property A is dependent on another non-functional property B if a change in B elicits a change in A.*

This definition does not constrain A and B to reside in different layers. It also keeps the possibility of mutually dependent NFPs. These facts prove to be a challenge when implementing SLA translation. Using the presented definition another challenge arises from the underlying dependency structure: cyclic dependencies in services’ NFPs.

Figure 9 shows an exemplary dependency graph. It represents the dependencies within a service as well as between NFPs in different services. Nodes represent NFPs, which are ordered in layers. Edges represent dependencies, which are not of Boolean nature, but rather contain weights. These weights are derived by functions, which are defined over specific ranges.

![Dependency Graph](image)

*Figure 9: A dependency graph depicting relations among QoS parameters.*

6.3 Assumptions for an Implementation

On the basis of the concept of the mentioned dependency graph, our task is to provide a translation mechanism between non-functional properties for SLA templates. This is required most of all for optimization during the negotiation phase. During negotiation the optimization components need to assess the requirements of fundamental services and whether their availability matches the computed requirements. This in turn influences the optimization since there is a monetary cost associated with each fundamental service. In this context we define a fundamental service as follows:

*A fundamental service is a service which is required for the implementation of another service. As such a service is fundamental if another service has a direct*
Various concerns have been taken into account for realizing an implementation of translation mechanisms:

- Ease of use for a domain expert
- Efficiency during execution
- Implementation effort
- Adaptability to a broad set of use cases
- Stability of the resulting implementation

The translation process of non-functional properties depends heavily on the domain of application. Due to this the translation component is in fact a domain dependent component, which relies on specific knowledge from domain experts. For each new domain a domain expert needs to provide their experience, as well as proven facts into the translation component. Our first requirement states that a domain expert, who likely is not an expert in the SLA@SOI framework implementation, needs to be able to insert their relevant knowledge into the SLA translation component on their own.

The second requirement assumes that the SLA translation component needs to work in real time, and that it is being relied upon during negotiation, which has strict time constraints due to a time-out mechanism. During negotiation, optimization routines need to be run to assess the best resource assignments and for decision making, which due to the problem’s complex nature require a long run time themselves. In such an environment it is imperative not to use up more time for additional tasks such as SLA translation than is necessary. During optimization it is furthermore assumed, that SLA translation will be invoked many times to assess various possible scenarios. For the mentioned reasons it is important to find a solution for SLA translation that runs in a very short time, and which can potentially be invoked and ran in parallel.

A state of the art analysis has shown that there has not been significant prior work on the topic of SLA translation and translation of non-functional properties across layers in service oriented architectures applicable to our framework’s scenario. As such it is assumed that a lot of work needs to be conducted to achieve a satisfactory realization of translation concepts. Although a complex theoretical discussion of the problem is possible, it is important to also realize these concepts in a tractable way. As such methods need to be chosen and developed that lead to a satisfactory implementation in reasonable time.

The diverse nature of the use cases in SLA@SOI in general demands results from work package A5 that can be applied in a broad variety of contexts. This is especially true for the SLA translation component. A way needs to be found to keep the domain-specific implementation by an adoptee to a minimum, by keeping as much of the translation process generic. At the same time the generic part may make constraining assumptions that would rule out the use of the SLA translation component in one or more of the project’s use cases.

The final requirement demands that SLA translation is a stable sub-component, to be used during the negotiation process. This is important since it will be used heavily during the planning-and-optimization components in the higher layers. An unstable component would hinder a successful execution when faced with a high number of negotiations and lead to a bottleneck. This requirement is related both
to the requirements for efficiency during run time and a reduced implementation effort.

## 6.4 Related Work

The state of the art lists only a limited number of techniques which have been applied to achieve SLA translation.

### 6.4.1 Term Rewriting Systems

One potential method for SLA translation is term-rewriting systems as found in theoretical computer science. This is a methodology which is being used to help to prove theorems. Applying term rewriting systems to SLA translation however is considered to be cumbersome since these apply complex mathematical concepts, which have not yet been applied to the present domain. This assumes that some adaptation of term-rewriting system would be necessary to use them for SLA translation. Furthermore a suitable representation and transformation would need to be defined to be able to use term-rewriting in our application scenario.

### 6.4.2 Semantic Technology

Another concept for SLA translation, which has been tackled in SLA@SOI is ontologies. Using OWL or RDF it is possible to annotate SLAs and SLA templates to then apply pre-defined rules to translate SLAs into each other. This concept is especially useful if one moves through an open environment in which not all SLAs and SLA templates are known. Then annotations can be very helpful to identify SLA templates and SLAs which suit a certain request. Furthermore, even previously unknown providers and their SLAs can be taken into account in SLA translation as well as SLA discovery if they follow the mark-up conventions as defined by the domain-specific vocabulary. In our scenario however we assume that provider and customer already know each other and that both rely on the same vocabulary as defined by the SLA model’s extensions. In this way we are taking a similar approach as in the case of semantic technologies. The SLA model’s vocabulary however already provides a common semantic for all words, and so a further annotation with OWL is not required.

## 6.5 A Rule-based Approach

From this review of the state of the art we came to the conclusion to take an approach which resembles the OWL approach. We use the SLA model for providing non-functional properties and a common understanding of those via vocabularies. With the SLA model as foundation, we apply a rule-based framework. The concrete implementation uses Drools as a rule engine. This has several advantages:

- An advanced implementation of the Rete algorithm
- Extensive Tooling
- Stable software with good support
- Already proven in other components in SLA@SOI as well as in numerous industrial applications
This set up, which can be seen in Figure 10, meets the previously mentioned requirements very well. Using the rule-based approach we can provide ease-of-use to the domain expert. Rule engines have been explicitly developed to provide a user-friendly way of incorporating domain knowledge to expert systems. Furthermore Drools is running an advanced version of the Rete algorithm, which has very good run-time performance even if a high number of complex facts are provided for evaluation. This makes it possible to run the SLA translation procedure many times during negotiations. Using Drools furthermore makes it possible to implement even more complex translation concepts in an easy way as rules. This leads to a reduced implementation effort and to a more stable result, since Drools itself is widely adopted and a proven technology.

Our implementation consists mostly of rules, which are laid out in a layered architecture. The bottom-most layer works on a Java-based representation of a SLA or SLA template. This is the same representation that is being used across the SLAMs internally and throughout most of the framework. As such the SLA is strongly domain- and even use case dependent. Typically each use case has its own, specific representation of an SLA. The first layer of SLA translation extracts agreement terms. This also makes the first layer dependent on the actual representation of the SLA and as such on the use case. In the next step these terms are further decomposed in the next rule-based layer into QoS terms. This makes the second layer only dependent on the widely used guarantee terms within the SLA model. The third layer receives QoS terms which now only depend on the common vocabulary that is established among the users of the SLA@SOI framework. This layer contains the rules, which define the dependencies among the various available non-functional properties. Finally in the fourth layer these rules are being applied in a generic fashion to the extracted QoS terms. As such the topmost layer is not dependent on actual domain-specific data and comprises generic rules for SLA translation.
The various layers in Figure 11 reflect different levels of abstraction which have an impact on the adoption of the SLA translation component for a different scenario. Let us assume that we have a working SLA translation component including all use case specific extensions. In this case the following layers will be affected by changes to the scenario:

- Changing the representation of the SLA itself, without changes in content
  - In this case only the bottom-most layer needs to be adapted. Here only the extraction routines for agreement terms need to be modified to the changed representation.

- Changing the structure and content of agreement terms inside the SLA.
  - In this case the second layer will be affected, since new types of agreement terms need to be handled for QoS extraction.

- Introducing new QoS terms to the SLA vocabulary.
  - In this case the top-most layer needs to be extended to incorporate the dependency relationships between the newly added Qos terms and the previously available ones.

When changes occur one should be aware that the addition and application of a new QoS term most likely will also have an effect on the representation of agreement terms and possibly also on the general representation of the SLA. As such, changes in one of the higher levels may also lead to a ripple effect on the lower layers.

The presented architecture has been implemented in a way that allows translation between two defined layers in the service architecture. This means that in our previous example we could translate SLAs from the SaaS layer to the IaaS layer as well as the other way round. This can be achieved by an implementation via rules that are not dependent on the direction of translation. An effect of this is that the translation rules cover general service dependencies which are bi-directional.
6.5.1 Covering complex translation scenarios

**One-to-Many QoS**

The simplest SLA translation scenarios cover the translation of one source SLA to one target SLA in a different layer. In this the dependencies among QoS terms in one layer to QoS terms in another layer always have a one-to-one relationship. Such a case could also easily be covered without a rule engine. One-to-many relations can however also be easily covered with the rule based approach. Examples for this can be seen further below, where a QoS term’s rule execution will affect multiple QoS terms in the target SLA. In the following scenarios we will always assume that a one-to-many relationship between QoS parameters is given.

**One-to-Many SLAs**

In the case of the translation of one SLA to multiple other SLAs the same basic rules can be applied as in the one-to-one case. Since the SLA translation component gets its SLAs from the planning and optimization component, it is assumed that all of the provided SLAs need to be used for translation because re-selection is assumed to be handled by the POC. Domain-specific dependency rules need to describe how the translation should work in the one-to-many case. A QoS term here might be distributed with equal values, the minimal value might be assumed for all of them and so on. There needs to be a distribution rule for each QoS term, which defines the behaviour for the one-to-many case.

**Many-to-One SLAs (One-to-Many Backwards)**

Since the same rules are being used bi-directionally, the one-to-many case becomes a many to one case if the translation is executed in the reverse order. Similar to the distribution rules in the one-to-many case, here aggregation rules need to be defined, so that a correct aggregation of QoS terms from many QoS terms to one QoS term can be computed. In both cases first the target values for each QoS term are being computed, then these are distributed/aggregated to their final result.

**Cyclic dependencies**

As mentioned previously, cyclic dependencies may arise from our definition of a dependency graph. To support realistic application scenarios we are allowing the definition of rules that represent dependencies within one layer. Since one can define mutually dependent connections, it is possible to create cycles. Due to the inherent complexity in detecting cycles in graphs, we do not provide cycle detection in the SLA translation component. We instead assume that the domain expert will resolve dependency cycles already in his or her translation model.

**Conflicting rules**

It may happen that multiple rules have an impact on the value of one QoS term during translation. In this case a conflict arises, which needs to be resolved. The default behaviour of the supplied rules chooses the value that is provided by the dependency rule with the highest priority. It is possible to extend this conflict
resolution method to more complex scenarios such as aggregating the potential value by various methods, such as in the many-to-one translation scenario.

**Translation functions per range**

To be able to provide a flexible approach, even when modelling a complex translation scenario, it is possible to define ranges for all rules. Then a rule will only be fired if the sourceQoS-term falls into this range. In this way it is possible to define the translation behaviour as a composite model for various non-overlapping sub-ranges. Ranges and the applied functions can be implemented in special Java-classes, which are provided as additional domain knowledge to the rule-based mechanism. For best results, the boundaries of ranges should match, such that a consecutive translation is possible. In this context the functional values that arise from the translation functions should also match at the boundaries.

**6.5.2 Performance analysis**

To verify the applicability of the described approach the implementation has been tested for performance with a high number of facts fed into the rule engine. Table 4 contains the run time for the translation process for various amounts of translation rules, which consist of basic arithmetic operations.

<table>
<thead>
<tr>
<th># Rules</th>
<th>1</th>
<th>2</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time [ms]</td>
<td>28.3</td>
<td>37.5</td>
<td>62.35</td>
<td>140.75</td>
<td>248.8</td>
<td>432.35</td>
</tr>
</tbody>
</table>

*Table 4: Performance Measurements of the SLA translation component*

These run time measurements show that even models with a complexity of one hundred translation rules yield in an execution time of only half a second. As a result it can be said, that this approach suffices even to model complex translation scenarios for real-time negotiations.

**6.6 Future Directions**

Based on our presented work we have seen further topics of interest, which we recommend for future research. This especially includes devising a methodology on how to derive translation models in an automatic or semi-automatic way. Similar work has been conducted using available design data for IT architectures. Another approach taken makes use of measured system data for specific workloads. The measured data is then analysed using statistical methods to derive relations among various components in multi-tier architectures. Based on this there are approaches making use of techniques in machine learning like tree augmented Bayesian networks to detect the quality of dependencies among various NFPs. Finally modelling approaches using queuing networks could be employed for SLA translation. Here a related effort has been made by work package A6 and our Y3 work on the SW-POC (Section 8.2).

**6.7 Conclusion**

Conclusively we have devised an approach for SLA translation in Y3, which provides a partially generic framework for SLA translation. The translation process is conducted via a set of translation rules, which can be modified, filled and
extended by domain experts. We have furthermore shown that the approach provides good characteristics with regards to its time efficiency, so that it can be employed in on-line negotiation, planning and optimization scenarios.


7 SLA Negotiation

SLA@SOI framework provides dedicated software machinery in order for the parties to negotiate with each other using automated agents modelled as SLAMs. Each SLAM inherits a set of generic components and the negotiation platform is one of them. As introduced in Y2, the negotiation platform is composed of a generic protocol encoding approach that could be used to model concrete interaction behaviours as well as a processing component called the Protocol Engine. The latter is capable of understanding and executing the protocols in a domain-agnostic rather generic manner. The former however provides hooks for loosely incorporating domain-sensitivity to the negotiation process. For this to happen, a certain encoding convention is specified as also presented in Y2 documentation and the SLA@SOI book [2] that allows the encoding of interaction behaviour as a state machine. Apart from the flexible approach to model various interaction behaviours, the SLA@SOI negotiation platform delivers a readily useable concrete protocol modelling the most common interaction behaviour prevalent in almost all B line use cases. This behaviour can be summarized by abbreviations such as INC (Initiate, Negotiate and Create Agreement) or RNC (Renegotiate, Negotiate and Create Agreement) that represent sequences of operations. Needless to say, negotiations can be cancelled or existing agreements terminated at any time as well.

The behaviour encoded in protocol determines allowed and disallowed operations in a certain state of its state machine, limits the rounds on each state (if applicable), transitioning of states, process timeout and so on and so forth. For messaging, the synchronous request response like style of communication is used as is the requirement of use cases considered under the B line. The messaging is possible either within a single framework instance where multiple SLAMs negotiate with each other or among remote framework instances where distributed SLAMs may be located. In the latter case, the web-service interface of the negotiation platform is used. In addition in Y3, the feasibility of inducting an asynchronous style of communication has been analysed and found to be easy to adopt if needed. This additive feature was studied under the objective of approaching step-wise compatibility with contemporary specifications like WSAG or negotiation scenarios like auctions that are not demonstrated by the use cases of interest to SLA@SOI, but considered important for a broader adoption of our framework.

Based on the feedback received on previous work, the state of the art on negotiations is extended thoroughly to better situate the work done under SLA@SOI negotiations. It also helped to refine the Y3 requirements that stem from collaborative work with the work package A2 on customizing the generic parameters of the protocol in a consensual manner among the possible negotiators. This Y3 feature eloquently relates negotiation to the agent’s rationality based on past negotiation experience with the negotiator and own business policy that is effective at the time. Section 7.2 addresses this feature in more detail. The survey improved existing understanding as well as reaffirming some of the original assumptions and design principles established in Y2. A brief summary of state of the art is presented here with a comparative hindsight.

7.1 Observations from State of the Art
OPELIX (an FP7 European research project) [10] addresses various aspects of business services from offering a service to its discovery and subsequent negotiation making an electronic contract. Negotiations are conducted in a bilateral fashion.

Inspire [18] supports a manual form of bilateral negotiations that allows a human in the loop in order to maintain an administrative control on offers exchanged among the parties. Automated decision guiding logic is kept confidential and not shared.

Aspire [17] builds upon Inspire to add partially automated negotiation support where agents may suggest users upon the choice of operations. The agents also maintain a strategy that helps process the offers and generate any counter offers.

Then we notice marketplace architectures like the e-Agora [7] project in which users employ intelligent agents to interact with the marketplace. The system provides a process model and a set of supported protocols. The process is defined as a series of activities and phases; protocols are defined by means of rules and restrictions on negotiation activities. On similar lines, the project Kasbah [6] allows negotiating parties to negotiate over a marketplace using agents that implement a certain strategy. The negotiation protocol provides a simple take it or leave it styled bilateral negotiations.

AuctionBot [33] allows for conducting auctions among software agents on the basis of particular parameters: e.g., number of participants, discrete number of commodities under bidding along with rules that implement bid assessing functions.

We also observe a set of projects which intertwines the negotiation protocol with the underlying message communication mode to varying degrees. In this regard, ASAPM [8] is a multi-agent system where automated negotiations take place using FIPA’s Iterated Contract Net Protocol (ICNP). Convergence upon agreement is supported by the protocol through multiple rounds of negotiation.

Among broker based frameworks, the BREin [12] project provides for conducting SLA negotiations by building over FIPA Contract Net Protocol. The protocol’s scope encompasses multiple tiers, thereby addressing a possible hierarchy of negotiations.

The CAAT [24] project defines another agent based system that employs automatic negotiations involving two to three parties. The negotiation protocol defines a sequence of interactions that take place by exchanging messages built upon the FIPA Agent Communication Language (ACL). Messages refer to concepts in ontology to convey a certain action to be performed.

SECSE [9] provides a flexible multi-agent marketplace that can be tailored depending on the multiplicity, workflow, protocol and decision model that fit a specific application domain. Negotiation protocol is synergized with decision models that can be plugged into the system in order to process offers and generate counter offers exchanged during negotiations. SECSE supports automated and manual negotiations where mediation may be provided by the marketplace to find solutions satisfying both parties provided that participants trust the mediator with their objectives. The negotiation protocol is defined as a set of rules in the JBoss rule syntax.

Several of these projects and frameworks sufficiently address various aspects required of negotiations under different contexts and requirements. However, to
generalize upon the diverse usage scenarios, it becomes important to be able to provide an approach which flexibly allows to 1) Design custom interaction behaviours and that 2) Depending upon the scenario, allows either synchronous or asynchronous modes of messaging to be used. Therefore, the protocol must be defined and operated at a higher level of abstraction so it does not gets restricted to limitations of the underlying messaging platform and 3) the negotiation protocol must allow some sort of hooks to easily customize negotiations per application domain.

Apart from existing frameworks and projects related to negotiations, currently we also observe standardizing efforts like WSAG [4] – an initiative by the Open Grid Forum (OGF). WSAG is a specification for web service based agreements and includes a description language for defining service agreement templates as well as a negotiation protocol for performing take it or leave it styled bilateral negotiations. WSAG-N [32] broadens the scope to specifying custom interaction behaviours and thus supporting a host of negotiation protocols written as per the given specification.

In [33], negotiation rules are studied under the context of auctions. Three activities are extracted as applicable to all auction protocols. These include handling of requests, computing exchanges and sharing of intermediate information helpful to reach a conclusion. The activities are complemented by a set of standard parameterized rules that impose restrictions, e.g., rules related to bids, computing exchange (counter offer) and the visibility of bids among participants. Although acknowledged, the structuring of these activities and rules to model custom interaction behaviour is left onto the protocol designer. On somewhat similar lines, [19] have developed a negotiation framework that can be used to model a variety of negotiations. They provide a taxonomy of predefined rules and a simple interaction protocol that uses them to realize a certain negotiation mechanism based on an asynchronous mode of communication as specified by FIPA ACL messaging. In addition, an OWL-Lite based ontology language is developed to represent the service template and offers. Both [33] and [19] target price-centric negotiations that try to build upon well engineered rule sets. Although befitting controlled traditional auction settings, these approaches become restrictive when it comes to SLA negotiations taking place in open world service oriented markets.

Analysing the above systems from the architectural perspective, different patterns are observed. 1) Broker based architectures, where a broker component manages one-to-one negotiations on behalf of involved parties. 2) Market-place based architectures, where the parties involved in M-to-N negotiations are managed by an intermediate marketplace. Both of these approaches require negotiation participants to expose their preferences to the negotiation framework. 3) Independent agents that negotiate with each other without mediation. These freely compete or cooperate based on individual rationality.

From the protocol description perspective, we observe rule based approaches where rules capture protocol semantics in an unambiguous manner and regulate the negotiation process. Despite this commonality, each solution differs in its scope, objectives and design approach. Other approaches employ ontologies and schemas to represent message content and semantics, yet others adopt parameter based configuration of the negotiation protocols.

SLA negotiations are usually based on service templates that the service providers make publicly available for negotiations. The template contains a set of properties with price being just one of them. Most of these properties concern the
quality of service (QoS) that the customer and the provider negotiate to agree upon. Each QoS property contains a set or range of values that the customer may choose from. This is a fundamental shift from single attribute price-centric model towards a multi-attribute one. Needless to say, a single template may also be used to conduct multi-unit, multi-commodity negotiations which represent realistic but computationally complex scenarios.

The problem is further complicated by the fact that in service oriented markets, most agents are self-interested and would not like to share information related to their business objectives or utility maximizing functions. This introduces challenges for some of the above mentioned approaches which try to deliver a standard rule to judge improvement in offers received in subsequent negotiation rounds. Among self-interested agents conducting SLA negotiations, complicated correlations among the negotiable properties are kept private. A generic approach for conducting SLA negotiations therefore requires more flexibility and loose coupling between the domain-specific and generic aspects. An attempt to draw this fine line has been made in [11] where a set of generally applicable negotiation parameters have been identified and implemented as an XML language. A meta-negotiation phase preceding negotiations allows the agents to fix the values of these parameters (e.g., timeouts and negotiation rounds) that are later enforced as a concrete protocol.

Firstly, this work highlights the need for a pre-negotiation phase to dynamically configure protocols which allows parties to adjust various degrees of freedom they are willing to afford each other. Secondly, a major design principle established here is the separation of negotiation protocol from the negotiation strategy itself. This helps separate the domain-specific computationally complex tasks (e.g., offer processing and counter-offer generation) from the general aspects of the negotiation protocol.

A walk through the of above state of the art and our observations as listed above have helped in drawing many of the design principles that were adopted for the SLA@SOI negotiation platform. In Y3, we have received encouraging results from the B line use cases which have used the negotiation platform and successfully demonstrated negotiations among localized or remote SLA Managers which engage each other in a bilateral manner forming hierarchies (where necessary) and negotiating as per the strategy used by their individual Planning and Optimization (POC) components. Renegotiations are also supported on similar lines. The need to renegotiate is determined by the strategy used by the Provisioning and Adjustment (PAC) component that intelligently processes monitoring data in liaison with the monitoring components in order to foresee and pre-empt violation of one or more SLA terms. As per its Y3 requirements, the SLA@SOI negotiation platform is extended to allow for customizable negotiations. This is explained in the next section.
### 7.2 Customizable Negotiations

In negotiations, it is of great importance for each party to be able to measure the feasibility of the negotiation before commencing it. This can be determined by considering two important aspects:

1. The past negotiation experience with the negotiator and
2. The current business policy.

These two factors can be used to set bounds on the various degrees of freedom a negotiation affords (see Section 7.1). These degrees may also determine for instance whether a negotiation is feasible at all. Therefore, there is a need for some sort of pre-negotiation mechanism that allows customizing the generic parameters of the negotiation protocol in a mutually consensual manner. Customization mechanism adds a new phase in the negotiation process. Therefore, we introduce a new state in the interaction modelled as a state machine for Y2. The new generalized state machine is seen in Figure 12.

![Generalized State Machine](Figure 12)

**Figure 12:** Generalized State Machine

As can be seen in Figure 12, the initialize state is optionally followed by a customize state. The customize state allows several rounds of customization during or at the end of which the negotiation state may be triggered, provided a consensus is reached to fix values of generalized protocol parameters. The rest of the interaction remains the same where the negotiate state also allows the agreed upon number of rounds to exchange offers while keeping counter offers limited to a certain number. At some stage within the allowed amount of negotiation rounds, the system may transit to a decide state if a request to create an agreement is made. Needless to say, the negotiation may arrive intentionally or unintentionally into an end state which marks the cancellation or termination of the negotiation process.

To understand it better, Figure 13 demonstrates the process from the perspective of the message sequence that takes place between a customer and a provider.
During the customization phase, the following protocol parameters are fixed:

**Credentials**: allow parties to verify each other if such an understanding exists. This could be an individual key under Primary Key Infrastructure (PKI) based certification environments.

**CustomizationRounds**: informs the negotiating partner to try to reach consensus on customizable parameters in these many rounds, starting with 2.

**ProcessTimeout**: determines the life time of the negotiation process. Negotiation is considered invalid after this timeout has occurred.

**NegotiationRounds**: determine the maximum number of allowed rounds for exchanging offers. If it is set to zero, negotiation would not take place.

**MaxCounterOffers**: sets a cap on the number of counter offers allowed in response to a submitted offer.

**OptionalCritiqueOnQoS**: serves as a tip to the POC to optionally annotate critique on QoS terms of a generated counter-offer. Critique may involve keywords like INCREASE, DECREASE, CHANGE, etc. to help convey a message to the negotiator to consider submitting values for which the chances of reaching agreement is higher. In this way agents may guide or pull each other towards their direction of interest.

**IsSealed**: is of interest in multilateral negotiations such as auctions. For example, it would be false for an English auction but true for First-Price-Sealed-Bid or Vickrey auction.
7.3 Profiling for Negotiations

As hinted in Section 7.2, past negotiation history and current business policy are important factors that are considered during the customization phase to set appropriate bounds to the configurable open parameters that govern concrete protocol behaviour. The idea is to profile negotiation experience whenever a party negotiated with the other as well as profiling a minimalistic business and product histories based on past negotiation experience with the party. The former is logged and maintained by the Protocol Engine during negotiation time, while the latter is performed by any domain-specific manager components by using DML operations exposed by the Protocol Engine for adding or modifying business related history. Under the roles entertained by SLA@SOI, the customers then profile the provider behaviour and the providers profile their customers.

Some information of interest that is profiled is outlined below:

- Attempts to negotiate, renegotiate or terminate SLAs are maintained as negotiation history.
- Frequency of point 1 along with outcomes is used to determine Negotiation_Rank.
- The number of current SLAs is maintained under business history.
- SLA_Worth determines how much an SLA is worth to a provider or a customer.
- Service_Satisfaction_Level determines how much a customer is satisfied with the provider. This value is proportional to the violations of SLA.
- Billing_Satisfaction_Level is determined by provider for a customer against each SLA.
- Penalties_Allowed shows the tolerance limit of a party towards the other.
- Penalty_Satisfaction_Level determines how much a party is satisfied with penalty fulfilment of the other party.
- Provider_Rank is established from SLA_Worth, Service_Satisfaction_Level & Penalty_Satisfaction_Level.
- Customer_Rank is established from SLA_Worth, Billing_Satisfaction_Level & Penalty_Satisfaction_Level.

The domain-sensitive rules then come into action by raising the abstraction level, classifying the party that wishes to negotiate and assigning a befitting rank to the party and for the possible negotiation, before negotiation actually takes place. Furthermore, the rules that set limits on the protocol parameters also benefit from these ranks to assign acceptable values to the parameters. It is to be noted here that domain-sensitive rules represent the “current” policy of the party. This classification may change based on multiple business, economic or social factors. Since all negotiation protocol rules are encoded using Drools metadata anchor tags, the Protocol Engine’s IControl interface makes it very convenient to edit these rules in the form of Java policy objects as and when needed.
8 Planning and Optimization

8.1 Infrastructure as a Service

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.

As per the achievement in Y2, 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.

However, the feasibility of advanced reservation in the IaaS scenario was not fully discussed. In Y3, we take this one step further. We introduce an SLA-based advanced reservation methodology by using computational geometry, which is able to verify, record and manage the infrastructure resources efficiently. Based on that model, the service provider can easily verify the quality of service (resource availability and service time span), and thereby the feasibility of satisfying the customer's request can be determined. Furthermore, a flexible alternative solution could be generated as a counter offer to the customer, when the service provider lacks resources. Therefore the model on one hand increases the utilization of the resources and attempts to satisfy as many customers as possible, while on the other hand strengthens the reputation of the service provider via increased probability to honour the established SLAs.

8.1.1 Problem Statement

IaaS providers seek to maximize their profits and achieve business sustainability. Nevertheless, due to the limited resources, sometimes customer's requests cannot be entertained. Therefore, by using advance reservation, IaaS providers are able to reserve the infrastructure resources in advance and customers can invoke the services during a specific time interval in the future. In this case, customers may benefit from cheaper prices rather renting resources in an immediate provisioning mode [34], service providers could also build a good reputation that will lead to longer contracts and repeat customers.

The problem that this work tries to solve is how to efficiently represent the advanced reservations in the IaaS scenario in a way that:

- All reservations are represented in a context. This context varies dynamically when a new reservation is set up and an old reservation expires.
- A decision maker, i.e. SLAM, can decide the feasibility of incoming request, in the form of an SLA, for reservation of resources by taking service availability and reliability into account. If the request is deemed feasible, a new reservation is added into the context.
• A search engine can offer an alternative solution by taking required availability, reliability as well as a flexible time interval into account.
• While settling a reservation down into the context, the fragments of the server are traced and managed.

### 8.1.2 Problem Modelling

The following notations will be adopted in later sections:

- **Si**, where \( i > 0 \): Represents a physical server. It contains a certain number of CPUs, gigabyte(s) of memory and hard disk.
- **Ti**, where \( i > 0 \): A time point in the future on one server. Its unit could be either coarse-grained (e.g., by day) or fine-grained (e.g., by hour).
- **Pi**, where \( i > 0 \): It is a point that represents a time interval on one server which has the consistent infrastructure resource configuration. It is equal to a fragment.
- **Ri**, where \( i > 1 \): A request from a customer. It is a tentative point. If an agreement is finally reached by both parties, a request point will be distributed into one or more points.
- **Basic Unit (BU)**: We assume that S1 has a configuration of 4 Quad-Core Intel Xeon and 24 GB memory and 2400 GB hard disk. The quantification can be realized simply by setting a basic unit with fixed configuration, namely, each BU has 1 core, 1.5 GB memory and 150 GB hard disk. In this case, S1 can provide maximally 16 BUs.

### Computational Geometry Representation

Figure 14 (a1) and Figure 15 (b1), (c1) represent reservation histogram scheduling. Several requests from customers are scheduled on S1, where x-coordinate indicates time \( Ti \) and y-coordinate indicates quantification of S1. Therefore, customers can customize the VM configuration by setting the number of BU according to their preferences.

Our approach uses concepts from computational geometry to represent the time intervals corresponding to periods as points on a plane, as illustrated in Figure 14 (a2) and Figure 15 (b2), (c2). The x-coordinate indicates ending time and the y-coordinate indicates starting time. Since the ending time of a service is greater than the starting time, all points will always be above the diagonal. The whole server is represented as a plane, in which lie points and segments. Each point indicates a period of time over which the server has a consistent and continuous available resource configuration. The configuration information is dynamically updated according with the new requests. The more the points in the plane, the higher is the degree of the server’s partitioning. Therefore, fragmentation means that a server contains a number of time slices throughout its reservation time. The virtual resource configuration in each time slice differs from its neighbouring slices.
In Figure 14 (a1), a customer sends request R1 for 4 BUs as VM configuration in a time span from T1 to T2. It is apparent that S1 is capable of satisfying R1. Meanwhile, S1 is partitioned into three fragments with respective available resource quantities: fragment F1 with 16 BUs, F2 with 12 BUs and F3 with 16 BUs. We can map the reservation histogram in Figure 14 (a1) to a computational geometry representation in Figure 14 (a2). There, R1 is a tentative point when two parties reach an agreement and is finally merged to be P1, P2 and P3.

Then, in a time span from T2 to T3, R2 arrives for 9 BUs as VM configuration and is handled successfully by S1. This leads to an update of fragment F3 with available resources of 7 BUs and a new fragment F4 with 16 BUs. Consequently, in Figure 14 (a2), a new point P4 is created through R2. The similar approach for handling R3 in Figure 15 (b1), (b2) leads to a new point P5.

Figure 14 and Figure 15 illustrate how these simple VM requests can be mapped into a computational geometry context. A point on the plane represents a one-dimensional attribute, namely the number of available BU for one VM. However, the service provider would best allow customers with flexibility, adaptability and choice (Bartlett, 2007). For IaaS, customers are free to customize the VM configuration. Thus, different VM configurations from different requests cannot be simply quantified to a coarse grained composition of BU and such a reservation histogram scheduling becomes inadequate.

Our approach applies not only to the above situation but also to flexible VM configuration, because an accepted request will dynamically impact its related points, inside which the server information at that specific time interval will also be updated respectively. In this case, a point on the plane represents three-dimensional attributes, namely the available cores, memory and hard disk sizes. Therefore, in our experimental part, SLA requests are generated randomly with various VM configurations.
Figure 15: Computational geometry representation of different plans

**Implementation**

In Figure 16, a class diagram is given, where we can see how the context plane is constructed and maintained. 

- \(<\text{Segment}>\) represents the segment between starting point and ending point.
- \(<\text{RequestPoint}>\) represents the detailed information about the VMs of customer; it is a tentative point in the context at the very beginning. 
- \(<\text{ReservationPoint}>\) is a reserved point, which inherits all the features from \(<\text{RequestPoint}>\) and represents a valid service that will be delivered in the future to the customer. Both points contain a \(<\text{VirtualMachineConfig}>\). The logic part of the implementation is \(<\text{ResourceManagementPlane}>\), it could add and delete the point, or segment. When the context mismatches the request from the customer with its own resource pool, it will try to move the point and get the alternative solutions.
Resource Availability

In IaaS, effective availability management can directly impact customer satisfaction and determines the reputation of service providers [5]. It is therefore necessary to ensure that the service provider delivers the right levels of QoS (e.g., service availability) to satisfy its customers in parallel with its own business objectives.

Service availability is the ability of an IT service or component to perform its required function at a stated instant or over a stated period of time [5]. Specifically, it is the probability that the service is up and running, and it could be formulated as below:

\[
    Service\,\,Availability = \frac{T_a}{T_a + T_b} \times 100,
\]

where \( T_a \) is service uptime and \( T_b \) is service downtime.

When a service is consumed by a customer, the service provider has to ensure that the service availability is in accordance with the one defined in the SLA, in order to avoid the potential penalties due to SLA violations. In [5], the scope of service availability management covers planning, implementation, monitoring and adjustment of IT infrastructure availability, where planning of availability happens during the SLA negotiation phase. Before any Service Level Requirement (SLR) is accepted and ultimately the SLR or SLA is agreed between the business and the IT organization, it is essential that the availability requirements of the business are analysed to assess if and how the IT infrastructure can deliver the required levels.

During IaaS SLA negotiation, a user's service availability request should be translated into corresponding service provider's internal technical terms. One
such term is resource availability, which means the percentage of time that infrastructure resources are available through the whole service life cycle. In [21], authors outline that the service availability guaranteed by three large cloud providers (Amazon, Google and Rackspace Cloud) is more than 99.9% in order to obtain good reputation in today's competitive market. Therefore, as a service provider, we propose to provide 99.9% service availability as well. For that we have to ensure that the resource availability of the required service lifecycle is 100% during the negotiation and planning phase of service availability management.

Our method is able to check the resource availability of SLR in advance. In a2 and b2, all the requests can be satisfied with 100% resource availability, whereas in Figure 15 (c1), (c2), when R4 comes for 4 BUs as VM configuration in a time span from T5 to T6, the service provider lacks resources from T2 to T6. For this mismatch between the requirement and capability, there are 5 possible reactions:

- The service provider can simply reject the request.
- For the existing resource capacity, an attractive price can be provided.
- Outsourcing to third parties for extending local resources can take place [13].
- Suspending and resuming other service(s) in some rational manner [16].
- Finding alternative solution(s) considered close to the original requested time interval.

In our work, we focus on the last approach. In computational geometry context, finding an alternative solution is equal to finding a proper location for R4 in Figure 17 (d1) (d2). In (d2), an alternative solution planner is introduced that moves request point R4 on the track of the segment which goes through the centre of R4 and also parallel to the diagonal, because all points on that track always have the identical time interval as R4's. We name such kind of segment as the target segment of the request point. R4 keeps moving on the target segment in two directions one after another until the server is able to provision the service with resource availability of 100%. Then the service provider will send this counter-offer to the customer. The searching scope is controlled within certain time units. The complexity of this linear search methodology is O(2n), which depends on the searching time units in two directions. If the service provider fails to find an alternative solution within the searching constraint, according to our previous work in [20], the service provider can outsource to other service providers to extend its resource capacity temporarily.

**Figure 17**: Moving the request point to find alternative representations
Virtual Fragments of Server

In the previous section, we propose a strategy for modelling and controlling resource availability.

By moving the request point the resource unavailability problem can be resolved. However, this might not be an optimal solution. From a computational geometry representation, the number of points reflects the degree of fragmentation. The more points introduced, the more complicated a plane has to be maintained. Furthermore, the increased fragmentation reduces the potential scheduling opportunities and results in lower utilization [25]. Rarely used points must be eliminated at the very beginning.

We strive to merge the request point with available points while trying to avoid adding new points into the plane except it is proved to be necessary.

At first, the alternative solution planner tries to search the available points on the target segment of the request point within limited scope. If there exist such points and some or all of them can afford the requirements for VMs, then the best one is chosen according to other selection criteria, like the shortest distance, optimal profit and so on. In turn, if there is no such point, meaning that R4 does not fit to any available point, the alternative solution planner has to further find out if there is some point that contains R4 or is contained by R4. We say that P1 = (T1, T2) completely contains P2 = (T3, T4) if and only if T1 < T3 and T2 > T4. If it is true, after the comparison between requested capacity and the capacity of each chosen point, we can simply merge R4 into the chosen point(s) with an attractive price if the total requested duration is slightly longer than the chosen point(s)'s duration or with a preferential activity that the customer can use the VMs longer than expected if the total requested duration is slightly shorter than the chosen point(s)'s duration. Here "slightly" is supposed to be 2 time units in our experimental evaluation and should be specified by the service provider.

On the contrary, if the difference of durations is obvious, then a new fragment will be introduced. Furthermore, for each server with specific hardware configuration, there are upper bounds to its fragmentation. Normally, the higher capability the server has, the higher the fragmentation upper bound could be. Finally, the expired reservations (points) will be removed from the context periodically, thus the degree of the fragmentation can be reduced and controlled.

Experimental Verification

To evaluate the model with regard to its validity, we established an online simulation scenario with specific functions to increase the resource utilization by searching alternative solutions and the number of satisfied requests. Resources under negotiation are VMs with CPU cores, memory and hard disk. There are 4 types of servers and Table 5 illustrates the infrastructure capacity of each kind of server. The overall hardware infrastructure is: 10*S1, 10*S2, 10*S3, 10*S4.

<table>
<thead>
<tr>
<th>Server</th>
<th>CPU Core</th>
<th>Memory</th>
<th>Hard Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>8</td>
<td>16 GB</td>
<td>500 GB</td>
</tr>
<tr>
<td>S2</td>
<td>8</td>
<td>16 GB</td>
<td>500 GB</td>
</tr>
<tr>
<td>S3</td>
<td>16</td>
<td>64 GB</td>
<td>500 GB</td>
</tr>
</tbody>
</table>
Table 5: Server capacities

There were 3 types of testing workload: the workload with 800 requests, with 1200 requests and with 1600 requests. During SLA negotiation customers can request from 1 to 8 CPU cores, 512 MB to 16 GB memory and 20 GB to 400 GB hard disk.

The requests array [800 requests, 1200 requests, 1600 requests] is evaluated in combination with a searching scope array [ +/- 0, +/- 10, +/- 20, +/- 30], therefore there are in total 12 combinations. The result for each combination is a satisfaction rate. In Table 6 we can see that the wider the searching scope available to the alternative solution planner, the more requests that can be satisfied. For 800 requests, the service provider can handle almost all the requests (95.38%) without finding alternative solutions. For 1600 requests, not every incoming request can be satisfied. For 1200 requests, by searching alternative solutions, almost 90% requests can be satisfied, which is quite close to the service provider’s capability.

<table>
<thead>
<tr>
<th>Workload</th>
<th>+/- 0</th>
<th>+/- 10</th>
<th>+/- 20</th>
<th>+/- 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>800 requests</td>
<td>95.38%</td>
<td>99.38%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>1200 requests</td>
<td>70.42%</td>
<td>86.25%</td>
<td>89.58%</td>
<td>93.67%</td>
</tr>
<tr>
<td>1600 requests</td>
<td>55.19%</td>
<td>70.12%</td>
<td>72.06%</td>
<td>72.88%</td>
</tr>
</tbody>
</table>

Table 6: Satisfaction ratio of requests with different shifting steps

Figure 18 is a summary of fragment statistics of all the servers in [1200 requests, +/- 10] without controlling fragmentation (red points) and with controlling fragmentation (green points). By controlling fragmentation, there are in average 36 fragments in each server, which are 10 fragments less than the approach without considering fragmentation. Figure 19 is a summary of resources utilization of all the servers in [1200 requests, +/- 0] and [1200 requests, +/- 20] scenarios. We can see that by searching alternative solutions on one hand, significantly more customers’ requests can be satisfied (19.16% more). On the other hand, the service provider has higher utilization of resources. For example for CPU cores in (a), (79:55% - 70:30%) * (320 cores * 365 days) = 10804 core * day. 10804 means that 29.6 more cores can be rented to customers throughout the whole year.
8.2 Software as a Service

Complex services in the service oriented world are made up of yet further services upon which dependencies are established during negotiation time. The SW-POC is part of a SLAM that is referred to as SW-SLAM. The SW-SLAM represents a SaaS provider and makes up the middle tier of the three tier scenario envisioned by the B2 demonstrator. As such, it is the most connected component with respect to the interactions it engages with or instigates. These include collaborating with the Software Service Manager (SSM) to build an SCM...
(Service Construction Model) based representation of service dependencies as a service builder object. The SSM allows the SW-POC to make tentative reservations during negotiations against the available capacity for the software service being negotiated. If negotiation succeeds, the SSM provides the capability to confirm the booking of a software service which usually results in reserving a license for the customer.

One of the core features of the SW-POC is the ability to process negotiation requests made against the B2 SLA template which offers two software services to customers. The customer of an SaaS provider could be yet another SaaS provider that aggregates upon the acquired service to offer high end business or complex software services. On the other end of the B2 dependency chain, the SaaS provider itself assumes the role of a customer to negotiate across the domain with an IaaS provider, for a specific virtual infrastructure resource.

This demonstrates hierarchical bilateral negotiations as shown in Figure 20.

![Figure 20: Hierarchical negotiation with the SW-SLAM in the middle](image)

As seen in Figure 20, under the SW-SLAM, the SW-POC on one hand is engaged in negotiating with its direct customer and on the other hand, instigates a nested negotiation with an IaaS provider. For the latter, the SW-POC prepares an offer containing desired QoS properties for the virtual resource (more precisely a virtual machine). These properties are discovered through a process of translation which helps:

1) To relate the software service properties to those of the infrastructure, such that the service levels requested by the customer are met when the software service is eventually deployed on the infrastructure resource.

2) After feasible solutions are found for the customer request on technical grounds of i) own capacity and ii) determination of required infrastructure quality levels, the SW-POC attempts to reduce the solution space to pick top few near optimal solutions that would satisfy the SaaS provider’s current preferences associated with the QoS properties offered in its service template.

If no feasible solution is found, counter-offers may be generated that suggest technically realistic values of the QoS properties. Several rounds of negotiation may be required to reach an agreement. If an agreement is reached i.e., an SLA is created, a provisioning plan is created, that is executed by the Provisioning and Adjustment (PAC) component. Under automated negotiation, this provisioning is
instigated when the SLA creation succeeds. The SW-POC also configures the monitoring manager accordingly by passing it the monitoring features that are maintained by the SSM component in its service builder object dedicated to the service instance successfully negotiated.

8.2.1 Translation Problem & Solution Approach

In order to understand the problem of planning and optimizing at the software layer, we need to address the QoS terms that are related to the problem from both sides of the negotiation realm. The SaaS customer is offered two services namely Inventory Service (IS) and Payment Service (PS). A single operation exists for PS while two exist for IS, giving a total of three service operations. Each service operation is decorated with two QoS terms each in the Software SLA template. One of these terms is the invocation rate (ir) and the other is the completion time (ct). The ir represents the number of times the service operation can be invoked per second while the ct represents the completion time of each invocation. The SaaS provider provides a (discrete) range of values for ir and ct of each service operation in the Software SLA template.

The customer chooses a certain value as desired and awaits a response from the provider to the submitted offer. Meanwhile, the SaaS provider determines whether the provided combination of values can be realistically delivered through a certain virtual machine that can be negotiated for through a nested negotiation with the IaaS provider. For this, an Infrastructure SLA template of the IaaS provider is consulted. Three QoS terms are available to choose from in the Infrastructure SLA template. These are the CPU speed (cpus), number of cores (nc) and memory (m). What is needed here is a mechanism that allows the identification or estimation of the performance of the three software service operations, each under its given load (ir) while running on a simulated virtual machine configured per certain fixed values of cpus, nc and m QoS terms. This functionality is provided by the Prediction Server developed under the A6 work package via its Simulation Service (SS). The interaction that is used for this purpose is the <<evaluate>> interaction.

Needless to say, this poses a combinatorial problem of exponential magnitude given 9 variables, each having a range of different size. Considering every single possible discrete value combination among these 9 variables, the total number of combinations reaches into trillions. Therefore, it becomes evident to discretize the continuous range of each variable at various intervals to limit the number of possible combinations and invoke the SS with a single concrete value of QoS terms for both Software and Infrastructure SLA templates. Since the SS is a time consuming service, it might take up to 1 minute to perform a simulation with a reliable depth. If this time is shortened, the reliability of the results becomes questionable. In order to find a fitting combination of Software and corresponding Infrastructure QoS properties, several hundred simulations may be needed. This therefore is done in an offline manner to collect a host of results for a host of possible combinations made a-priori using the above mentioned discretization approach. The results received are stored as a set of rules that can then be efficiently consulted during the time constraint runtime negotiations.
9  SLA Monitoring

The general architecture of the SLA@SOI framework supports the integration of different types of generic or special purpose monitoring engines. These engines may internally realise different monitoring approaches (reasoning mechanisms) but externally support the same common interface. The established interface fixes the form in which the different monitoring engines receive the guarantee terms of the SLAs that need to be monitored and the monitoring results that they report back to the SLA@SOI framework, thus enabling the basic interoperability that the design of the monitoring infrastructure of SLA@SOI wants to achieve. However, due to differences in the languages that the monitoring engines use in order to express operational monitoring specifications, the monitoring of SLAs expressed in the SLA specification language of SLA@SOI requires the translation of these SLAs into operational monitoring specifications, i.e., specifications that can be checked by a low level monitor plugged into the SLA@SOI framework. To address this problem, the architecture of the monitoring infrastructure of SLA@SOI uses wrappers of the monitoring engines, called Reasoning Component Gateways (RCGs), which have the responsibility for translating: (a) the SLAs expressed in the common language of the SLA@SOI framework into the language of the particular engine that they support, and (b) the results produced by the particular engine into the common monitoring schema used by the framework.

In this section, we describe the reasoning component gateway and the monitoring engine that has been used in the SLA@SOI framework for monitoring SLAs at the software service layer. This engine is called EVEREST [31] and the reasoning component gateway is called EVEREST RCG. In the previous SLA@SOI A5 period (as reported in [29]) we reported versions of EVEREST and EVEREST RCG that support the monitoring of a subset of SLA and SLA(T) abstract syntax constructs. In Y3, we have specifically contributed to the SLA@SOI monitoring framework with the following achievements:

- **Extension of SLA Monitoring Capabilities:** Support for the monitoring of an extended set of SLA terms and expressions, as described in the SLA model specification, is now implemented.

- **Testing of the Extended SLA Monitoring Capabilities:** Functional correctness of the implementation of extended SLA monitoring capability has been tested thoroughly.

- **Integration Testing:** Integration of the EVEREST RCG with Event Bus and Monitoring Manager [28] (via a client) has been tested. Also the deployment of EVEREST RCG as an OSGI-enabled service has been tested.

9.1  Overall architecture of the SLA@SOI monitoring infrastructure

This section gives an overview of the monitoring infrastructure of SLA@SOI framework. As shown in Figure 21, the monitoring infrastructure comprises the EVEREST Reasoning Engine and EVEREST RCG.

EVEREST is a general-purpose engine for monitoring behavioural and quality properties of distributed systems based on events captured from them during the operation of these systems at runtime. The properties that can be monitored by EVEREST are expressed in an Event Calculus [27] based language called EC-
Assertion (see Section 9.1.1). EC-Assertion realises a form of Event Calculus in which the properties to be monitored are expressed in terms of monitoring rules that specify patterns of events that should (or shouldn’t) occur within specific periods of time, and may be related to each other and/or the state of the system that is being monitored with temporal or other data dependencies. EVEREST supports the monitoring of different QoS terms with the help of a set of predefined parametric monitoring templates (see Section 9.1.2). These templates are specified using a language that is a combination of EC-Assertion and the Formal Template Language (FTL) [22].

The RCG provides the interface to access the reasoning engine. The RCG consists of two components, namely an Event Translator and EVEREST RCG Translator. EVEREST RCG is responsible for receiving a Monitoring System Configuration (MSC) from the Monitoring Manager [28] and producing operational monitoring specifications for the reasoning engine of EVEREST. The operational monitoring specifications are subsequently sent to this engine as illustrated in Figure 21.

![Figure 21: Organisation of EVEREST and EVEREST RCG](image)

### 9.1.1 Overview of EC-Assertion

EC-Assertion is based on Event Calculus and is accompanied by an XML schema that enables the representation of monitorable properties in a system exchangeable format.

Event Calculus (abbreviated as as “EC” in the following discussion) is a first order temporal logic language that expresses properties of dynamic systems (i.e., systems that can consume and generate events in ways that depend on and can
alter their internal state) in terms of two basic modelling constructs, namely events and fluents. An event in EC is something that occurs at a specific instance of time, has instantaneous duration, and may cause some change in the state of the reality (system) that is being modelled. This state is represented by fluents.

The occurrence of an event in EC is represented by the predicate \textit{Happens}(e, t, \(\mathcal{H}(t_1, t_2)\)). This predicate represents the occurrence of an event e at some time point t that is within the time range \(\mathcal{H}(t_1, t_2)\) and is of instantaneous duration. The boundaries of \(\mathcal{H}(t_1, t_2)\) can be specified by using either time constants or arithmetic expressions over time variables of other predicates in the EC formula that includes the Happens predicate. Events in EC can affect the overall state of a system in two possible ways: they can initiate or terminate a specific state within it. To represent these effects, EC uses two specific predicates, namely the predicate \textit{Initiates}(e,f,t) and the predicate \textit{Terminates}(e, f, t). The predicate \textit{Initiates}(e,f,t) signifies that a fluent f starts to hold after the event e occurs at time t. The predicate \textit{Terminates}(e, f, t) signifies that a fluent f ceases to hold after the event e occurs at time t. EC also uses two additional predicates, namely \textit{Initially}(f) and \textit{HoldsAt}(f, t). The first of these predicates signifies that a fluent f holds at the start of the operation of a system. The second predicate signifies that a fluent f holds at time t.

EC defines a set of axioms that can be used to determine when a fluent holds, based on initiation and termination, events that regard it.

Events in EC-Assertion can be invocations of system operations, responses from such operations, or exchanges of messages between different system components. Furthermore, fluents are defined as relations between objects and represented as terms of the form \(rel(O_1, \ldots, O_n)\), where \(rel\) is the name of a relation which associates the objects \(O_1, \ldots, O_n\).

The properties to be monitored at runtime are specified in EC-Assertion in terms of monitoring specifications that consist of monitoring rules and assumptions. Monitoring rules and assumptions are expressed in terms of the predicates listed above and have the general form \(\text{body} \Rightarrow \text{head}\). The meaning of a monitoring rule is that if its body evaluates to \textit{True}, its head must also evaluate to \textit{True}, whilst the meaning of the assumption is that when its body evaluates to \textit{True}, its head can be deduced from it. A detailed description of EC-Assertion can be found in [31].

### 9.1.2 Parametric templates for basic QoS terms

For each QoS Term which the EVEREST reasoning engine supports, there is a parametric monitoring template which includes a set of assumptions, used for the QoS Term computation and monitoring. The use of such templates is necessary since the SLA model does not provide formal definitions of standard QoS terms in a form that would enable their processing and generation of the corresponding EC-Assertion formulas from basic EC predicates and fluents. To automate the process of template selection and instantiation, the monitoring templates are described using the Formal Template Language (FTL) [22]. FTL is a generic formal language for expressing templates of any target language. FTL is generative, i.e., it describes sentences of some target language (in this case EC-Assertion) and can generate sentences when provided with an instantiation. A brief introduction of FTL is given in [36] and a detailed description of the EVEREST monitoring specifications generation process using EC-Assertion aware FTL templates is provided in [35].

As introduced in Section 9.1, the EVEREST RCG produces operational monitoring specifications for the reasoning engine of EVEREST; the component EVEREST RCG Translator inside Everest RCG is responsible for generating the operational
monitoring specifications. In particular, the EVEREST RCG Translator consists of the following components: the Parser, the ASTTranslator, and the Instantiator. The Parser transforms SLA expressions, which is represented as abstract syntax JAVA objects in MSC, to an intermediate notational form known as Abstract Syntax Trees (AST). The AST Translator processes the parsed ASTs and selects the appropriate FTL templates from the FTL Templates Repository. Finally, the Instantiator generates operational EVEREST monitoring specifications from the selected FTL templates by analysing the monitorable SLA expressions that are assigned to EVEREST. A detail description of parametric monitoring templates and the transformation of SLA guarantee terms into EC-Assertions can be found in [29].

### 9.2 Enhancement of EVEREST and EVEREST RCG

At the end of Y2 of the project, we released versions of EVEREST and EVEREST RCG supporting the monitoring of a subset of SLA and SLA template abstract syntax constructs. More specifically, EVEREST supported the monitoring of four QoS terms, namely Throughput, MTTR, MTTF and Reliability. Also there was limited support to monitor SLA guarantee terms that involve aggregate functions, arithmetic functions and context functions.

In Y3, we have extended EVEREST and EVEREST RCG to support monitoring of an extended set of SLA terms and expressions, as described in the SLA model specification. Table 7 shows the list of all the SLA model elements and QoS terms that are supported by new versions of EVEREST and EVEREST RCG that have been developed in Y3. In this list, the SLA model elements and QoS terms that have been implemented in Y3 are shown in bold and the corresponding remarks are shown in the Remarks column. The remaining SLA model elements and QoS terms which appear in Table 6 are also supported as they had been implemented in the version of EVEREST released at the end of Y2.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SLA Template / Model Elements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Agreement Terms – Local Variable Declaration, Precondition, Guaranteed States</strong></td>
<td>EVEREST now supports the monitoring of the Agreement Terms that contain preconditions (the remaining types of agreement terms were supported by the end of Y2 implementation).</td>
</tr>
<tr>
<td><strong>Interface Declaration</strong></td>
<td>Supported by end of Y2 implementation.</td>
</tr>
<tr>
<td><strong>Variable Declaration – Expression, Customisable</strong></td>
<td>EVEREST now supports the monitoring of the Agreement Terms or Guaranteed States that involve elements (e.g., variable declaration or interface declaration) containing customised variable declarations. The remaining types of variable declarations (i.e., Expression) were supported by the end of Y2 implementation.</td>
</tr>
<tr>
<td><strong>Guaranteed States – Precondition, ConstraintExpr</strong></td>
<td>EVEREST now supports the monitoring of the Guaranteed States that contain preconditions. ConstraintExpr was supported by the end of Y2 implementation.</td>
</tr>
<tr>
<td><strong>ConstraintExpr – CompoundConstraintExpr</strong></td>
<td>ConstraintExpr was supported by the end of Y2 implementation.</td>
</tr>
</tbody>
</table>
**TypeConstraintExpr** implementation.

**DomainExpr** – **SimpleDomainExpr, CompoundDomainExpr, ValueExpr**

EVEREST now supports the monitoring of **Agreement Terms or Guaranteed States** that involve elements containing simple domain expression and compound domain expressions (e.g. variable declaration or interface declaration). **ValueExpr** was supported by the end of Y2 implementation.

**CompoundConstraintExpr** – **Subexpression**

EVEREST now supports the monitoring of **Agreement Terms or Guaranteed States** that involve elements containing compound constraint expressions including lists of **Subexpression** (e.g. variable declaration or interface declaration).

**EventExpr, FuncExpr**

**EventExpr** and **FuncExpr** were supported by the end of Y2 implementation.

**Operators**

**Logical Operators** – and, or, not

All logical operators were supported by the end of Y2 implementation.

**Comparison Operators** – equals, not_equal, less_than, less_than_or_equal, greater_than, greater_than_or_equal

All the listed comparison operators were supported by the end of Y2 implementation.

**Functions**

**Arithmetic functions** – add, subtract, divide, multiply, **modulo, round**

EVEREST now supports the monitoring of **Agreement Terms or Guaranteed States** that involve elements containing the arithmetic functions **modulo** and **round** (e.g. variable declarations). With the implementation of monitoring support for these two arithmetic functions, EVEREST now supports the monitoring of **ALL** the arithmetic functions defined in the default vocabulary of SLA(T).

**Aggregate functions** – sum, mean, min, max, count, **std, median, mode**

EVEREST now supports the monitoring of **Agreement Terms or Guaranteed States** that involve elements containing the aggregate functions **std, median and mode** (e.g. variable declaration). With the implementation of monitoring support for these three aggregate functions, EVEREST supports the monitoring of **ALL** the aggregate functions defined in the default vocabulary of SLA(T), except percentage.

**Time Series** – series

**Context functions** – time_is, day_is, **month_is, year_is**

EVEREST now supports the monitoring of **Agreement Terms or Guaranteed States** that involve elements the context functions **month_is** and **year_is** (e.g. variable declaration) containing. With the implementation of monitoring support for these two context functions, EVEREST now supports the monitoring of **ALL** the context functions defined in the default vocabulary of SLA(T).

**Event Functions** – periodic

**QoS Terms**

**Numeric Measures** –

EVEREST now supports the monitoring of **Agreement**
### Terms or Guaranteed States

Terms or Guaranteed States that involve element (e.g., variable declaration) containing the QoS terms *availability, accessibility, arrival rate, completion time*. The numeric measures *throughput, mttr, mttf* and *reliability* were supported by the implementation of EVEREST released in the end of Y2 of the project. The standard QoS terms not supported by EVEREST are:

- **Qos:data_volume**: The measurement of the size of parameters is not defined in the standard SLA model. The term is not generally measurable if data are encrypted.
- **Qos:accuracy**: The term is not monitorable in its abstract form.
- **Qos:isolation**

The standard conformance QoS terms: these terms are not monitorable in the abstract form that they have in the SLA model.

The standard security QoS terms: these terms are not monitorable in the abstract form that they have in the SLA model.

### Metrics

**Units of Duration** – *s, ms, min, hrs, day, week, month, year*

EVEREST now supports the monitoring of the Agreement Terms or Guaranteed States that involve elements containing the time unit *week* (e.g., variable declaration). The units of duration *ms, min, hrs, day, month* and *year* were supported by the implementation of EVEREST released in the end of Y2 of the project. With the implementation of monitoring support for *week*, EVEREST now supports the monitoring all the time units defined in the default vocabulary of SLA(T) except *tick* and *us*.

| Table 7: Terms supported by SLA monitoring |

### 9.3 Testing and Validation

To verify the functional correctness of the extended monitoring capabilities of EVEREST and EVERST RCG we performed a series of tests. We devised the test cases based upon each of the model elements, SLA terms and expressions supported by EVEREST. Also the SLA specifications for both the B4 and B6 use cases have been taken into account in devising the test cases. Figure 22 partially illustrates the specification, planning and results of these validation tests.
Figure 22: Partial Results of Testing with Use case Elements

9.4 Future Work

Future work should focus on covering further elements of the SLA specification. More specifically support for the following should be provided in the future:

- **Guaranteed Actions**, i.e., a set of actions that one of the parties to the SLA is obligated to perform, are not yet supported by the translation process.

- Support for monitoring of security related terms and an extended set of comparison operators (e.g. member_of, subset_of etc.)
10 Editing of SLAs and SLA Templates

One of the most significant outputs of the work package A5 is our SLA model, coined SLA*. The model on its own provides a way to describe requirements across all use cases and as such has a wide field of application. This wide area of targeted applications of the SLA model leads to an increase in model complexity. During the application of the SLA model as well as the latter evaluation by the use cases it became apparent, that though the model is very good at covering the use cases’ descriptive requirements, it is hard to make use of the model. The main reason for this, is that the SLA model has several layers of abstraction, each of which has its own validation mechanisms.

At first there is a syntactical checking in the first representational layer. In the case of XML-based representations this syntactical check happens against a pre-defined XML-schema. The second layer of verification is provided by the syntax-converter, which during conversion is capable of discovering additional faults in syntax. The next layer is covered by the Java-implementation of the SLA model, which has even more checking such as checks for unique ids within the template. Finally the highest degree of SLA-template validation is achieved by using the template-registry, which also validates semantic requirements and makes sure that the template fits into the context. This encompasses checking for the uniqueness of global IDs which need to be unique in a domain’s context.

This layered architecture has made it hard for adoptees of the SLA model to engineer valid SLA-templates, which in turn are a necessity for further use within the whole framework, since negotiation, planning, optimization, monitoring and further components require valid SLAs and SLA-templates to work correctly.

Concretely users in the past had to first familiarize themselves with the SLA model. This in itself takes some effort, due to the model’s complexity and unique nature, which results in a steep learning curve. From this understanding of the SLA model the adoptee then needs to represent their own requirements as an SLA template as textual document, likely XML-based. These resulting documents are long (e.g. several thousand lines of XML-code) and thus hard to read for a human. In the next step the user needs to use a provided validation-component to run the created document through the syntax-converter and the template-registry’s validation mechanism. Errors which appear during this process are often not easy to understand to someone who is not well versed with the SLA model. The fact that this process does not provide any information on the specific location of the error makes fault-identification even more tedious.

These considerations have led to the development of an integrated component to realize this composite process. This component is called “SLA Editor” and encompasses the following features:

- Creation of SLAs and SLA-templates from scratch
- Editing of SLAs and SLA-templates
- Integrated validation using core SLA*-components
  - SLA model Java-implementation
  - Template-registry
- Hooks for negotiation-time editing of SLAs
- Support for the user-friendly BNF-based representation
- Integration on top of the Eclipse-platform
- A graphical user-interface

The SLA-editor targets both editing and creating SLAs as well as SLA-templates. As such it can be used as a practical extension to other existing tooling for semi-automated negotiations. Due to its integration with the Eclipse-platform it can be used as a component within the SLA@SOI studio framework. Since the SLA-editor is integrated with the template-registry, it provides the highest layer of validation. At the same time it does not rely on the verbose and for humans hard to edit XML-based representation. An XML document can rather be produced as a final output of the SLA editor using the syntax-converter for rendering. The editor uses the easy to read BNF-representation of the SLA model, which has syntax similar to JSON. This representation has been favoured by use case adoptees during Y2 as the best human-readable representation of the SLA model.

Creating an editor from scratch which provides the mentioned features can be a daunting and time-consuming task. Therefore it is important to employ the right tooling. For this purpose various technologies have been examined. Among these parser-generators stand out as promising to provide a big saving in implementation effort. Several parser-generators such as JavaCC, Yacc, Lemon, BISON, ANTLR and X-text have been evaluated with the result, that X-text offers the most advantages to our endeavour. X-text can generate an Eclipse-based editor on top of a provided grammar, described as a domain-specific language. This grammar is already available for the SLA model and needs to be transferred into X-text's own DSL. X-text furthermore provides features such as context-based code-completion and error-analysis based on the provided grammar.

Even when using X-text's provided feature-set, additional features need to be implemented and integrated manually with the generated editor. This is true for the integration within the SLA@SOI studio, which includes providing hooks for semi-automatic negotiations. The negotiation process which includes the SLA editor is described furthermore below in detail. Additionally the template-registry's validation mechanisms need to be integrated, so that its error-information is generated during editing. This error-information will be used in the best-possible manner to hint at open errors in the SLA-template, which cannot be covered by the grammar alone, such as mistakes that regard extensions to the SLA model.

10.1.1 Domain-specific Extensions

SLA*'s extensions comprise a set of words in a domain-specific vocabulary. As an additional feature we target on implementing a mechanism that will allow for editing-time validation of those extensions. A proper integration of extensions into the SLA editor offers the potential to avoid many time-wasting errors with the extensions themselves. Such errors include wrong usage of namespaces and even missing namespaces. Also misspelled or unknown words within a vocabulary's namespace can be detected and hinted upon.

It would have a clearly positive impact on the usability of the SLA editor if the mentioned features for the extension mechanisms could be realized. The extent to which they actually will be realized in the end depends on issues that will arise during implementation. As such the feature-set will be extended as much as the available effort for this task allows.
10.1.2 Creation and Editing Process

The process of creating SLAs and SLA-templates is identical. The user can easily create an empty new document in the editor’s graphical user-interface. While editing the document code-completion mechanisms as known by common IDE’s are available and will hint at options. For this to work, extensions in the form of Java-classes need to be provided at the editor’s compilation time.

Two examples of the SLA-editor during usage can be seen in Figure 23 and Figure 24 depicting a hint on an erroneous line and the provided code-completion features respectively.

Figure 23: The SLA editor showing a syntax-error

Figure 24: The SLA editor showing code-completion

10.1.3 Negotiation process

Editing SLA-templates for SLA management and SLA negotiation has been identified as a key-feature throughout all the use cases. The use cases however use custom implementations within their graphical tools, which are tailored to special domains. The SLA-editor also provides the means to support the negotiation process for semi-automatic or even manual negotiations in a generic fashion. This feature can be used by adoptees for the initial development phase or even as an integral part of a whole negotiation platform.

To support the negotiation-process within the SLA-editor, the editor-component provides hooks in form of an API to be used for negotiation. These hooks can be
used to load templates and will notify other components upon the confirmation of user inputs.

During negotiations the editor furthermore will reside in a special negotiation-state, which permits only partial changes to the provided SLA template. This entails that it is only possible to change values of agreement-terms within the predefined ranges. It is not possible to make changes to the structure of agreement terms or to enter invalid values.
11 Skeleton SLAM & SLAM Wizard

One of the most important features of G-SLAM is its reusability by new SLAMs with minimal effort during implementation. The Skeleton SLAM defines and gathers the basic structure and components, so the development of new SLAMs is faster and simpler. This skeleton not only contains classes and resources but also includes the skeleton (ready to be fulfilled) of domain-specific components (PAC and POC), so the programmer can take care of implementation details of the concrete SLAM. Configuration files and building maven files are also included for the generation of bundles of the new SLAM.

The SK-SLAM consists of four sub-projects. The main project performs basically two tasks: the initialization of the SLAManagerContext via the G-SLAM service and second, the linking of the domain-specific implementation of PAC and POC with the generic components. The two latest components are defined in the sub-projects skslam-pac and skslam-poc respectively. The fourth sub-project called skslam-core is used for the definition of new interfaces and services that will be shared within the SK-SLAM. With this design, the independence across components within the SLAM enables the plugging-in and plugging-off of new or existing entities respectively. The skeleton SLAM’s structure is shown in Figure 25.

```
SKELETON - SLAM
+ skslam-main
  + src
    + java
      + resources
    + tests
    pom.xml
    readme.txt

+ skslam-core
  + src
    + java
      + resources
    + tests
    pom.xml
    readme.txt

+ skslam-pac
  + src
    + java
      + resources
    + tests
    pom.xml
    readme.txt

+ skslam-poc
  + src
    + java
      + resources
    + tests
    pom.xml
    readme.txt
```

**Figure 25:** The structure of the skeleton SLAM

The Skeleton component provides a maven plugin implementation called *maven-slam-plugin*. This plugin enables the generation of a basic domain-specific SLAM based on the Skeleton structure via simple maven commands, as it’s shown in Figure 26.
After running this script the domain-specific SLAM will contain the directories shown in Figure 27.

**Figure 26:** Using the maven-plugin to generate a SLAM

```
script
@echo off
SET PLUGIN=org.slasoi.slam.factory:maven-slam-plugin:0.1-SNAPSHOT
SET PARAM0=generate
SET PARAM1=Dskeleton.directory.generate/repository
SET PARAM2=Dskeleton.dslam.name=b3slam
SET PARAM3=Dskeleton.dslam.namespace=org.slasoi.usecases.b3

mvn %PLUGIN% %PARAM0% %PARAM1% %PARAM2% %PARAM3%
```

**Figure 27:** The resulting B3-SLAM

The b3slam-pac and b3slam-poc modules will contain the implementation of the domain-specific components. Note that the b3slam-main module is responsible for the loading and linking of all components within b3SLAM, which is automatically generated by the maven-slam-plugin.
12 Performance Evaluation

Many of the SLA@SOI framework components have unit tests and other bundles which allow testing their functionality off-line, outside of OSGi platform. Additionally the framework exposes a generic scenario called 'integration scenario' for testing a hierarchy of SLAMs (business, software and infrastructure SLAM) and thus the negotiation of several SLAs in three layers. Nevertheless those test scenarios are not enough to test the whole framework under heavy demand of client request neither to check the performance and scalability of all components working together within the SLA@SOI framework is not checked under these conditions. For that reason simulations have been selected for testing and evaluating the whole framework performance.

In Y3 a simulator prototype in charge of building a simulated infrastructure on which the SLA@SOI framework can be subjected to thousands of client requests. Based on the Cloudsim toolkit, the simulator is able to reserve and to provision virtual machines demanded by an IS-SLAM. In the client side, the simulator provides also a GUI where the user can define some simulation parameters, such as number of clients making request, frequency of those request and others. With a random mechanism the client takes a SLA template (provided by the IS-SLAM) and set the supported vm parameters and the negotiation is initialized between the customer and the IS-SLAM.

12.1 Simulated Components

Due to dynamic nature of the framework and given that several components react to events produced by other entities in production time, the simulator had to imitate the behaviour of external components to support the thousands of negotiations during simulation. It’s clear that bundles such as Monitoring tools and Bridges or Proxies with real infrastructure might be not used in simulation mode. For this reason, ISM and Monitoring Manager have been also simulated.

Figure 28: Architecture of the SLAM-simulation
Figure 28 shows a global view of the simulator running in OSGi using the SLA@SOI framework. The i-Customer represents the GUI where will be generate the thousands of custom requests. The i-SLAM box represents the infrastructure-SLAM which interacts in middle of i-Customer and the ISM-agent. The ISM-agent contacts internally to the Cloudsim by making reservations and performing provisioning of VMs.

### 12.2 Simulation Scenario

The Simulation environment considered the scenario shown in Table 8.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure</td>
<td>This is the datacenter or infrastructure which contains 10 hosts.</td>
</tr>
<tr>
<td>Host</td>
<td>Simulated machine with following features:</td>
</tr>
<tr>
<td></td>
<td>64 GB RAM, 32 CPUs each 3.0 Hz, 100TB</td>
</tr>
<tr>
<td>IS-SLAM</td>
<td>Infrastructure SLAM in charge of VM services. This SLAM is based on the SLA@SOI framework</td>
</tr>
<tr>
<td>Customer</td>
<td>User interested in reservation of several VMs</td>
</tr>
<tr>
<td>VM</td>
<td>Offered Virtual Machine. For the experiment were considered five types of VMs: DEVELOPER HIGH PERFORMANCE, DEVELOPER, OFFICE, RESEARCH, DESIGN AND MODELLING</td>
</tr>
<tr>
<td>Simulation Period</td>
<td>The experiment took into account one month of VM service with 3000 customers.</td>
</tr>
<tr>
<td>Request / Template</td>
<td>A customer was able to make randomly requests with maximal five VMs, each one reserved for a time window of two hours.</td>
</tr>
<tr>
<td>Request Frequency</td>
<td>A customer created a request within a month with following probabilistic frequency:</td>
</tr>
</tbody>
</table>

![Graph showing # Requests vs Day]

Table 8: The simulation scenario applied for performance evaluation of SLAMs

### 12.3 Simulation Results

Given the previous simulation scenario, 10 simulations were executed on a computer with 4 Intel processors each 3.0 GHz, 4 GB RAM and 200GB HDD. The
The framework was initialized under Equinox 3.5 and JDK 1.6 for Windows 7 64 bits. A local MySQL database was used to provide the required functionality for the SLA and SLAT registries.

Taking into account the simulation results, around 10,000 negotiations, we concluded that:

- The framework shows stability and scalability
- Despite of the simulated infrastructure, the framework show good response times during negotiation; that is less than 3 seconds per negotiation.
- The performance of some components could be improved and some memory leaks were resolved.

As near future work, we want to evaluate the performance of the framework using multiples SLAMs and using the profiling and multiple level negotiations supported by the protocol-engine.

12.4 Profiling

As part of this performance evaluation the usage of a profiler was introduced. A profiler is a toolkit which allows monitoring and analysis of an application during runtime. Since the size and complexity of the SLA@SOI framework grows, it is required to maintain the performance of each component of the framework. Keeping their performance at the required level becomes progressively even more difficult in such dynamic an environment as OSGi.

Actually there are several solutions which provide flexible and powerful profiling. YourKit is an innovative open source tool for profiling applications in Java. With a minimal configuration the Yourkit artifacts can be adopted into OSGi.

In Figure 29 the initial interface of YourKit can be seen. Profiling locally as well as remotely allows monitoring of threads, cpu use, garbage collection, telemetry, etc.

![Figure 29: The setup for profiling SLAMs using a simulation environment](image)

Once Yourkit tool is connected to the SLA@SOI framework, the user is able to navigate through CPU, Threads, Deadlocks, Memory and other items. As depicted in the following picture, using a filter sentence, the current number of objects (in the slasoi package) and their size can be monitored as shown in Figure 30.
After several negotiations via the SLA@SOI framework, one could verify the number of created objects and how many objects have been released. These kinds of observations will help to find memory leaks, since some operations could be creating objects and holding them in some container (such as a Java Vector) causing huge memory consumption and therefore over time ultimately resulting in out-of-memory issues.

**Figure 30:** Profiling the simulation of the SLAM
13   Conclusions

13.1   Summary

This deliverable presents the results for work package A5 during Y3. It lists the key innovations and provides summaries of their technical backgrounds. The key innovations contain novelties such as the completely overhauled SLA model, customisable negotiations, negotiation profiling, advance reservation schemes for IaaS, genetic algorithms for SaaS composition, increased support for QoS using event calculus in monitoring and editing of SLAs and SLA templates. Overall there has been a clear progress and improvement against the feature set that A5 provided by the end of Y2. By the end of Y3 a set of integrated SLA management techniques has been developed, which targets an audience with a wide variety of architectural backgrounds.

During the development in Y3 we have witnessed that our previous work from Y1 and Y2 has been fruitful. As such it was possible to extend the available feature set to the one delivered at the project’s end.

Considering the entire project’s duration we see confirmation in the approach to first develop a conceptual model including contributions to the ad hoc framework. From there it was possible to refine our concepts and to realise them in software implementations in the two following project years. During Y2 and Y3 the foundations laid in Y1 and early Y2, including the framework’s architecture, have been filled with functionality. In Y3 the available functionality has been extended and made more stable for application in real-world use cases.

13.2   Lessons Learned

13.2.1   It is infeasible to provide truly generic solutions

During Y1 and Y2 it became apparent that it is hard to provide truly generic approaches in SLA management. Especially activities regarding service optimization, provisioning and adjustment require domain-specific knowledge to achieve a high level of satisfiability and accuracy. This fact has led to the development, that parts of the components developed by A5 are in fact domain dependent. The boundaries dividing domain-independent and domain-specific parts need to be chosen wisely. A trade-off needs to be made between the complexity of adopting these components and providing results of high quality.

13.2.2   Provided results need to be easily adoptable

The use case adoptions and related evaluations have taught us in A5 that the adoption needs to be taken into account as prime concern. Often results and processes, which seem straight forward to the creators of A5 components, were non-intuitive to use case adoptees in the beginning. For this reason we are providing the SLA editor and have taken rule-based approaches building on coherent technology (e.g. JBoss Drools) across A5 and in fact the whole framework.
13.2.3 A trade-off between requirements and realization needs to be found

The requirements stated by the use cases in SLA@SOI cover a huge variety of applications. These requirements have to be aggregated and put into a homogenous model, which can be implemented. This process has largely been conducted by A1 and partially by A5, resulting in the architecture model. Still building on this model, individual components still need to have an appropriate scale. As such it is important to find a reasonable trade-off between the coverage provided by the modules in A5 and the diverse requirements of the B-line. Taking this trade-off based approach we have found, that it is possible to provide a significant degree of satisfaction within the posed constraints in effort and time.

13.2.4 Many scientific and engineering approaches need to be integrated for good results

To accommodate the wide variety of posed problems to A5, we have learned that also a wide variety of approaches is required. These approaches span how across areas such a numerical optimization in linear, quadratic and non-linear programming. Furthermore especially in the field of SLAs regarding IaaS mixed integer and integer programming problems arise. Furthermore in some cases the utility to optimize is not fully known in an analytical way, which requires more generic approaches such as evolutionary methods and generic optimization techniques such as conjugate gradients. These techniques in our case are complemented by methods related to semantic technologies (SLA model), event calculus (SLA monitoring), computational geometry (IS-POC), simulation techniques, AI expert systems (adjustment, negotiation) and more. SLA management, which targets multiple domains as such requires a wide variety of expertise to cover all sub-problems and integrate those into one coherent solution.

13.2.5 The choice of integration technology is a prime strategic concern

Finally all researched methods need to be integrated within one technological platform. In our case, we have been using mostly OSGi as an integration technology. OSGi is considered a cutting edge product. During integration, which has been a major topic in A5, besides A1 it became apparent that it is important to quickly acquire a high degree of know how in the integration technology, so that efforts can be focused on the functional implementation of our components. As such the integration technology in general should be chosen according to its potential benefits, but also based on the knowledge available within the team of developers, who need to have a good working understanding of the deployed platform.

13.3 Outlook on Future Work

From the perspective of A5 as a whole a tremendous amount of results has been achieved. Nevertheless technical and scientific questions beyond our current state remain unanswered, due to the limited time frame of the project. A common theme throughout work package A5 is the application of its generic components to new domains. During our development phase we have realised, that such a generic approach is not always possible. As an outlook to the future it remains open to push the boundaries of what can be done in a generic way in the field of
SLA management. Although it will remain necessary to have domain-specific parts it should still be possible to shrink the impact that domain-specific parts have on our SLA management architecture.

Future directions for the individual tasks can be found in the respective sections of this document.
14 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 *. 

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

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

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

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

Agreement Term Agreement terms define the content of a service level agreement.

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

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

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

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

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

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

Infrastructure Manager A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.

Infrastructure Provider A specific kind of service provider that focuses on the provisioning of infrastructure services.
<table>
<thead>
<tr>
<th>Service Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.</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.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service Level Objective</strong></td>
<td>Service Level Objective represents the quality of service aspect of the <em>agreement</em>. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.</td>
</tr>
<tr>
<td><strong>Service Provider</strong></td>
<td>An organization supplying services to one or more internal customers or external customers.</td>
</tr>
<tr>
<td><strong>SLA Manager</strong></td>
<td>A specialization of <em>service provider</em>: person/system that is responsible for managing SLATs and SLA relationships.</td>
</tr>
<tr>
<td><strong>Software Designer</strong></td>
<td>A specialization of <em>software provider</em>: person that designs/develops the architecture and components of a specific SLA based application.</td>
</tr>
<tr>
<td><strong>Software Manager</strong></td>
<td>A specialization of <em>service provider</em>: person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.</td>
</tr>
<tr>
<td><strong>Software Provider</strong></td>
<td>An organization producing <em>software components</em> which might be used by a <em>service provider</em> to assemble actual services.</td>
</tr>
<tr>
<td><strong>Software Service</strong></td>
<td>A software service is a specific <em>IT service</em> which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.</td>
</tr>
<tr>
<td><strong>Software Component</strong></td>
<td>Software components are the entities produced at design-time by a <em>software provider</em>.</td>
</tr>
<tr>
<td><strong>Service Type</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
## Appendix B: Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>BM</td>
<td>Business Manager</td>
</tr>
<tr>
<td>B-SLAM</td>
<td>Business SLA Manager</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>IE</td>
<td>Interaction Event</td>
</tr>
<tr>
<td>FCR</td>
<td>Finite capacity regions</td>
</tr>
<tr>
<td>ISAM</td>
<td>Infrastructure SLA Manager</td>
</tr>
<tr>
<td>ISM</td>
<td>Infrastructure Service Manager</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>LLMS</td>
<td>Low Level Monitoring System</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queuing Networks</td>
</tr>
<tr>
<td>MA</td>
<td>Manageability Agent</td>
</tr>
<tr>
<td>MRE</td>
<td>Monitoring Result Event</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-functional property</td>
</tr>
<tr>
<td>ORC</td>
<td>Open Reference Case</td>
</tr>
<tr>
<td>OVF</td>
<td>Open Virtualization Format</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QPN</td>
<td>Queuing Petri Nets</td>
</tr>
<tr>
<td>PAC</td>
<td>Provisioning and Adjustment Component</td>
</tr>
<tr>
<td>POC</td>
<td>Planning and Optimization Component</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Objects</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SE</td>
<td>Service Evaluation</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLAM</td>
<td>SLA Manager</td>
</tr>
<tr>
<td>SLAT</td>
<td>Service Level Agreement Template</td>
</tr>
<tr>
<td>SM</td>
<td>Service Manager</td>
</tr>
<tr>
<td>SME</td>
<td>Small and Medium-sized Enterprise</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SW-SLAM</td>
<td>Software SLA Manager</td>
</tr>
<tr>
<td>SW-SM</td>
<td>Software Service Manager</td>
</tr>
<tr>
<td>TCO</td>
<td>Total Cost of Ownership</td>
</tr>
<tr>
<td>TOGAF</td>
<td>The Open Group Architecture Framework</td>
</tr>
</tbody>
</table>