

SLA@SOI Tutorial based on Open Reference Case

Whitepaper

Version 1.0; July 25, 2011

Contributors	
Partner	Contributors
SAP	Alexander Wert
TID	Agustin Andres Escamez Chimeno
XLAB	Miha Stopar, Primož Hadalin, Damjan Murn, Marjan Šterk
PMI	Sam Guinea
Intel	Andy Edmonds
FZI	Christoph Rathfelder, Franz Brosch
UDO	Kuan Lu, Miguel Rojas
FBK	Natallia Rasadka

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



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

Document History			
Version	Date	Author	Changes
0.1	12 May 2011	Miha Stopar	Draft
0.2	6 July 2011	Miha Stopar	The core of the document ready. Some contributions still missing.
1.0	25 July	Miha Stopar	Finalization.

Executive Summary

This document provides information on how the SLA@SOI framework was applied for an SLA-driven management of a service-oriented application. The reference service-oriented application is Open Reference Case (ORC) and represents a trading system dealing with the various aspects of handling sales at a supermarket. This includes the interaction at the cash desk with the customer, including product scanning and payment, as well as accounting the sale at the inventory. A potential customer for the ORC is for example a supermarket owner who wants to have several supermarkets connected to a single service provider (where ORC is running), supporting sales of goods by an IT system. Various services such as inventory management, credit card payments, preferred customer club card, accounting etc. are offered by the provider.

SLA@SOI framework enables SLA negotiation between customer and service provider about ORC quality of web services (inventory service, payment service etc.).

Table of Contents

1	Introduction	7
2	Service analysis	7
2.1	Role clarification	7
3	Service clarification	7
3.1	Negotiation overview	9
4	System analysis	11
4.1	Service implementations	12
4.1.1	ORC implementation	12
4.1.2	ORC installation and image preparation	13
4.2	Quality characteristics	15
4.3	SLA Managers	15
4.3.1	BSLAM	15
4.3.2	ISSLAM	19
4.3.3	SLA Registry	22
4.3.4	Syntax Converter	24
4.3.5	Protocol Engine	24
4.4	Service Managers	28
4.4.1	Software Service Manager	28
4.4.2	Infrastructure Service Manager	32
4.4.3	Service Construction Model	35
4.5	Manageability	42
4.6	Predictability	45
4.6.1	Creating Prediction Models	45
4.6.2	Setting up the Prediction Functionality	47
4.6.3	Using Quality Predictions	48
4.7	Monitorability	48
4.7.1	Monitoring Manager	48
4.7.2	SLA monitoring using the ASTRO engine	51
4.7.3	Infrastructure Monitoring	54
4.8	Adjustment	61
4.8.1	BPEL Adjustment	61
4.8.2	Infrastructure Adjustment	63
4.9	Template construction	63
4.9.1	Software SLAT	63
4.9.2	Infrastructure SLAT	68
5	Extending Skeleton SLAM and the SLA@SOI Studio	71
5.1	Skeleton SLAM	71
5.2	SLA@SOI Studio	73
5.3	SLA Manager Development with Studio	73
5.4	Filling-in the Skeleton SLAM for a Specific Use-Case	74
5.4.1	Implementing POC	74
5.4.2	Implementing PAC	75
5.4.3	Workarounds for Common Problems	75
	References	76
	Appendix A: Glossary	77
	Appendix B: Abbreviations	80

Table of Figures

Figure 1: Arrival rate for ORC payment service	8
Figure 2: Price specification action.....	8
Figure 3: Negotiation wizard (list of products).....	9
Figure 4: List of templates.....	10
Figure 5: Create Agreement/Re-Negotiate Window	11
Figure 6: ORC Architecture	12
Figure 7: ORC Deployment Options	13
Figure 8: Test PaymentService in SOAPUI	14
Figure 9 Hierarchical SLA Negotiations among SLAMs	25
Figure 10: SoftwareServiceManager	28
Figure 11 Infrastructure Service Manager.....	33
Figure 12: Service Hierarchy	36
Figure 13: Software Landscape Model	37
Figure 14: Service Implementation Model.....	38
Figure 15: Implementation Artefact	39
Figure 16: Example SCM	40
Figure 17: Select Model	41
Figure 18: The Manageability Interfaces.....	42
Figure 19: PCM Tooling, EMF Model Editors.....	46
Figure 20: PCM Tooling, GMF Model Editors	47
Figure 21: Studio wizard for generation of a new (skeleton) SLAM.....	74

1 Introduction

This guide provides an example on how to apply the overall SLA@SOI framework for a specific application. The purpose of this document is not to give theoretical explanations, but to provide code snippets, command line utilities and sample configuration files.

2 Service analysis

The goal of this phase is to understand the top-level context and requirements for the envisaged service offering.

2.1 Role clarification

Service provider in ORC scenario offers an IT infrastructure, where SLA@SOI framework is running and where ORC application can be provisioned and started by the framework.

3 Service clarification

The following top-level services are offered in the ORC scenario:

- Inventory Service (getProducts and bookSale operations)
- Payment Service (payment operation)

Service level objectives for the ORC operation are:

- completion time (operation must be completed in less than 20ms), which is a service provider obligation
- arrival rate (requests per second), which is a customer obligation

A few examples how the service provider's GUI that offers negotiation and provisioning of the services may look like are given below.

Figure 1 shows arrival rate loaded inside GUI as part of the ORC SLA template. Arrival rate guaranteed state for an ORC payment service is highlighted (100 requests per second), as well as penalty action for this service is highlighted. Note that SLA model supports guaranteed states and guaranteed actions.

Figure 2 shows a price specification action – defining price, billing frequency and other business terms.

Add New SLA

Choose SLA Template

SLA Templates

ID	Model Version	
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View

SLA Template Details

Agreement Terms

- ORC_CustomerConstraint_arrival_rate
- ORC_ResponseTime
- BusinessTerm

Guaranteed States

ID	State	Value
ORCThroughputConstraint	arrival_rate(ORCPaymentService) < "100" bx_per	

Guaranteed Actions

ID	Policy	Precondition	Postcondition
PenaltyAction_arrival_r	mandatory	violated[ORCThroughp	penalty{ price = "1" EUR } }
TerminationAction_arri	mandatory	violated[violation_coun	termination{ name = Terminator terminationClauseId

Previous Finish Cancel

Figure 1: Arrival rate for ORC payment service

Add New SLA

Choose SLA Template

SLA Templates

ID	Model Version	
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View
ORCBUSINESSSLAT1	sla_at_soi_sla_model_v1.0	View

SLA Template Details

Agreement Terms

- ORC_CustomerConstraint_arrival_rate
- ORC_ResponseTime
- BusinessTerm

Guaranteed States

ID	State	Value
----	-------	-------

Guaranteed Actions

ID	Policy	Precondition	Postcondition
PriceSpecificationActio	mandatory	violated[PriceSpecifice	productOfferingPrice{ id = 1 name = Gold validFrom = Thu Jan 01 00:00:00 CET 2009 validUntil = Thu Dec 31 23:59:59 CET 2009 billingFrequency = per_month product{ // Product Description id = 1 name = Product Name } component_product_offering_price{ id = 1 priceType = one_time_charge price = "20" EUR quantity = "1" } component_product_offering_price{ id = 2 priceType = per_month price = "10" EUR quantity = "1" } }

Previous

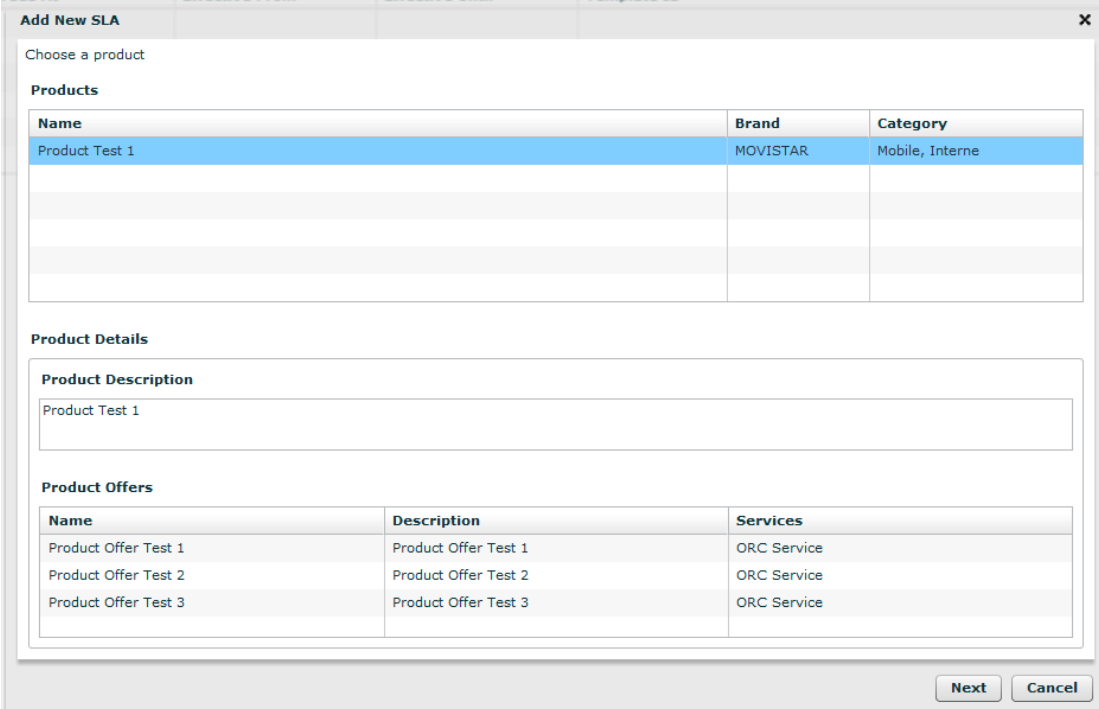
Figure 2: Price specification action

3.1 Negotiation overview

A brief introduction about how service provider's GUI can be written and which SLA@SOI framework packages are needed is given below.

A GUI ([source code](#) [1]) was implemented in order to enable the negotiation for ORC operations between customer and service provider.

User starts a negotiation by clicking the *Create Agreement* button that opens a wizard for a negotiation process (Figure 3).



The screenshot shows a window titled "Add New SLA" with a close button (X) in the top right corner. The window contains a "Choose a product" section with a table of products. Below the table is a "Product Details" section with a "Product Description" text area and a "Product Offers" table. At the bottom right, there are "Next" and "Cancel" buttons.

Name	Brand	Category
Product Test 1	MOVISTAR	Mobile, Interne

Product Details

Product Description

Product Test 1

Name	Description	Services
Product Offer Test 1	Product Offer Test 1	ORC Service
Product Offer Test 2	Product Offer Test 2	ORC Service
Product Offer Test 3	Product Offer Test 3	ORC Service

Figure 3: Negotiation wizard (list of products)

The following code is used inside GUI for obtaining the products:

```
import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub;

import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub.GetProducts;

import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub.GetProductsResponse;

GetProducts getProductsReq = new GetProducts();
getProductsReq.setCustomerID(customerId);

GetProductsResponse getProductsRes =
queryProductCatalogWSClient.getProducts(getProductsReq);
```

The GUI interacts with the framework using web services interface. Web services stub classes are generated from WSDLs exposed by the framework using WSDL2Java tool, which is part of [Axis2](#) package [2]. More about interactions with framework can be found in the documentation on [SourceForge](#) [3].

Framework returns types defined in the SLA@SOI SLA Model, thus org.slasoi.models.slamodel dependency has to be added.

The framework returns a list of products assigned to the customer (Figure 3). The user then selects an SLA template, which displays a list of agreement terms and has an option to edit their values. Finally the user engages a negotiation process by clicking *Finish* button at the bottom of the wizard window.

The following code snippet is used for obtaining SLA templates:

```

import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub;

import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub.GetTemplates;

import
org.slasoi.businessmanager.ws.impl.BusinessManager_QueryProductCatalogStub.GetTemplatesResponse;

GetTemplates getTemplates = new GetTemplates();
getTemplates.setCustomerId(Long.parseLong(customerId));
getTemplates.setProductId(Long.parseLong(productId));
GetTemplatesResponse getTemplatesRes =
queryProductCatalogWSClient.getTemplates(getTemplates);

```

The framework returns a list of templates based on customer's and product's ID (Figure 4). Each product defines offers (services) to the customer, whereas templates define agreement terms of a product between the customer and the provider.

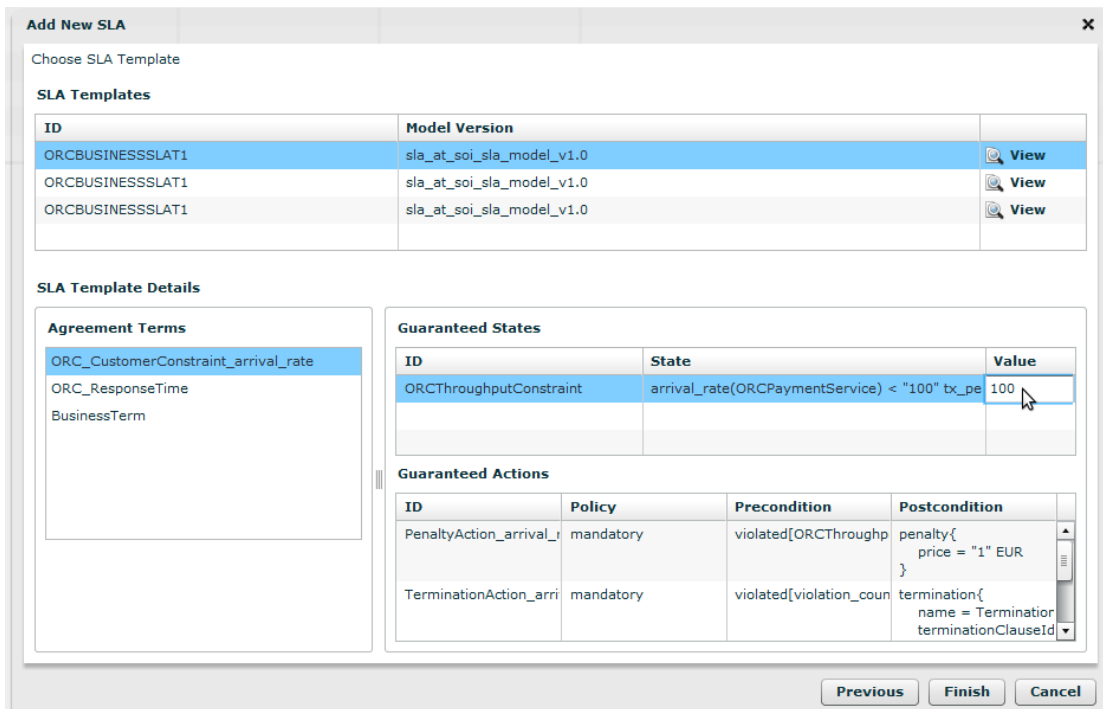


Figure 4: List of templates

The negotiation is then initialized by invoking *initiateNegotiation* which returns a negotiation ID:

```
InitiateNegotiation initiateNegotiationReq = new
InitiateNegotiation();
```

```
initiateNegotiationReq.setSlaTemplate(encode(slaTemplate));
InitiateNegotiationResponse initiateNegotiationRes =
negotiationStub.initiateNegotiation(initiateNegotiationReq);
String initiateNegResp = initiateNegotiationRes.get_return();
```

And finally a negotiation is engaged and a list of agreeable templates is returned:

```
Negotiate negotiateReq = new Negotiate();
negotiateReq.setNegotiationID(initiateNegResp);
negotiateReq.setSlaTemplate(encode(slaTemplate));
String[] altSlatsString =
negotiationStub.negotiate(negotiateReq).get_return();
```

After negotiation a window is opened (Figure 5) which allows the user to either select a template and create an agreement or once again change the values of the template and re-negotiate:

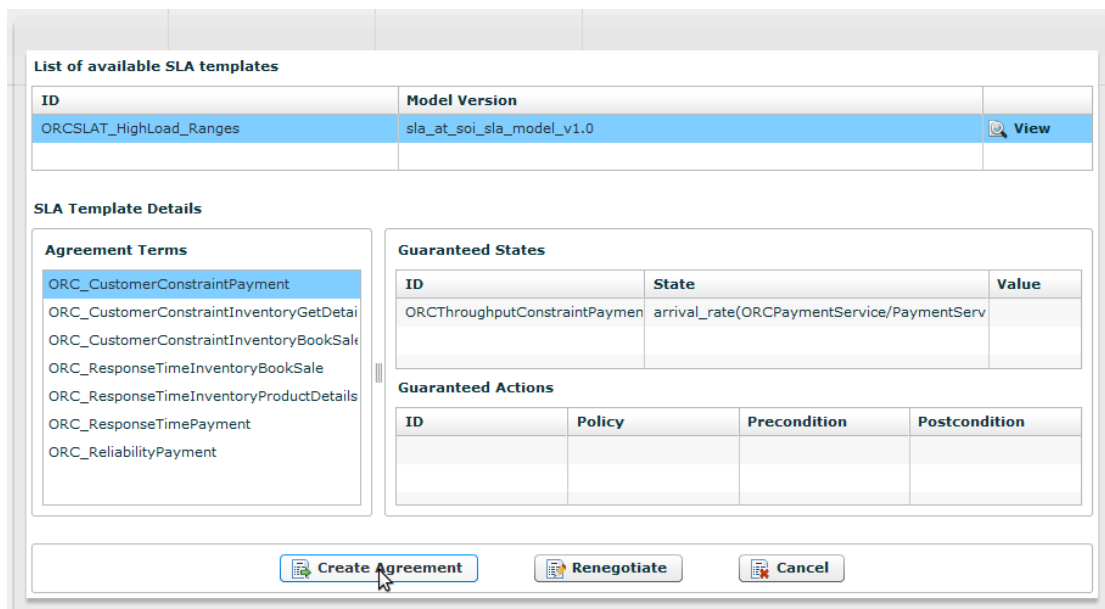


Figure 5: Create Agreement/Re-Negotiate Window

Some additional information about how web services can be exposed when extending SLA@SOI framework skeleton SLAM can be found inside [SLA@SOI Studio documentation](#) [9].

4 System analysis

The goal of this phase is to understand the capabilities and constraints of the underlying system that shall eventually deliver the services defined before.

4.1 Service implementations

4.1.1 ORC implementation

The Open Reference Case (ORC) is a Software as a Service (SaaS) solution supporting the sales process in super markets. In this document it is used as a running example. A more detailed description can be found in [8].

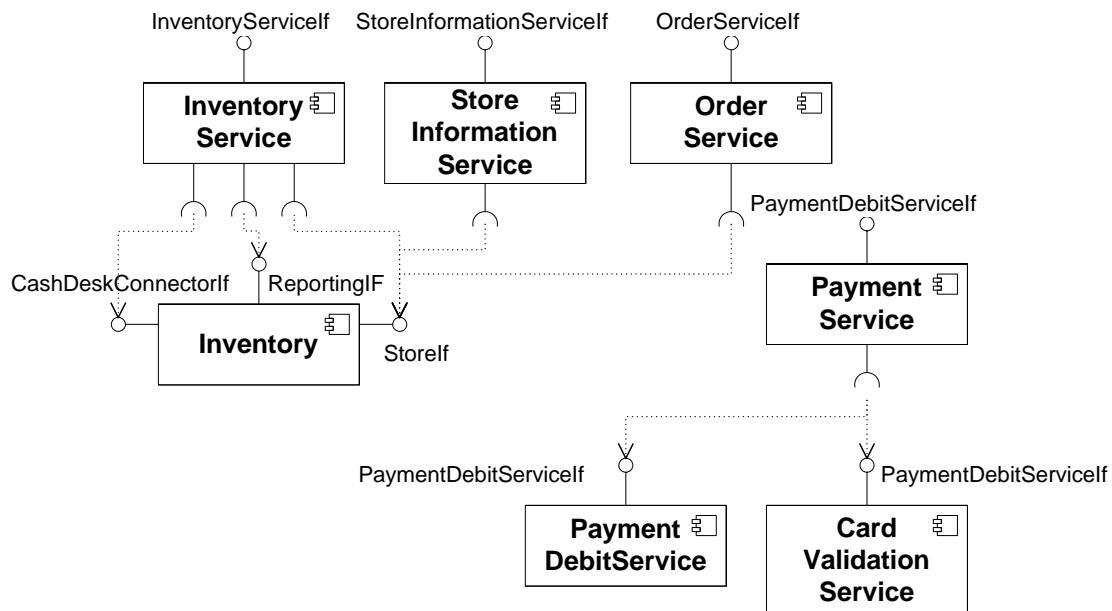


Figure 6: ORC Architecture

As sketched in Figure 6, the ORC consists of several components. The five components **InventoryService**, **StoreInformationService**, **OrderService**, **PaymentDebitService**, and **CardValidationService** implement web service interfaces. **InventoryService**, **StoreInformationService**, and **OrderService** are connected to a legacy system responsible for managing the inventory. This system is represented by the **Inventory** component, which provides three interfaces. The interface **CashDeskConnectorIf** defines a method for getting product information like description and price based on the products bar code. The interfaces **StoreIf** and **ReportingIF** deliver results of database queries. The components **PaymentDebitService** and **CardValidationService** provide functionality for handling credit card payments. External service providers or banks often provide such services, however they can also be implemented locally. These two services are composed within the **PaymentService**, which is a BPEL-based service composition.

The ORC software solution can be deployed on one single machine as well as distributed over several machines. This allows selecting the most appropriate deployment depending on extra-functional requirements like completion time or cost depending on the requirements and the expected amount of customers in the shop. Figure 7 illustrates some of these deployment options. Although, the services are quite independent, there are some constraints limiting the deployment options. The three services **InventoryService**, **StoreInformationService**, and **OrderService** must be deployed on the same machine, as they are service wrappers for the inventory component, which has shared functionality and configurations. The **CardValidationService** and the **PaymentDebitService** are bundled in one deployment unit, as a separation of card validation and debiting makes no sense, as they are always provided by the same institution. In the SLA@SOI framework, we use virtualized machines, which allow

us to deploy new instances by copying existing images and starting them. This dramatically reduces the effort compared to a manual installation on real machines. However, it is possible to install the ORC manually without virtualization.

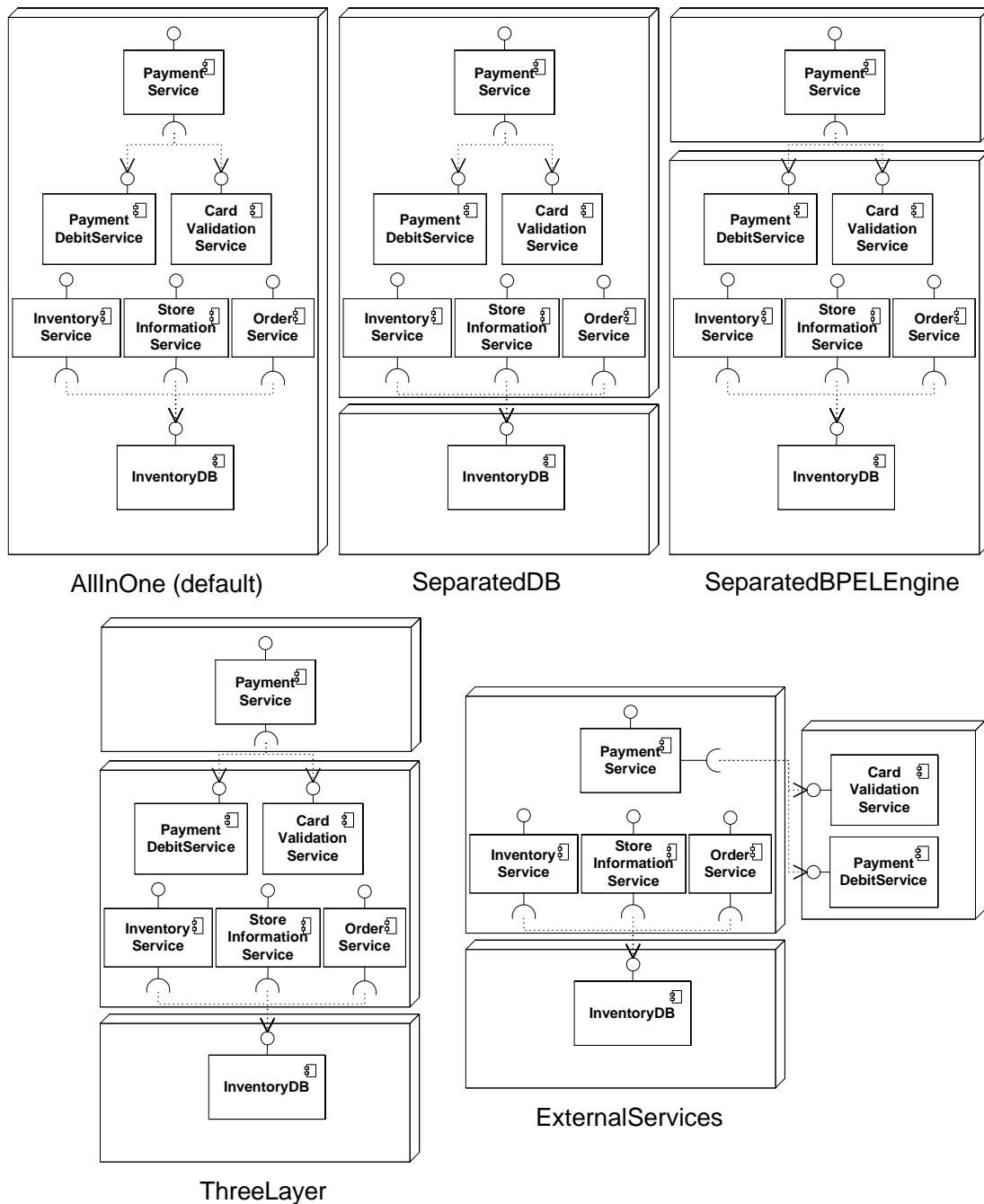


Figure 7: ORC Deployment Options

4.1.2 ORC installation and image preparation

The implementation of the ORC is publicly available on [SourceForge](#) [5]. Detailed installation instructions for Ubuntu based systems can be found in [6].

In order to check if the deployment of ORC services is correct you have to open a web browser. Go to the URL `http://<<IP_of_ORCServices>>:8080/axis` and click on `List`. All services of the ORC should be listed including a link to the

WSDL. To check the deployment of the composite service PaymentService you have to go to the URL `http://<<IP_of_ORCServices>>:8080/BpelAdmin/`. Click on `Deployed Services`, the `PaymentService` should be listed there.

The functionality of the services can be tested with a SOAP client like SOAPUI ¹. In the following the tests are explained for the `PaymentService` and `InventoryService`. For the `PaymentService`, you have to create a new SOAPUI project and enter following wsdl location:

```
http://<<IP_of_ORCServices>>:8080/active-bpel/
services/PaymentService?wsdl
```

As shown in Figure 8 you have to enter following parameter values:

```
cardInformation: testtest
```

```
cardnumber: 7777
```

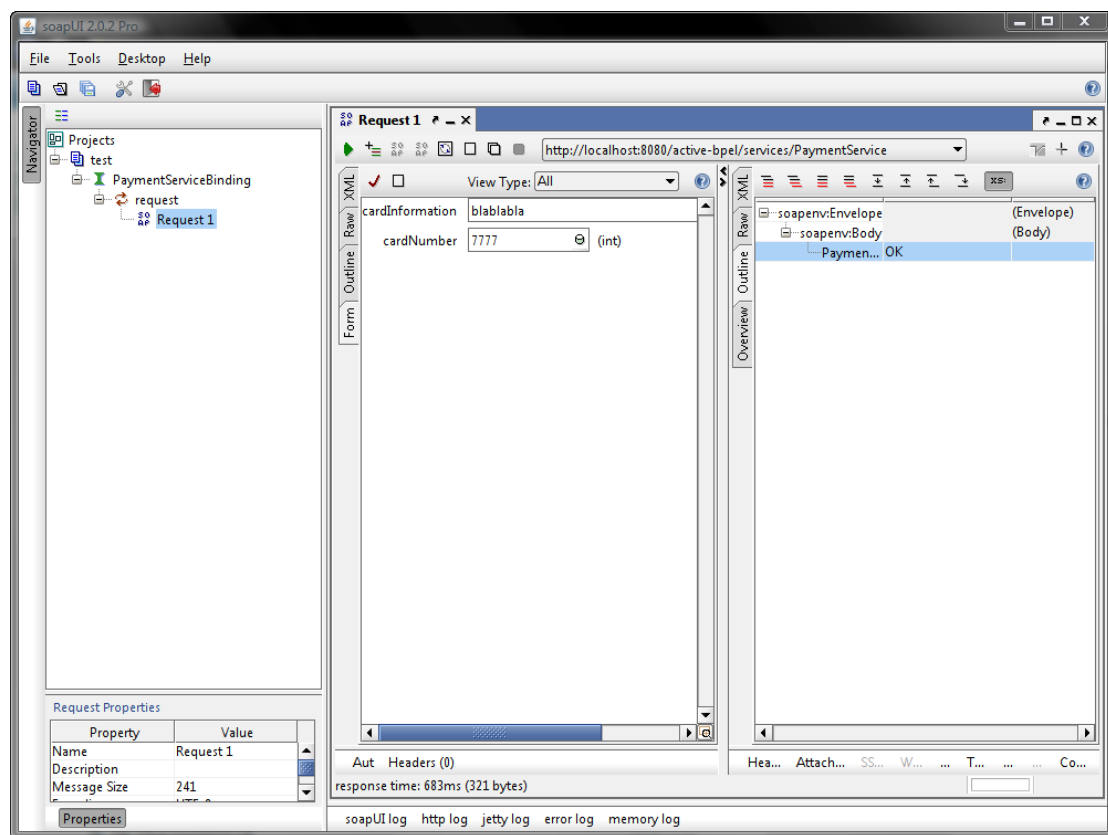


Figure 8: Test PaymentService in SOAPUI

The service should return `OK` or `not enough money`.

For the `InventoryService`, a new SOAPUI Project with following URL to the WSDL needs to be created:

Error! Hyperlink reference not valid. .

Valid parameter values are:

```
StoreID: In0 = 65536
```

```
BarCode: In1 = 794
```

¹ <http://www.soapui.org>

In addition to the services already described in the previous section, the ORC provides two configuration services, which support the simulation of different scenarios in the SLA framework:

- The `BankConfigurationService` can be used to simulate an SLA violation caused by the software layer. The service provides one operation with a boolean parameter, which allows to specify if the banking service should be slowed down or not.
- `ORCConfigurationService` is used to reconfigure the network connections. This reconfiguration is realized by manipulating the hosts file of the system. For this reason the operation provided by the service requires an array of hostname/IP pairs. The implementation of the ORC uses two host names (`ORCBasicServices` and `ORCDatabase`) to communicate with the hosts running the basic services `InventoryService`, `StoreInformationService` and `OrderService` and with the host running the database.

4.2 Quality characteristics

The ORC is a sale system used in several supermarkets, which should reduce the operational costs of a supermarket on the one hand and on the other hand reduce the waiting time of customers at a cash desk. For this reasons the two most important quality characteristics are the costs and the completion time of service invocations. The ORC supports the payment process at the cash desk thus the reliability of the services plays an important role and should be reflected in the SLAs. From the service providers' point of view, it is required to specify and limit the load that is induced by a certain customer on his systems. This knowledge is required to determine the required infrastructure services. For this reason an additional important quality characteristic is the load on the system, which is measured in invocations per second.

4.3 SLA Managers

4.3.1 BSLAM

Parsing BSLAT files

The first thing we need in BSLAM implementation is parsing the BSLAT files correctly, to this end we need to import several packages from different libraries, for instance *slamodel* and *businessterms*:

1. Packages from *slamodel*:

```
import org.slasoi.slamodel.core.EventExpr;
import org.slasoi.slamodel.core.FunctionalExpr;
import org.slasoi.slamodel.core.SimpleDomainExpr;
import org.slasoi.slamodel.core.TypeConstraintExpr;
import org.slasoi.slamodel.primitives.CONST;
import org.slasoi.slamodel.primitives.Expr;
```

```

import org.slasoi.slamodel.primitives.ID;
import org.slasoi.slamodel.primitives.STND;
import org.slasoi.slamodel.primitives.ValueExpr;
import org.slasoi.slamodel.sla.AgreementTerm;
import org.slasoi.slamodel.sla.CustomAction;
import org.slasoi.slamodel.sla.Guaranteed;
import org.slasoi.slamodel.sla.Guaranteed.Action;
import org.slasoi.slamodel.sla.Guaranteed.State;
import org.slasoi.slamodel.sla.InterfaceDeclr;
import org.slasoi.slamodel.sla.Party;
import org.slasoi.slamodel.sla.SLATemplate;
import org.slasoi.slamodel.sla.business.Penalty;
import org.slasoi.slamodel.sla.business.ProductOfferingPrice;
import org.slasoi.slamodel.sla.business.Termination;
import org.slasoi.slamodel.sla.business.TerminationClause;
import org.slasoi.slamodel.vocab.business;
import org.slasoi.slamodel.vocab.core;
import org.slasoi.slamodel.vocab.sla;

```

2. Packages from *businessTerms* are necessary to convert BSLAT to Java objects. JAXB was used to perform serialization and deserialization of business terms java objects:

```

import org.slaatsoi.business.schema.AperiodicScheduleType;
import org.slaatsoi.business.schemaAutomaticScheduleType;
import org.slaatsoi.business.schema.BackupRecoveryMechanism;
import
org.slaatsoi.business.schema.BackupRecoveryMechanism.BackupMechanism;
import
org.slaatsoi.business.schema.BackupRecoveryMechanismType;
import org.slaatsoi.business.schema.DayOfWeekType;
import org.slaatsoi.business.schema.DeliveryMethodType;
import org.slaatsoi.business.schema.DurationType;
import org.slaatsoi.business.schema.EndsAfterOccurrencesType;
import org.slaatsoi.business.schema.EndsNeverType;
import org.slaatsoi.business.schema.EndsOnType;
import org.slaatsoi.business.schema.Monitoring;
import org.slaatsoi.business.schema.Monitoring.Parameter;
import org.slaatsoi.business.schema.ObjectFactory;
import org.slaatsoi.business.schema.OnDemandScheduleType;
import org.slaatsoi.business.schema.PeriodicScheduleType;
import org.slaatsoi.business.schema.PeriodicityType;
import org.slaatsoi.business.schema.ReceiverType;
import org.slaatsoi.business.schema.ReceiversType;
import org.slaatsoi.business.schema.RepeatDailyType;
import org.slaatsoi.business.schema.RepeatHourlyType;
import org.slaatsoi.business.schema.RepeatMinutelyType;

```

```

import org.slaatsoi.business.schema.RepeatMonthlyType;
import org.slaatsoi.business.schema.RepeatSecondlyType;
import org.slaatsoi.business.schema.RepeatWeekendType;
import org.slaatsoi.business.schema.RepeatWeeklyType;
import org.slaatsoi.business.schema.RepeatYearlyType;
import
org.slaatsoi.business.schema.ReportCreationScheduleType;
import
org.slaatsoi.business.schema.ReportDeliveryScheduleType;
import org.slaatsoi.business.schema.ReportFormatType;
import org.slaatsoi.business.schema.ReportType;
import org.slaatsoi.business.schema.Reporting;
import org.slaatsoi.business.schema.ReportingTargetType;
import org.slaatsoi.business.schema.ReportingTargetsType;
import org.slaatsoi.business.schema.Support;
import org.slaatsoi.business.schema.Support.AvailablePeriod;
import org.slaatsoi.business.schema.Support.Procedures;
import
org.slaatsoi.business.schema.Support.Procedures.SeverityLevels;
import
org.slaatsoi.business.schema.Support.Procedures.SeverityLevels.
SeverityLevel;
import org.slaatsoi.business.schema.Support.TimePeriodType;
import org.slaatsoi.business.schema.TimePeriodType;
import org.slaatsoi.business.schema.TimePeriodType.Ends;
import org.slaatsoi.business.schema.TimeUnitKindType;
import org.slaatsoi.business.schema.UpdateProcess;

```

BSLAM interfaces

PAC

BSLAM PAC extends *ProvisioningAndAdjustment* from *org.slasoi.gslam.pac* inheriting two interfaces from it:

- `org.slasoi.gslam.core.pac.ProvisioningAdjustment`
- `org.slasoi.gslam.core.context.SLAMContextAware`

With *SLAMContextAware* we can inject the context of registered services into BPAC to access the running instances of those.

ProvisioningAdjustment in *GSLAM-PAC* already implements *ProvisioningAdjustment* interface from *core* package, However BSLAM-PAC overrides method

```
public void notifyEvent(Event event)
```

to check whether captured event implies sending a report or a termination clause.

BSLAM-PAC implements too the specific interface:

```
public interface IBusinessProvisioningAdjustment {
    String BUSINESS_PAC_ID="BPAC";
}
```

```

    void trackEvent(List<AdjustmentNotificationType>
violationList);
}

```

to handle *Violations* and *Warnings* interception, introduce the affected SLA in Drools Knowledge base and deliver the corresponding event through *sendEvent* method of *ProvisioningAdjustment*:

```

for (AdjustmentNotificationType violation : violationList) {
    // Retrieves SLA from registry

    UUID[] id = new UUID[1];
    id[0] = new UUID(violation.getSlaID());
    SLA[] slas = null;

    try {
        slas = sLManagerContext.getSLARegistry().
            getIQuery().getSLA(id);
    }
    catch (InvalidUUIDException e) {
        ...
    }
    catch (SLAManagerContextException e) {
        ...
    }

    // Insert SLA in knowledgeBase
    for (SLA sla : slas) {
        SharedKnowledgePlane.
            getInstance(BUSINESS_PAC_ID).
                getStatefulSession().insert(sla);
    }
    // Creates event and inserts it into drools knowledge base
    switch(violation.getType()){
    case RECOVERY :{
        sendEvent(new Event(EventType.ViolationRecoveryEvent,
            Identifier.PAC, violation));

        break;
    }
    case NOTIFICATION:{
        sendEvent(new Event(EventType.WarningEvent,
            Identifier.PAC, violation));

        break;
    }
    default:{
        sendEvent(new Event(EventType.ViolationEvent,
            Identifier.PAC, violation));
    }
    }
}

```

```
}  
}
```

4.3.2 ISSLAM

General considerations:

- Understanding the SLA model and SLA manager architecture.
- Infrastructure SLA templates (ISLAT) have to be developed here. Therefore, Java-based implementation will be instantiated here for parsing ISLAT. For parsing SLATs the following classes should be imported:

```
import org.slasoi.slamodel.core.CompoundConstraintExpr;  
import org.slasoi.slamodel.core.CompoundDomainExpr;  
import org.slasoi.slamodel.core.ConstraintExpr;  
import org.slasoi.slamodel.core.DomainExpr;  
import org.slasoi.slamodel.core.FunctionalExpr;  
import org.slasoi.slamodel.core.SimpleDomainExpr;  
import org.slasoi.slamodel.core.TypeConstraintExpr;  
import org.slasoi.slamodel.primitives.BOOL;  
import org.slasoi.slamodel.primitives.CONST;  
import org.slasoi.slamodel.primitives.Expr;  
import org.slasoi.slamodel.primitives.ID;  
import org.slasoi.slamodel.primitives.ValueExpr;  
import org.slasoi.slamodel.service.Interface;  
import org.slasoi.slamodel.service.ResourceType;  
import org.slasoi.slamodel.sla.AgreementTerm;  
import org.slasoi.slamodel.sla.Customisable;  
import org.slasoi.slamodel.sla.Guaranteed;  
import org.slasoi.slamodel.sla.InterfaceDeclr;  
import org.slasoi.slamodel.sla.SLA;  
import org.slasoi.slamodel.sla.SLATemplate;  
import org.slasoi.slamodel.sla.VariableDeclr;  
import org.slasoi.slamodel.sla.business.  
    ComponentProductOfferingPrice;  
import  
org.slasoi.slamodel.sla.business.ProductOfferingPrice;  
import org.slasoi.slamodel.vocab.B4Terms;  
import org.slasoi.slamodel.vocab.common;  
import org.slasoi.slamodel.vocab.core;  
import org.slasoi.slamodel.vocab.resources;  
import org.slasoi.slamodel.vocab.units;
```

- Infrastructure SLA manager (ISSLAM) communicates with Infrastructure Service Manager (ISM). Therefore, Java-based implementation will be instantiated here for the interaction with service manager. For example, a specific class named "ServiceManagerHandler" could be created, inside which the instance of ISM service is included. This instance will be

assigned automatically as part of the context of ISLAM while the OSGi environment is started.

```
public class ServiceManagerHandlerImpl implements
ServiceManagerHandler {
public static ServiceManagerHandlerImpl instance;
private IsmOcciService infraServiceManager;
/**
 * Gets the infrastructure service manager.
 */
public IsmOcciService getInfraServiceManager() {
    return infraServiceManager;
}
/**
 * Sets the infrastructure service manager.
 */
public void setInfraServiceManager(IsmOcciService
infraServiceManager) {
    this.infraServiceManager = infraServiceManager;
}
}
```

The communication between ISLAM and ISM could be divided into negotiation stage and provisioning stage. During negotiation stage, Infrastructure Planning and Optimization Component (IPOC) will query the ISM for the feasibility of provisioning the request from customers, as the code below:

```
/**
 * Queries the resource capability of service manager.
 */
public ProvisionRequestType query(ProvisionRequestType
provisioningRequest) {
    try {

        return
this.infraServiceManager.query(provisioningRequest);
    }
    catch (DescriptorException e) {
        e.printStackTrace();
        return null;
    }
}
```

If enough resources are available, then IPOC can reserve the service for customer through:

```
/**
 * Reserves the resources from infrastructure service
manager.
 */
```

```

public ReservationResponseType reserve(ProvisionRequestType
provisioningRequest) {
    try {

        return
this.infraServiceManager.reserve(provisioningRequest);
    }
    catch (DescriptorException e) {
        e.printStackTrace();
        return null;
    }
}

```

During Provisioning stage, when customer wants to establish the SLA with ISLAM, then IPOC will inform Infrastructure Provisioning and Adjustment Component about provisioning of the service through its <<INotification>> interface as below:

```

class POCINotification implements INotification {
    public void activate(SLA newSLA) {
        assert (newSLA != null) : "it requires an SLA !=
null.";
        try {
            PlanHandler planHandler;
            planHandler = new PlanHandlerImpl(newSLA);
            Plan plan = planHandler.planMaker();
            if (plan != null) {
                ProvisioningAdjustment pac =
context.getProvisioningAdjustment();
                pac.executePlan(plan);
            }
            else {
                LOGGER
                    .error("Infrastructure does not
have enough resources while querying, plan can not be
executed.");
            }
        }
        catch (InvalidSLAFormatException e) {
            e.printStackTrace();
        }
        catch (SLAManagerContextException e) {
            e.printStackTrace();
        }
        catch (PlanFoundException e) {
            e.printStackTrace();
        }
        catch (PlanFormatException e) {

```

```
        e.printStackTrace();
    }
}
}
```

- Based on the Skeleton-SLA manager - inside POC, there are 4 interfaces that have to be implemented in Java: << IAssessmentAndCustomize >>, << INotification >>, << IPlanStatus >> and << IReplan >>, where << IAssessmentAndCustomize >> contains the intelligent part of planning and optimization. The detailed information regarding those implementations can be found on [SLA@SOI Wiki](#) [7].
- The Java implementation of interaction between domain specific SLA managers and monitoring components (Monitoring Manager and Infrastructure Monitoring Agent) should also be considered. In there, ISLAM and ISM cooperate with those two monitoring components for passing MonitoringFeatures set and MonitoringSystemConfiguration to facilitate the monitoring requirement. In "createAgreement" method of << IAssessmentAndCustomize >>, before service provider finally establishes the final agreement, it has to verify the monitorability of the service. Firstly, it has to fetch the "MonitoringFeature" from the IMA through ISM and pass it to Monitoring Manager component. If the response from Monitoring Manager is "MonitoringSystemConfiguration" rather than null, it means the service can be monitored. See the code below:

```
context.getMonitorManager().checkMonitorability(sla,
ServiceManagerHandlerImpl.getInstance().getMonitoringFeature()
);
```

When we get "MonitoringSystemConfiguration", it has to be passed back to the IMA for conducting monitoring activity through the code below:

```
this.infraServiceManager.createComputeConfiguration(
    request.getImage(),
    request.getClientType(),
    request.getLocation(),
    request.getResource().get(Constant.CPU).getAmount(),
    ((CPU) request.getResource().get(Constant.CPU)).getSpeed(),
    request.getResource().get(Constant.Memory).getAmount() * 128,
    Serialization.getInstance().serialize(monitoredConfig),
    request.getVmName(),
    extension);
```

4.3.3 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 SLA Manager components. It exposes also web services interface 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 interactions². The register interaction stores an SLA in the registry, with pointers to dependent/depending SLAs. The second interaction, query, retrieves an SLA, its status, its status history, and the dependent/depending SLAs.

Each SLA Manager contains an instance of the SLA-Registry and can be accessed as follows:

```

}
public void slaRegistryAccess()
{
    try {
        SLARegistry slaRegistry = swContext.getSLARegistry();

        IQuery      iQuery      = slaRegistry.getIQuery() ;
        IRegister   iRegister   = slaRegistry.getIRegister();
        IAdminUtils iAdmin     = slaRegistry.getIAdmin() ;

        iQuery.getSLA( new UUID[]{} ); // this returns SLA references filtered by the specified SLA ids.
        iRegister.register( new SLA(), null, SLAState.OBSERVED ); // this saves a new SLA into the SLA-Registry
                                                                // and initializes in 'OBSERVED' state
        iAdmin.clearAll(); // only for administrative using;
                          // the SLA registry of this SLA manager will be clean.
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected SLAManagerContext      swContext;
protected GenericSLAManagerServices gslamServices;

protected PlanningOptimization  swPOC;
protected ProvisioningAdjustment swPAC;

protected BundleContext         osgiContext;

```

The interfaces to access the SLA registry are located in the package 'org.slasoi.gslam.core.negotiation.SLARegistry'. These interfaces are imported by default in the Skeleton-SLAM and are available by the G-SLAM under OSGi.³

In the current implementation the SLARegistry supports six statuses of the SLAs as follows:

SLA STATE	DESCRIPTION
OBSERVED	The SLA is being observed or monitored.
VIOLATED	The SLA has been violated. i.e. some GuaranteedTerm could not be longer guaranteed

² See details in the SLARegistry SourceForge Repository: <http://sourceforge.net/apps/trac/sla-at-soi/wiki/GenericSlaManager/SLARegistry>

³ Recall that the Generic SLA Manager, also known as G-SLAM provides a generic architecture, which can be used across different domains and use cases for managing the whole SLA life cycle including activities like negotiating SLAs, provisioning resources, monitoring and adjustment. The key feature of the OSGi platform is the high degree of flexibility provided for dynamic behavior, customizable system deployment and reconfiguration of each piece integrates the G-SLAM. The G-SLAM kernel orchestrates the general purpose components which are SLATemplateRegistry, SLARegistry, SyntaxConverter, MonitorManager and the ProtocolEngine.

WARN	The SLA is 'close' to be violated.
EXPIRED	The SLA has expired
RENEGOTIATED	The SLA has been renegotiated
UNSIGNED	The SLA has been created but has to be signed manually.

4.3.4 *Syntax Converter*

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

The syntax-converter is a generic component which does not need to be customized to a specific domain during framework adoption. An extension is however possible for providing conversion features for new over-the-wire serializations.

The syntax-converter takes two kinds of abstract inputs:

The java-based representation of an SLA model. In this case the syntax-converter converts the SLA model from its Java-representation to serialized representation such as XML. There are two serialization mechanisms available by default: the XML-based representation of the SLA model and serialization, which uses WS-Agreement to embed the SLA model's XML-terms.

In the other direction the syntax-converter takes one of the known serializations of the SLA model and converts them into the Java-based version. This version is then delegated to the framework's inner components for further processing and storage.

Due to its functionality the syntax-converter is a central component during negotiation, SLA query and SLA storage procedures. Especially for remote communications with other SLAMs, the syntax-converter is at the core.

For negotiations and SLA processing local to one SLAM or multiple SLAMs to one host, the syntax-converter can be bypassed, making use of direct communication via OSGi-services.

4.3.5 *Protocol Engine*

The Protocol Engine is the negotiation platform of the SLA@SOI framework. It is a generic component and is part of GSLAM. All skeleton SLAMs therefore inherit it. The Protocol Engine works along with Syntax Converter component where the former provides negotiation capabilities and latter provides serialization of the SLA model from XML format to Java and vice versa as well as posting web service based or OSGi based requests to their end point references (EPR). The Protocol Engine is accessible within the context of a SLAM and can be referenced easily

from other components like the POCs and PACs if negotiation or renegotiation needs to be triggered. In addition, manager components or test components can get a reference of its interface, the <<INegotiation>> interface to invoke operations that allow negotiations among two SLAMs in a bidirectional manner. To control this interaction, a Generic Negotiation Protocol is developed which is executed by the Protocol Engine at each negotiating end. The Protocol allows and disallows certain actions that are interpreted by the Protocol Engine as events related to each operation of the INegotiation interface.

In the three-tier scenario that is of most interest to multiple adoption stakeholders, the Protocol Engine provides easy to manage automated negotiations. Negotiations are based on the SLA Template of the provider SLAM. There could be multiple users for one provider that may be negotiating independently in parallel with the provider. The protocol makes a strict separation of the negotiation strategy employed by each provider (inside POC) from the interaction that is defined in the Generic Negotiation Protocol. Figure 9 illustrates the three- tier negotiation hierarchy:

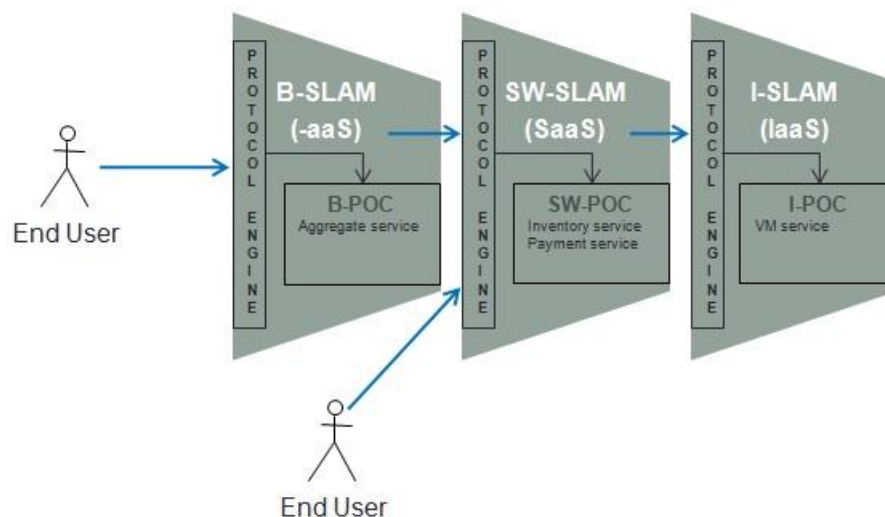


Figure 9 Hierarchical SLA Negotiations among SLAMs

Programmatic Access

In this section, we present an easy approach to write a test bundle or a client program instigates a negotiation at the starting end of this negotiation hierarchy. This client program could be used in the above scenario (cf. Figure 9) where an end user wishes to negotiate a business service provided by the BSLAM. This service is further dependent on one or more software services provided by a SWSLAM which further depends on an ISLAM to deploy its service on a VM of an infrastructure provider (the ISLAM).

To begin, an SLA Template provided by the BSLAM is used by the end user to initiate negotiation. This could be done programmatically by writing a client bundle in OSGi as follows:

```
public void start(BundleContext context) throws
java.lang.Exception {
    this.context = context;
    GenericSLAManagerUtils genericSLAManagerUtils;
    this.tracker = new ServiceTracker(context,
GenericSLAManagerUtils.class.getName(), null);
```

```

        tracker.open();
        try {
            genericSLAManagerUtils = (GenericSLAManagerUtils)
tracker.waitForService(5000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Next, a handle to the Protocol Engine's negotiation interface <<INegotiation>> could be accessed as follows:

```

try {
    if (genericSLAManagerUtils != null) {
        SLAManagerContext[] slaManagerContexts =
genericSLAManagerUtils.getContextSet("GLOBAL");
        ISyntaxConverter sc=
((Hashtable<ISyntaxConverter.SyntaxConverterType,
ISyntaxConverter>) slaManagerContexts[0]
.getSyntaxConverters())
.get(ISyntaxConverter.SyntaxConverterType.SLASOISyntaxConverte
r);
        INegotiation negotiationClient =
sc.getNegotiationClient("http://localhost:8080/services/SWNego
tiation?wsdl");
    }
    catch (GenericSLAManagerUtilsException e) {
        e.printStackTrace();
    }

    catch (SLAManagerContextException e) {
        e.printStackTrace();
    }
}

```

The third and the last step would be to invoke the initiateNegotiation operation, which is usually followed by invoking negotiate operation several times and concluding with the createAgreement operation. This is shown in following code snippet:

```

public void Test(){
    try {
        String negotiationID =
iNegotiation.initiateNegotiation(softwareSLATemplate);
        SLATemplate slaTemplates[] =
iNegotiation.negotiate(negotiationID, softwareSLATemplate);
        SLA sla = iNegotiation.createAgreement(negotiationID,
slaTemplates[0]);
    }
    catch (OperationNotPossibleException e) {
        e.printStackTrace();
    }
}

```

```

    }
    catch (OperationInProgressException e) {
        e.printStackTrace();
    }
    catch (InvalidNegotiationIDException e) {
        e.printStackTrace();
    }
    catch (SLACreationException e) {
        e.printStackTrace();
    }
}

```

Pre-Requisite Settings

The SLA Template used for initiating negotiation should contain the party information annotated accordingly. The Party annotation must specify the role of the party as either Provider or Customer as well as the EPR (*gslam_epr*) to contact them. An example is shown below:

```

<slasoi:Party>
<slasoi:Text/>
<slasoi:Properties>
<slasoi:Entry>
<slasoi:Key>http://www.slaatsoi.org/slamodel#gslam_epr</slasoi:Key>
<slasoi:Value>http://localhost:8080/services/ISNegotiation?wsdl</slasoi:Value> </slasoi:Entry>
</slasoi:Properties> <slasoi:ID>INTEL.IE</slasoi:ID>
<slasoi:Role>http://www.slaatsoi.org/slamodel#provider</slasoi:Role>
</slasoi:Party>

```

The *gslam_epr* is a standard term with following characteristics:

- applies-to: Party
- description: standard annotation key indicating the endpoint reference of a party (in SLA Template).
- value: the actual EPR (a URI) of a SLA-Manager where the party can be contacted for negotiation.
- notes: the service provider and customer must set this annotation before negotiation can be started.

Other than above prerequisite settings, there might be domain specific settings as expected by each SLAM that require certain annotations inside the SLA Template related to that SLAM that must be present in the SLA Template. These domain specific annotations have nothing to do with the functioning of the Protocol Engine rather with domain specific components like SWSLAM's or ISLAM's POC or PAC. In our above scenario, the SWSLAM represents a provider who e.g., annotates a *service_id* into his template that is not of direct interest to the end user but is necessary for the provider to properly process (tentative) allocation issues related to each offer exchanged between the provider and customer. More information on the usage of Protocol Engine, its design, features and adoption

example may be seen at <http://sourceforge.net/apps/trac/sla-at-soi/wiki/ProtocolEngine>

4.4 Service Managers

4.4.1 Software Service Manager

The SoftwareServiceManager encapsulates all management activities specific to software services. In the following, we briefly explain the main subcomponents of the SoftwareServiceManager and their relationships.

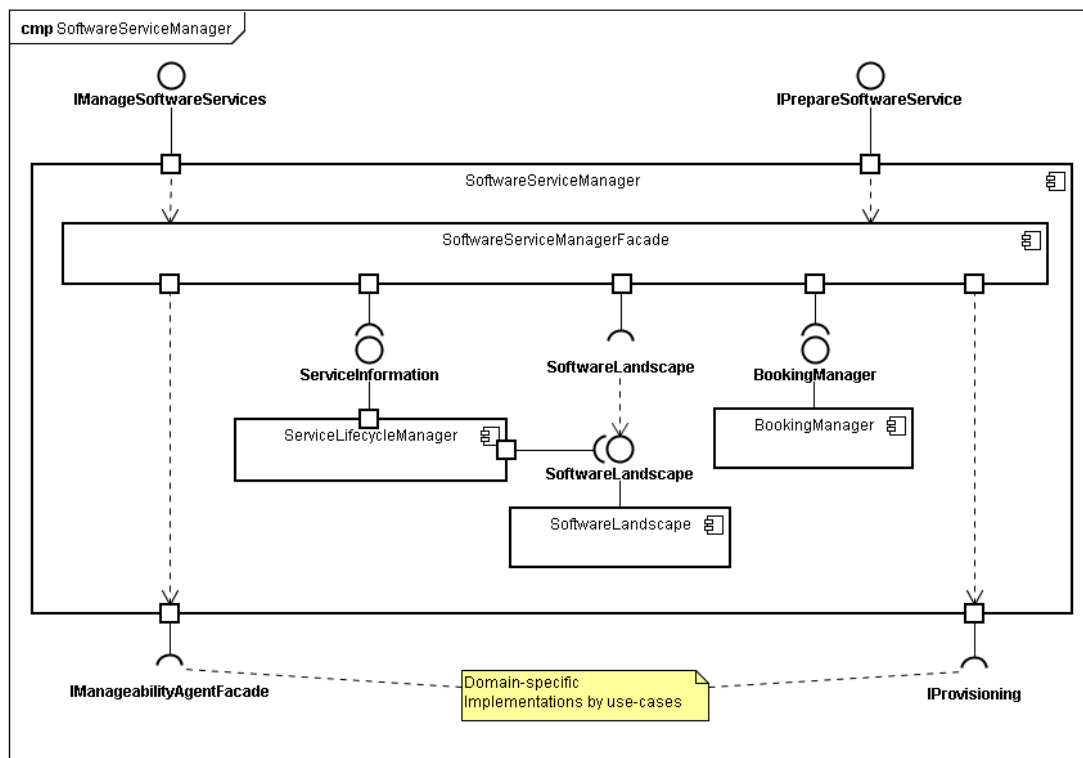


Figure 10: SoftwareServiceManager

Subcomponents

SoftwareServiceManager component is primarily comprised of four subcomponents: SoftwareServiceManagerFacade, SoftwareServiceLifecycleManager, BookingManager and SoftwareLandscape. These subcomponents are described in details in the following paragraphs. These subcomponents collectively realizes the <<prepare_software_services>> and <<manage_software-services>> interactions.

The SoftwareServiceManagerFacade is the primary subcomponent encapsulating management functionality for services throughout their lifecycle. SoftwareServiceManagerFacade facilitates the <<manage_software_services>> and <<prepare_software_service>> interaction which is realized through IManageSoftwareServices and IPrepareSoftwareService interface of the SoftwareServiceManager. It follows Gamma's Facade pattern and hides the internal structure of SoftwareServiceManager from callers. In order to realize the SoftwareServiceManagers's functionality, it makes use of the

ServiceLifecycleManager, the SoftwareLandscape, and the BookingManager as well as the required interfaces IManageabilityAgentFacade and IProvisioning. The ServiceLifecycleManager component serves as a factory for creating new service builders for various service implementations that SoftwareServiceManager can support and as a management facility for status information of all services. Besides a createBuilder operation which initializes a service's life cycle, the ServiceLifecycleManager allows retrieving the service's status information, its instance's metadata, its ManageabilityAgentFacade, and its ServiceBuilder object. The SoftwareLandscape keeps track of available service types, service implementations and of provisioned service instances. The externally accessible interface IPrepareSoftwareServices allows the SoftwareSLAManager to query possible service types as well as service implementations for a particular service type. Additionally, the SoftwareLandscape provides metadata about all service implementations. The metadata includes (but is not limited to) dependencies of the implementation to other services, a description of the service realization, information related to provisioning, and available monitoring / manageability configuration options. However, SLAManagers can only read information about the available service implementations. The implementations and their metadata are maintained by the Service Provider. In addition to information about service implementations, the SoftwareLandscape administrates metadata about service instances of the implementations under its control. The metadata of these instances may contain the service endpoint, a reference to the corresponding manageability agent, and external services used. The BookingManager component is responsible for maintaining and managing the software resource booking and reservation related information. For example BookingManager can maintain information about software licenses required for software service provisioning. BookingManager component realizes the checkCapacity, reserve and book operations of the IPrepareSoftwareServices & IManageSoftwareServices interface.

Required interfaces for domain specific extensions

The SoftwareServiceManager depends on domain specific implementations of the IProvisioning and IManageabilityAgentFacade interfaces summarized in the following sections. The interface IProvisioning encapsulates actions for deploying, configuring and startup & shutdown related functionality of software services. The functionality defined in IProvisioning is predominantly domain and software service specific. Therefore, the SoftwareServiceManager adopts plugin type mechanisms whereby domain specific plugins (implementing the IProvisioning interface) can be deployed so that SoftwareServiceManager can use these plugins to perform software service configuration, startup & shutdown operations. The interface IProvisioning includes the startServiceInstance and stopServiceInstance operations. The IManageabilityAgentFacade interface encapsulates actions for managing and dispatching operation invocation to the domain specific manageability agents. MAManager keeps track of the available manageability agents for the service implementations currently supported by the SoftwareServiceManager. MAManager component provides realization of getManagedObject and execute operations of the IManageSoftwareServices interface.

Implementation

The SLA@SOI Framework provides an implementation of some of the components of SoftwareServiceManager and the rest will be developed by the use cases as these require domain specific knowledge and logic.

Realization of sub-components

- SoftwareLandscape: keeps track of available service implementations and of provisioned service instances.
- BookingManager
- SoftwareServiceLifecycleManager

IProvisioning

Domain-specific interface that is to be implemented by all use cases employing a SoftwareServiceManager. This interface encapsulates the basic functionality for starting and stopping service instances. Furthermore, it allows the SoftwareServiceManager to access (or create) a domain-specific ManageabilityAgents for an existing service instance. Finally, all implementations of this interface encapsulate the knowledge necessary to determine the endpoints for their services. The IProvisioning interface contains the following methods:

startServiceInstance

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

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

- **Parameters:**
 - *builder* - The builder object used to set up the service instance. The builder must be fully specified, i.e., all dependencies to external services have to be resolved.
 - *connectionSettings* - Connection settings for the message-oriented middleware hosting the notification channel.
 - *notificationChannel* - The channel used to notify the responsible parties once the service's provisioning has been completed.
- **Throws:**
 - *ServiceStartupException* - Thrown if an error occurs during the initialization of the provisioning process.
 - *MessagingException* - Thrown if the messaging system cannot be reached.

stopServiceInstance

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

Stops a running service instance.

- **Parameters:**
 - *builder* - Builder object that has been used to create the service instance.

getEndpoints

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

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

- **Parameters:**
 - *builder* - Builder object used to create the service instance
- **Returns:**
 - Endpoint(s) of the service.

getManagibilityAgent

```
IMangeabilityAgentFacade getManagibilityAgent(ServiceBuilder builder)
```

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

- **Parameters:**
 - *builder* - Builder object used to create the service instance.
- **Returns:**
 - ManageabilityAgent of a particular service instance.

IMangeabilityAgentFacade

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

getInstance

```
ServiceInstance getInstance()
```

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

getSensorSubscriptionData

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

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

configureMonitoringSystem

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

- **Parameters:**
 - *configuration* - Configuration to be applied.

deconfigureMonitoring

```
void deconfigureMonitoring()
```

Disables the current configuration.

executeAction

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

Executes domain-specific action for the adjustment of a ServiceInstance.

- **Parameters:**
 - *action* - Domain-specific implementation of the IEffectorAction interface that contains information about what action to execute and its parameters.
- **Returns:**
 - The result of the action in a domain-specific format.

4.4.2 Infrastructure Service Manager

The InfrastructureServiceManager (ISM) is responsible for the creation, lifecycle management, internal optimisation and manipulation of infrastructure resources. These infrastructure resources are broadly grouped into the functional areas of:

- Compute - an entity that performs computation
- Network - an entity that links together 2 or more resources
- Storage - an entity that allows for state to be persisted

The ISM communicates with the actual provisioning system(s) that can provision such resources and is not aware of SLA concerns. The ISM exposes its functionality to clients through an interface and an associated data model and both abstract the low-level details of the provisioning system supported. This interface and model are now standardised, implementing the [Open Cloud Computing Interface \(OCCI\)](#) [10]. Examples of interactions with an OCCI compliant service can be found in the [HTTP rendering specification](#) [11]. The ISM is a composition of a number of components. Further details and how to install the ISM [can be found on the SLA@SOI wiki](#) [12].

The lowest level component of the Infrastructure Service Manager (ISM) is an extended version of Apache Tashi. [Apache Tashi](#) [13] is a system for provisioning

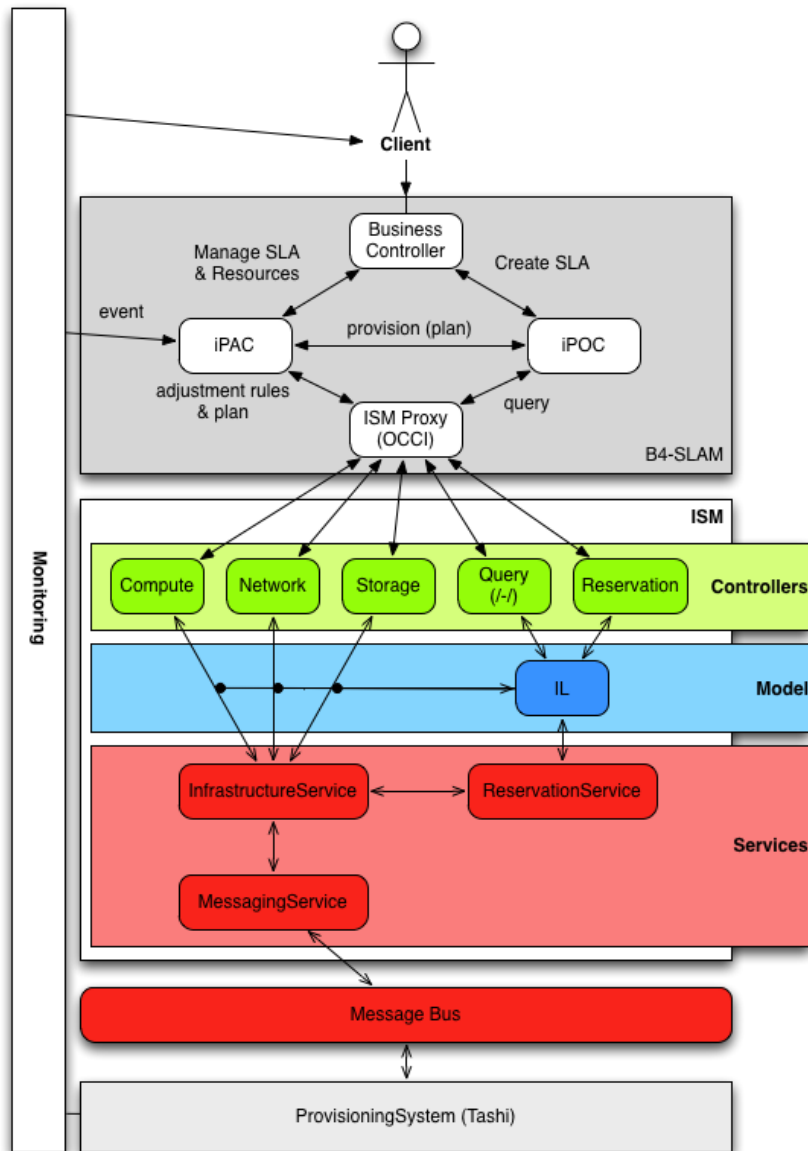


Figure 11 Infrastructure Service Manager

and management of virtual machines. For the SLA@SOI needs and to make Tashi SLA-aware a considerable amount of extensions have been added to Apache Tashi. Tashi is waiting for commands from ISM for provisioning and managing virtual machines. Tashi and ISM communicate using XMPP bridge. The XMPP communication has to be configured in the Tashi configuration file.

Some new SLA terms have been integrated in Tashi, for example CPU speed, location and isolation. If some additional terms need to be added, a Tashi scheduler can be extended or a new scheduler implementation can be provided. For new SLA terms implementation inside scheduler please check the SchedulerCommon class on [SourceForge](#). Please check scheduler description on <https://sourceforge.net/apps/trac/sla-at-soi/wiki/SlasoiSchedulers> for more information.

For information about installation and configuration of the SLA@SOI Tashi you can check <https://sourceforge.net/apps/trac/sla-at-soi/wiki/SlasoiTashi>. The installation of Tashi is quite complex, because it includes some networking

configuration, but it is the same for Apache Tashi and SLA@SOI Tashi version. There is only one additional step to enable the SLA@SOI Tashi. In order to enable the extended schedule parameters (like auditability, isolation, location...) the database table that is holding physical servers information needs to be updated. This can be done easily by using tashi-client.py script:

```
python bin/tashi-client.py setHostConfig -hostId 2 -elemName
auditability -value True
```

or:

```
python bin/tashi-client.py setHostConfig -hostId 2 -elemName
location -value SI
```

In order to extend the remote Tashi API (XMPP commands) a [tashi_commands.py](#) file can be extended. This can be done simply by adding a new BaseCommand implementation. The example command which destroys the virtual machine is:

```
class DeleteComputeCommand(BaseCommand):

    def __init__(self):
        super(DeleteComputeCommand,
self).__init__('deletecompute', 'Deletes a compute resource
based on the supplied model')

    def __call__(self, user, prot, args):
        print "DeleteComputeCommand called"
        try:
            (config, configFiles) = getConfig(["Client"])
            client = createClient(config)
        except:
            print "ERROR ----- could not connect
to tashi -----"

        decoder = json.JSONDecoder()
        req, end = decoder.raw_decode(args)
        resourceReq = req[0]

        vmId = parseIdFromMessage(resourceReq, client)
        if vmId == None:
            prot.send_error("@error VM destroy failed: Id
for a VM to be destroyed was not found", user)
            return

        try:
            client.destroyVm(vmId)
            print "completed deletion of VM id: " + vmId
            print "Sending the following message: " +
"@deletevm " + str(args) + " To: " + str(user)
            prot.send_result("@deletevm " + str(vmId),
user)
        except Exception as e:
```

```
        print e
        prot.send_error("@error VM destroy failed: " +
str(e), user)
```

Tashi XMPP remote commands functionality communicates with Tashi using Client object (please check createClient method).

Sample Python code for invoking DeleteComputeCommand is given below:

```
import xmpp, json

username = 'your_xmpp_username'
passwd = 'your_xmpp_password'
to='your_tashi_listener_user@your_xmpp_server'

client = xmpp.Client('your_xmpp_server')
client.connect(server=('your_xmpp_server',5222))
client.auth(username, passwd, 'your_xmpp_server')
client.sendInitPresence()

msg = 'deletecompute [{"id":1}]'
message = xmpp.Message(to, msg)
message.setAttr('type', 'chat')
client.send(message)

client.disconnect()
```

The virtual machine that is to be started by SLA@SOI framework has to contain the pre-installed use case software. That means ORC in this example. For a preparation of the virtual machine please check the ORC installation and image preparation section. For implementation of the manageability interfaces check the manageability sections. Once the image is prepared, the virtual machine can be started using CreateComputeCommand, which is invoked by OCCI server inside Infrastructure Service Manager.

4.4.3 Service Construction Model

Service Construction & Software Landscape Meta Model

The Service Construction (Meta) Model (SCM) is motivated by the need to store and manage information about services inside the SLA@SOI framework. ServiceManagers have to provide data about the types of services offered, about alternative realizations of these service types, and about the service instances that have already been provisioned. Furthermore, SLAManagers require information about the dependencies of a service on other services, about features of the service itself, and about features of its associated monitoring system. Based on this information, SLAManagers can plan and negotiate SLAs with their customers and acquire other (external) services that are required. The Service Construction (Meta) Model is driven by the information necessary to create, evaluate, and maintain services and their associated SLAs. As such, it is an essential part for the communication of SLAManagers, ServiceEvaluation, and (Software-)ServiceManagers. With the SCM, (Software-)ServiceManagers can maintain information about provided and required service types, available service

implementations, and running service instances. SLAManagers and ServiceEvaluation can retrieve information about service dependencies, about features of the service, and about its monitoring system.

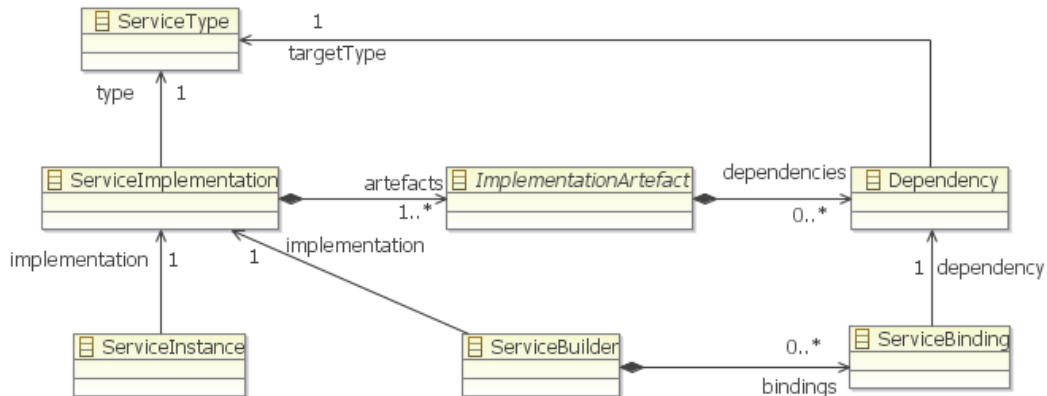


Figure 12: Service Hierarchy

Furthermore, they can resolve dependencies to external services and provide particular configurations of the service and its monitoring system in a generic way. ServiceManagers instantiate services that have been negotiated by the SLAManager based on their configuration. In the following sections, we introduce the basic elements of the SCM and describe their purpose.

During its life-cycle, a service exists in different aggregate states that have to be reflected inside the SLA@SOI framework. The service hierarchy shown in Figure 12 reflects the different states a service can assume: ServiceType, ServiceImplementation, and ServicesInstance.

A ServiceType describes the functionality that a service provides. For example, it contains pointers to the WSDL definitions of the service’s interfaces. The same ServiceType can be realized by multiple ServiceImplementations. A ServiceImplementation consists of a set of ImplementationArtefacts (such as software components, or appliances for virtual machines) that are required to instantiate the service. Each ImplementationArtefact has a set of Dependencies to other services. For example, a software service that is realized by an appliance depends on an infrastructure service that is able to host that particular appliance. Similar to the relation of implementation and instance in object-oriented languages, an arbitrary number of ServiceInstance can be created for each ServiceImplementation.

A ServiceInstance describes the properties of a service that is (about to be) provisioned and accessible. For example, a ServiceInstance contains the endpoint of a particular service. The endpoint either refers to a running service instance or points to the location where the service will be available according to the time constraints defined in the corresponding SLA.

In order to instantiate a service for a customer, various degrees of freedom have to be resolved. For example, all dependencies of an implementation on other services need to be bound to offers of an external service provider or of another SLA Manager. The ServiceBuilder provides a generic way to resolve service dependencies and provide custom configurations for a service and its associated monitoring system. For each dependency, the ServiceBuilder holds a ServiceBinding that maps the Dependency to a SLATemplate or one of its specializations (SLA and BusinessProduct). The SLATemplate contains all information necessary to assess and access a service outside of the current

SLAManager's domain. It includes quality constraints and, after the SLA has been agreed, endpoints of the service.

Software Landscape

Inside the ServiceManager, the Landscape is the central element holding and managing all elements introduced in the previous section (ServiceTypes, ServiceImplementation, ServiceInstance, and ServiceBuilder). The Landscape contains and organizes the various services which are being offered by a service provider. As such, the Landscape is the SCM's root element and can only exist once in each ServiceManager. Figure 13 shows the Landscape and its relations. A Landscape contains a set of provided and required ServiceTypes, a set of implementations of these types, and all instances that are (about to be) provisioned. We explicitly distinguish between ServiceTypes that are offered to customers and ServiceTypes that are required to fulfill their functionality. Furthermore, the Landscape holds all ServiceBuilders that have been used to provision a ServiceInstance. These ServiceBuilders allow for retrieving information about the service's configuration at runtime. In addition to the elements of the SCM, the Landscape contains a ServiceTopology which specifies how different software elements are connected to each other.

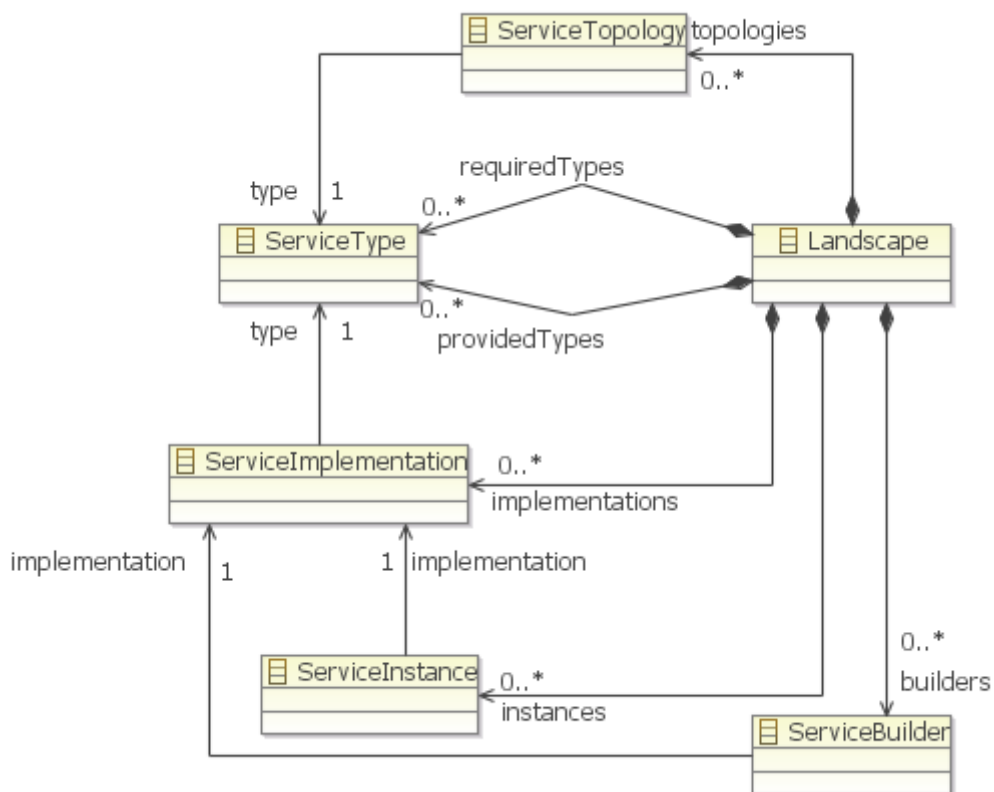


Figure 13: Software Landscape Model

Software Implementation

ServiceImplementations realize a specific ServiceType. They describe i) specific artefacts and assets (such as software components, or appliances) that are required to instantiate a service, ii) the dependencies of assets and artefacts on other services, and iii) the configurable features of the particular service implementation and its associated monitoring system. For example, the "All-In-

One implementation of the ORC's services depends on one infrastructure service that hosts its appliance. The thread pool size of its Active BPEL engine can be adjusted according to the services usage and deployment. Furthermore, the monitoring system allows for instrumenting each service operation and extract response times and throughput.

To express such properties, ServiceImplementations contain a set of ImplementationArtefacts, a set of ComponentMonitoringFeatures, and some ProvisioningInformation. ComponentMonitoringFeatures specify the capabilities of the monitoring system associated to a service. They contain information about the available sensor, effectors and reasoners. ProvisioningInformation and ImplementationArtefacts hold the information necessary to plan and execute the provisioning of a service.

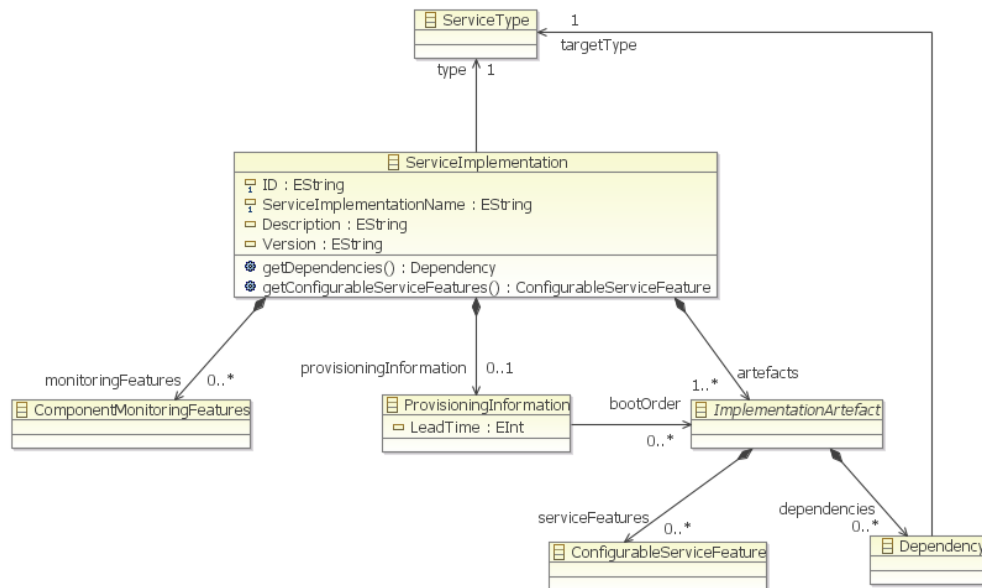


Figure 14: Service Implementation Model

ProvisioningInformation contains the LeadTime that is needed to start a particular software system. Furthermore, it specifies the boot order in which multiple ImplementationArtefacts are to be started. Each ImplementationArtefact represents a single unit such as appliances or software archives that have to be deployed separately. The metamodel of an ImplementationArtefact contains information about its Dependencies and the ConfigurableServiceFeatures associated with this artefact. Figure 15 presents a detailed view of the ImplementationArtefact and its elements.

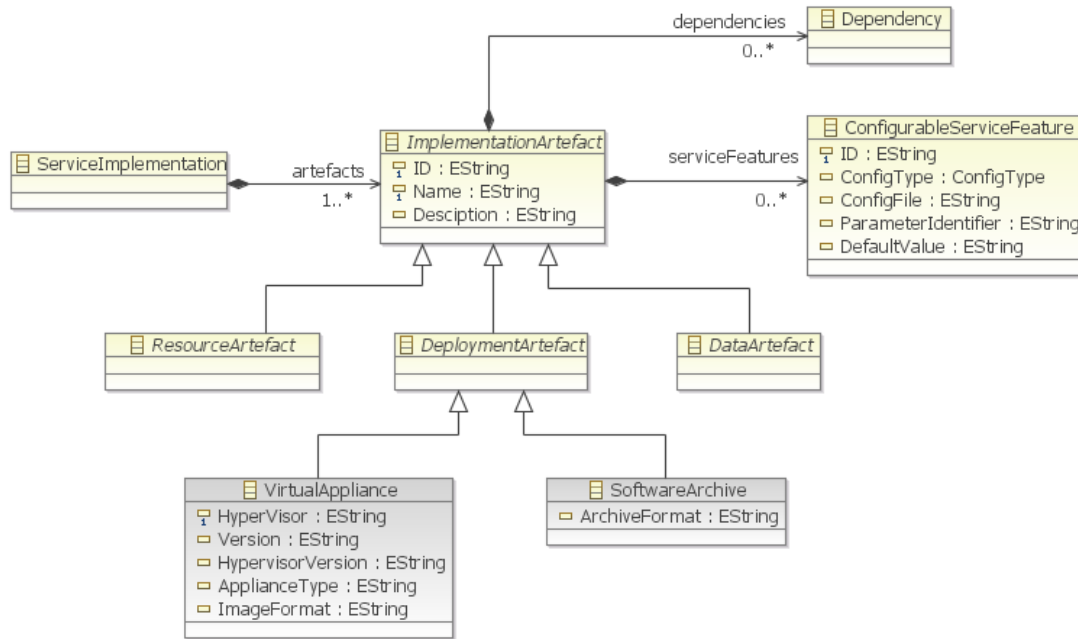


Figure 15: Implementation Artefact

A Dependency refers to ServiceTypes that are necessary to instantiate a given ServiceImplementation. Dependencies are part of ImplementationArtefacts to allow a direct association of a dependency to the artefact that requires it. This is necessary, if multiple artefacts require the same ServiceType with different quality characteristics. In this case, multiple SLAs need to be established for the same service type. Thus, a unique mapping between each SLA and the requiring ImplementationArtefact is necessary. A typical example for such a scenario are multiple dependencies to infrastructure services. If a ServiceImplementation comprises multiple appliances (e.g. one for the application server and one for the database), the model needs to reflect which appliance is to be hosted on which virtual machine.

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

ImplementationArtefacts are abstract entities that have to be specialized for different domains. In Figure 15, we show subclasses for ResourceArtefacts, DeploymentArtefacts, and DataArtefacts. DeploymentArtefacts are further refined to VirtualAppliances and SoftwareArchives. These elements contain detailed information necessary to deploy the artifact.

Please note that ImplementationArtefacts are only used inside a ServiceManager. Thus, the information about the internal structure of a service implementation does not have to be understood by SLAManagers and ServiceEvaluation. For this purpose ServiceImplementations contain explicit operations that aggregate the Dependencies and ConfigurableServiceFeatures for external processing.

Example

In the following section, we give a simple example of ServiceImplementations, Dependencies, ServiceBuilders, ServiceBindings and their usage in the overall system.

In Figure 16 the dependency is resolved by a ServiceBinding that links the Dependency to an SLATemplate of an external provider. The SLATemplate contains the specification of interface "IExternal". The external service type can again be realized by a ServiceImplementation.

In the following, we illustrate how the ServiceBuilders can be used for communication between SLAManagers, (Software-)ServiceManagers, and ServiceEvaluation.

- The SLAManager requests the ServiceImplementations of a particular ServiceType from the ServiceManager
- For each ServiceImplementation
 - The SLAManager creates a ServiceBuilder
 - The SLAManager resolves the dependencies of the ServiceImplementation using available SLATemplates, SLAs, and BusinessProducts
 - When all dependencies have been resolved, the ServiceBuilder is passed to ServiceEvaluation which assesses the expected quality of the setting given by the ServiceBuilder.
 - Steps 1, 2 and 3 may be repeated several times
- When a particular ServiceBuilder is to be instantiated, the SLAManager (tries to) agree on the selected SLATemplates (or SLAs) of the depending services and adds the corresponding Endpoints to the SLATemplates. The resulting object is passed to the ServiceManager which instantiates the requested service based on the settings given in the ServiceBuilder.

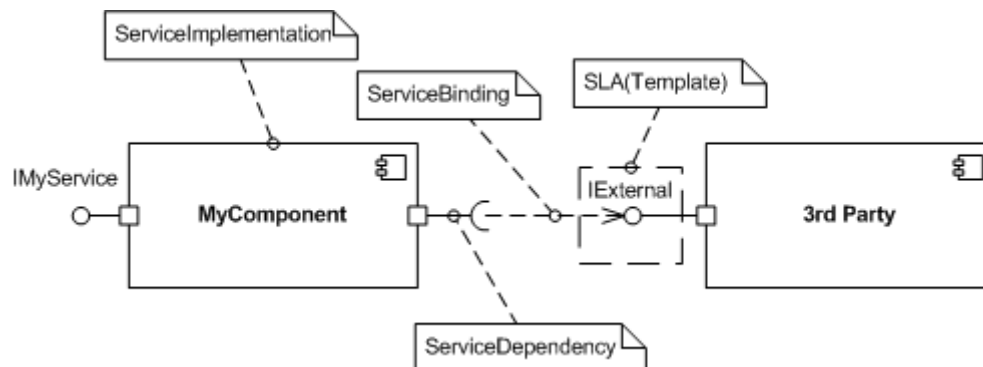


Figure 16: Example SCM

Building Your Own SCM Model

For the creation of a SCM model the [SCM Editor](#) [14] can be used.

Starting Editor and Creating New Project

In the directory "scmEditor/eclipse/" you can find an executable named scmEditor. After starting this executable you will see that scmEditor behaves in the same way as normal Eclipse. Thus you have to select a workspace first. Before a model instance (of a landscape, builder, ...) can be created you have to

create a project in the same way as in eclipse. In order to do this select "File -> New -> Project...", which will lead you to "new Project wizard" where you should select "General -> Project" and click on the "next" button. Now type a project name into the corresponding field and finish the project creation.

Creating New Model Instance

For creation of a new model instance right-click in the "Project Explorer" view on the project you want the instance to create in and then select "New -> Other..." in the context menu. Select "Example EMF Model Creation Wizards" and then the desired meta-model you'd like to create (for example "ServiceConstructionModel Model").

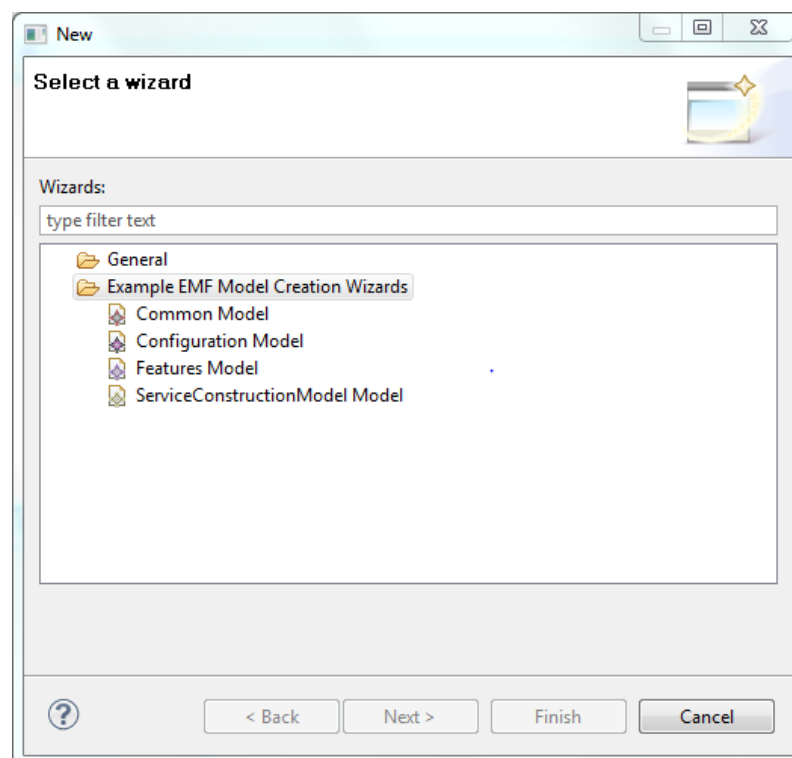


Figure 17: Select Model

Example: Creating SoftwareLandscape

In order to create a software landscape model instance select "ServiceConstructionModel Model" and click "Next". Now type a file name for the model file and then proceed. In the field "Model Object" select the entry Landscape and click in the "Finish" button. Now there should be a new model file in your project. Selecting this file by double-click shows up an editor, which enables you to create and configure your landscape instance. New child-objects or sibling-objects of an object in the editor can be created by right-clicking on the object and selecting the "New child -> ..." or "New Sibling -> ...". In the context menu the type of the new object can be selected. The list under "New Child" shows only the types which are valid for the selected parent object. In the example shown by Figure 17 we are creating an object of type "ServiceType" for the association "Provided Types" which stands for the service types which are provided by our landscape. By creating children for selected objects you can setup your desired landscape step by step. To configure the attributes and

properties of a certain object right-click on the object and select "Show Properties View". In the properties view all attributes of the selected object will be listed.

Note: For the most objects the value of the attribute "ID" is generated automatically and is not changeable.

In order to set a reference R from an object A to an object B (both have to be present in the model instance) you have to select B in the list of valid reference-objects which you get from the drop-down menu of the field (named like the reference R) in the properties-view of object A. The example shown in the image illustrates how to set a reference from ServiceImplementation A to the ServiceType B. If you want to add elements to attributes of list-types (or container in general), you will have to push the button ".." near the corresponding attribute field of the properties view, what will make a window pop up. In that window elements can be added to the list.

4.5 Manageability

In order to support the management of Open Reference Demonstrator we have implemented domain-specific Manageability facades for BPEL processes deployed within the Dynamic Orchestration Engine, and for Axis 1.4.2 atomic services.

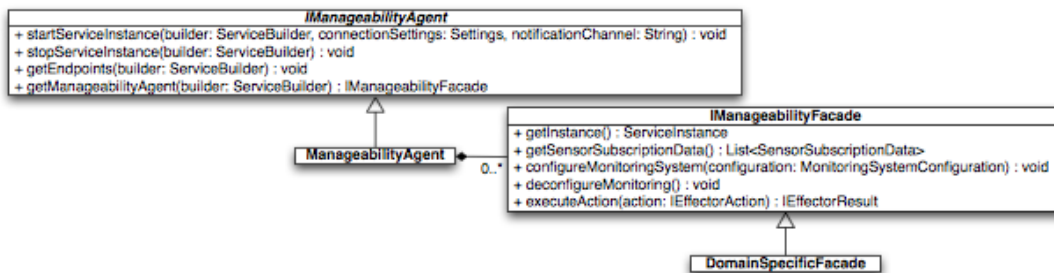


Figure 18: The Manageability Interfaces

The generic interface for a manageability façade is shown in Figure 18. For BPEL processes we implemented a complete façade with all the methods needed for the configuration of sensors and effectors for the processes. For AXIS atomic services we only implemented the methods in the façade that allow for sensor configurations. Effectors for atomic services were not needed for the Open Reference Demonstrator, since we rely on dynamic binding at the process binding level to adjust the demonstrator's behaviour. Since all is provided, no coding is required for running the Open Reference Demonstrator, which can be run out-of-the-box.

In the rest of this section we will show how the façade is used to manage the process deployed within the Dynamic Orchestration Engine. The code would be mostly the same if we were managing atomic services with axis so, due to lack of space, we will concentrate on BPEL processes. Should the user require management capabilities for services implemented using a different technology, he/she would need to provide a specific implementation of the IManageabilityFacade interface. In this case the user can take inspiration from the façade developed for the Dynamic Orchestration Engine.

If the user looks at the façade's implementation, he will notice that the façade is simply a gateway to the extended administration interface we have provided for the Dynamic Orchestration Engine, which can be found at the following url: <http://ORCServices:1030/active-bpel/services/ActiveBpelAdmin>. We are confident

that similar strategies can be used with any specific service implementations the user may have.

The first code snippet illustrates how we can issue the startup of a new process within the Dynamic Orchestration Engine. We start by creating a ManageabilityAgent Service. We then configure the settings needed to connect to the event bus that will carry the notification that the service has been successfully setup, and the name of the notificationChannel used on that event bus. Finally, we issue the startup by sending the manageability agent a ServiceBuilder, containing the service specific info that are needed to actually setup the service, and the above configurations.

```
Servicebuilder builder = new ServiceBuilderExtended();
MAService service = new MAServiceImpl();
Settings connectionSettings = settings;
String notificationChannel = "test-DOE-Paolo";

try {
    service.startServiceInstance(builder, connectionSettings,
notificationChannel);
} catch (ServiceStartupException e) {
    e.printStackTrace();
} catch (MessagingException e) {
    e.printStackTrace();
}
```

The following code exemplifies how a sensor can be setup on the Open Reference Demonstrator's main process using the Dynamic Orchestration Engine's manageability façade.

First we obtain the façade.

```
IManageabilityAgentFacade facade =
service.getManagibilityAgentFacade(builder);
```

Then we prepare the Sensor Configuration we want to adopt. The Sensor Configuration is part of an overall MonitoringConfiguration, an object that can contain different sensor and effector configurations to be enacted by the manageability agent. In this case we have only one component that needs to be configured, a sensor.

```
ConfigurationFactoryImpl factory = new
ConfigurationFactoryImpl();
MonitoringSystemConfiguration msc =
factory.createMonitoringSystemConfiguration();
msc.setUuid(UUID.randomUUID().toString());
Component[] components = new Component[1];
Component c = factory.createComponent();
c.setType("Sensor");
```

In this case the Sensor Configuration is a DOESensorConfiguration. It requires information that are specific to the Open Reference Demonstrator, such as the service's id, the id of a specific operation we want to attach the sensor to, and specific correlation information needed to ensure that the sensor is only active on specific instances of the operation request. The correlation key is the name of a variable we expect to find within the message that launches a new service

instance, and the correlation value is the value we expect to find therein. If the correlation information is found the sensor is activate for that instance, and for the specified operation.

```
DOESensorConfiguration config = new DOESensorConfiguration();
config.setConfigurationId(UUID.randomUUID().toString());
String serviceID = "paymentService";
config.setServiceID(serviceID);
String operationID =
"/process/flow/receive[@name=$$ReceivePaymentRequest$$]";
config.setOperationID(operationID);
String status = "input";
config.setStatus(status);
String correlationKey = "cardNumber";
config.setCorrelationKey(correlationKey);
String correlationValue = "7777";
config.setCorrelationValue(correlationValue);
```

We also provide information regarding where the sensed data will need to be sent to.

```
OutputReceiver[] newOutputReceivers = new
OutputReceiverImpl[1];
OutputReceiver receiver = new
ConfigurationFactoryImpl().createOutputReceiver();
receiver.setEventType("event");
receiver.setUuid("tcp:localhost:10000");
newOutputReceivers[0] = receiver;
config.setOutputReceivers(newOutputReceivers);
```

Finally, we complete the configuration and ask the façade to enact it on the service instance.

```
ComponentConfiguration[] configs = new
ComponentConfiguration[1];
configs[0] = config;
c.setConfigurations(configs);
components[0] = c;
msc.setComponents(components);
facade.configureMonitoringSystem(msc);
```

The following code, on the other hand, can be used to issue an adjustment on the process' bindings. We start by creating an UpdateBinding object, which is an implementation of the IEffectorAction abstract class. This object contains a new ServiceBuilder, which in turn contains the new bindings to be used. We then get the appropriate façade, and issue the changes by calling the method executeAction.

```
IEffectorAction action = new UpdateBinding(builder);
IManageabilityAgentFacade facade =
service.getManagibilityAgentFacade(builder);
IEffectorResult effectorResult = null;
```

```
try {
    effectorResult = facade.executeAction(action);
} catch (Exception e) {
    e.printStackTrace();
}
```

4.6 Predictability

In order to determine the expected performance and reliability of a software service, the SLA management framework includes the functionality to predict the service's response time and probability of failure on demand. The prediction results can be used by software service providers to determine feasible parameters for their offered SLA's and to determine a proper reaction (accept / reject / counter-offer) to an SLA request issued by a potential service customer during the SLA negotiation phase. To this end, the framework includes the *Service Evaluation* component, which carries out the quality predictions. A detailed description of the component and its functionality is given on SourceForge [SLA@SOI wiki](#)⁴. The following sections describe the necessary steps in creating the required prediction models (Section 4.6.1), setting up the prediction functionality (Section 4.6.2), and using the Service Evaluation component for predictions (Section 4.6.3).

4.6.1 Creating Prediction Models

In order to use the prediction functionality at the SLA negotiation time within the SLA management framework, a prediction model must have been created beforehand. The prediction model describes a service-based system under study, including the system's performance- and reliability-related aspects. This task is typically carried out by the service provider, potentially with the help of external software providers, who provide quality models together with their implementations. For each software service whose quality shall be predicted by the Service Evaluation, a corresponding system model has to be created.

The prediction meta-model used in SLA@SOI is based on the Palladio Component Model (PCM), which allows to specify component-based software architectures. Software providers can use the existing PCM tooling to create models of their service-based systems. The tooling is based on Eclipse and provides an integrated environment for graphical modelling of service components, including their behaviour, composition, and allocation to a physical resource environment. Editors for model creation are based on the Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework (GMF). For detailed information about how to install and use the PCM tooling, see the [Palladio homepage](#)⁵. As an example of a service model, a separate [documentation](#) describes the ORC model in detail⁶.

⁴ <http://sourceforge.net/apps/trac/sla-at-soi/wiki/SoftwareServiceEvaluation>

⁵ http://www.palladio-simulator.com/science/palladio_component_model/

⁶ <https://sla-at-soi.svn.sourceforge.net/svnroot/sla-at-soi/platform/trunk/doc/QoS-Model-Creation.doc>

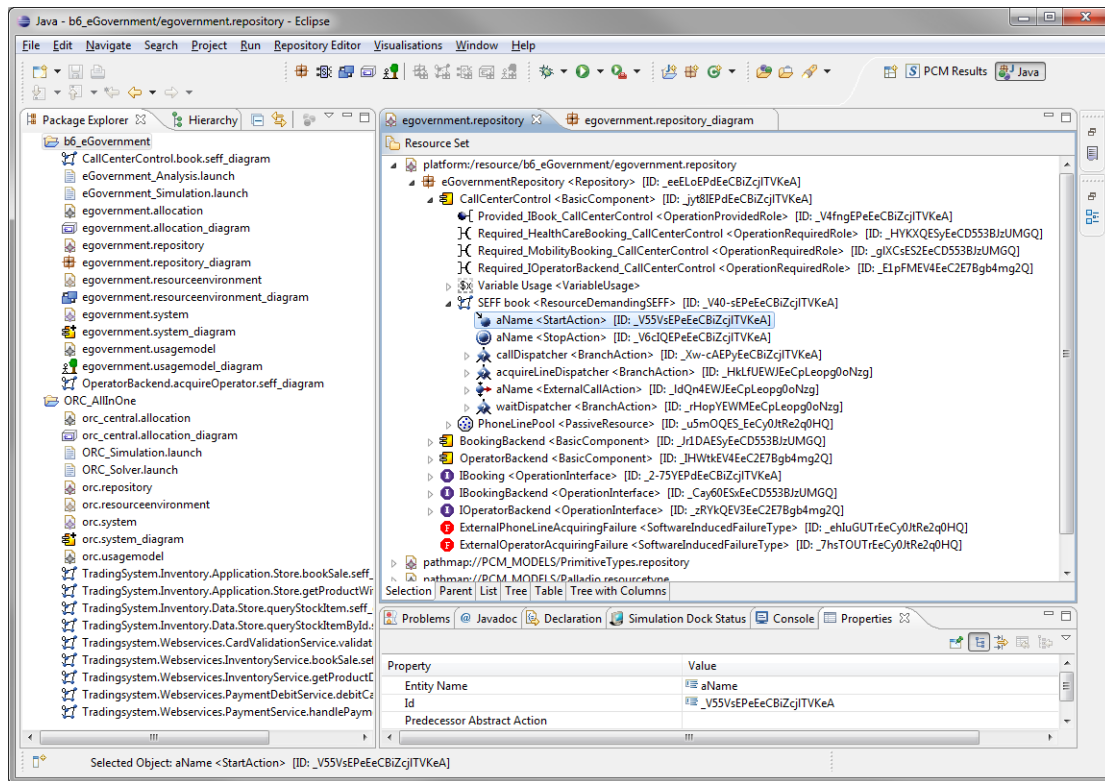


Figure 19: PCM Tooling, EMF Model Editors

To further illustrate the PCM tooling and its usage, Figure 19 shows a screenshot of the Eclipse workbench with the two pre-defined ORC and B6 models loaded as projects into the workspace (as shown in the Package Explorer view, left-hand side of the figure). Each project contains configuration files, diagram files, and the model files themselves. Individual model files can be edited via the standard EMF model editors, which organize the display of model elements in a tree view (as shown in the central part of the figure). The properties of individual model elements can be seen and edited in the Properties view (at the bottom).

Figure 20 demonstrates the graphical representation of model contents through GMF editors. In the figure, the service components of the E-Government prediction model and their provided and required interfaces are visualized by the editor. The editor allows for altering the model contents (e.g. create new components or change the interfaces of a component) through drag-and-drop operations.

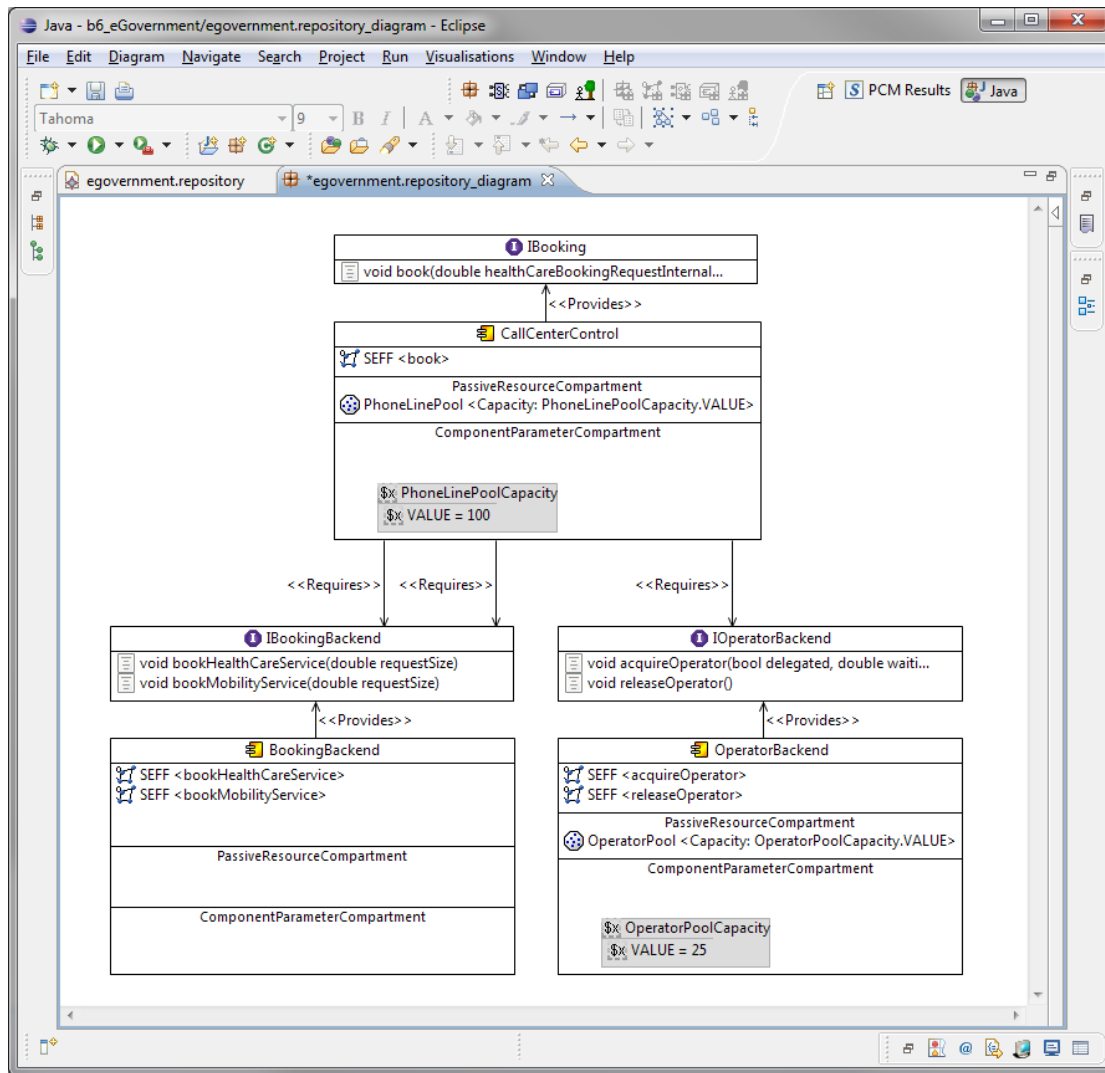


Figure 20: PCM Tooling, GMF Model Editors

4.6.2 Setting up the Prediction Functionality

In order to set up and run the Service Evaluation to conduct quality predictions, the following steps are required:

1. **Install the Service Evaluation Server.** This server (which is an extended Palladio installation) performs the predictions. It communicates with the SLA management framework via a web service interface. For detailed instructions on how to download and install the server, see the Section "Service Evaluation Server" on the [Service Evaluation page](#) on the SourceForge Wiki⁷.
2. **Setup the SLA@SOI platform.** The SLA management framework is delivered in terms of a runnable OSGi platform. The service evaluation is executed as a bundle within the platform. For instructions how to check out and build the platform, see [How to build the SLA@SOI platform](#)⁸.

⁷ <http://sourceforge.net/apps/trac/sla-at-soi/wiki/SoftwareServiceEvaluation>

⁸ <http://sourceforge.net/apps/trac/sla-at-soi/wiki/BuildPlatform>

3. **Fill the Prediction Model Repository.** In order to enable quality predictions by the Service Evaluation component, prediction models have to be prepared for each service-based system of interest (see Section 4.6.1). To this end, existing prediction models can be placed into the folder "`<platform / trunk> / common / osgi-config / software-servicemanager / quality_model_repository`". Each prediction model placed here needs its own sub folder. Two models exist per default, one for the ORC, and one the eGovernment use case in SLA@SOI. Further models can be added as required.
4. **Start the Service Evaluation Server.** Start the server according to the [instructions](#) on the Wiki⁴.
5. **Run the platform.** Execute the platform according to the instructions at [How to execute the SLA@SOI platform](#)⁹. In the console of the running platform, type "ss" to see all installed bundles. The following bundles will appear with status "ACTIVE" in the list: "org.slasoi.service-evaluation.prediction", "org.slasoi.service-evaluation.repository". Upon typing "bundle [XXX]", where "[XXX]" refers to the prediction bundle number, the registered service "IServiceEvaluation" shows that the service evaluation interface is available for service quality prediction.

4.6.3 Using Quality Predictions

Once the SLA@SOI platform is up and running as described in Section 4.6.2, the "org.slasoi.seval.prediction.service.IServiceEvaluation" interface of the Service Evaluation component (bundle "org.slasoi.service-evaluation.prediction") can be invoked to conduct performance and reliability predictions for the software services under study. To this end, the interface offers the operation "Set<IEvaluationResult> evaluate(Set<ServiceBuilder> realizations, SLATemplate template)", which expects an SLATemplate for the target service to evaluate as an input, as well as a set of ServiceBuilders describing multiple possible realizations of the requested service with specified quality parameters of the required external software and infrastructure services. In return, the operation provides the predicted quality of each realization in terms of an IEvaluationResult. For further details regarding the involved data structures, see the corresponding Wiki page⁴.

To test the operation with respect to the ORC scenario, execute either the corresponding unit tests of the prediction bundle ("org.slasoi.seval.prediction.AllPredictionTests"), or execute the integration test in the running platform by typing "invoke [XXX] run scenario=1 NO=14", where "[XXX]" refers to the bundle number of the integration test bundle. For further details about the execution of the integration tests, see the Wiki¹⁰.

4.7 Monitorability

4.7.1 Monitoring Manager

The MonitoringManager (MM) coordinates the generation of a monitoring configuration of the system. It decides, for an SLA specification instance it receives, which is the most appropriate monitoring configuration according to

⁹ <http://sourceforge.net/apps/trac/sla-at-soi/wiki/ExecutePlatform>

¹⁰ <http://sourceforge.net/apps/trac/sla-at-soi/wiki/DocumentationIntegrationTest>

configurable selection criteria. A monitoring configuration describes which components to configure and how their configurations can be used to obtain results of monitoring Guaranteed States. There are three types of Monitoring Features in the monitoring system. First, Sensors collect information from a service instance. Their designs and implementations are domain-specific. A sensor can be injected into the service instance (e.g., service instrumentation), or it can be outside the service instance intercepting service operation invocations. A sensor can send the collected information to a communication infrastructure (e.g. an Event Bus) or other components can request (query) information from it. There can be many types of sensors, depending on the type of information they want to collect, but all of them implement a common sensor interface. The interface provides methods for starting, stopping, and configuring a sensor. Second, Effectors are components for configuring service instance behaviour. Their designs and implementations are also domain-specific. An effector can also be injected into a service instance or can interface with a service configuration. There can be many types of effectors, depending on the service instance to be controlled, but all of them implement a common effector interface. The interface provides methods for configuring a service. The third type of monitoring feature is a Reasoner (or also known as a Reasoning Engine) which performs a computation based upon a series of inputs provided by events or messages sent from sensors or effectors. An example reasoner may provide a function to compute the average completion time of service requests.

In this case it accepts events from sensors detecting both requests and responses to a service operation and computes an average over a period of time. Reasoners also provide access to generic runtime monitoring frameworks such as EVEREST.

Configuring the Monitoring Manager Environment

The MM has two configuration files 1) Term Event Types Required, 2) Term Result Types. The configuration files are required to match appropriate events for terms stated in Guarantee States and for matching appropriate reasoners for the results expected of term types. These are located and configured as follows:

```
Event Types Map - <SLASOI_HOME>\generic-slamanager\monitoring-  
manager\term-events-map.cfg
```

```
the configuration file is based upon a single line entry for  
each term and event type pairing. ** example:
```

```
http://www.slaatsoi.org/commonTerms#availability,REQUESThttp://  
/www.slaatsoi.org/commonTerms#availability,RESPONSE
```

```
In this example the term 'availability' requires both a  
service REQUEST and RESPONSE event (from sensor features  
offering these event types).
```

```
Result Types Map - <SLASOI_HOME>\generic-  
slamanager\monitoring-manager\term-resulttype-map.cfg
```

```
The configuration file is based upon a single line entry for  
each term and result type pairing. ** example:
```

```
http://www.slaatsoi.org/commonTerms#availability,http://www.sl  
aatsoi.org/types#NUMBERhttp://www.slaatsoi.org/commonTerms#com  
pletion_time,http://www.slaatsoi.org/types#NUMBER
```

```
In this example the terms 'availability' and 'completion_time'  
result in a NUMBER type.
```

Creating an Instance of the Monitoring Manager

A new instance of the [MonitoringManager](#) [15] can be created using the `MonitoringManagerBuilder`¹¹ class and then used as follows:

```
MonitoringManagerBuilder MMBuilder = new
MonitoringManagerBuilder();
IMonitoringManager MM = MMBuilder.create();
```

Checking Monitorability

The [MonitoringManager](#) class has only one method - `checkMonitorability`. The method accepts an instance of a SLA model and an array of monitoring component features (`ComponentMonitoringFeatures`¹²). Note for the specification of the SLA model, please refer to the documentation of `org.slasoi.slamodel.sla.SLA`. For the specification of monitoring component features please refer to the documentation of `ComponentMonitoringFeatures`. The `checkMonitorability` method returns an instance of a `MonitoringSystemConfiguration` (which may be empty).

```
SLA sla = (create an instance of an SLA)...;
ComponentMonitoringFeatures[] features = (see below for
common features)...;
MonitoringSystemConfiguration config = null;
config = MM.checkMonitorability(sla, features);
```

Common Aspects of Monitoring Configurations

The monitoring manager requires a number of default reasoners for agreement term and guarantee term reasoning. This is in addition to any specific terms used in guaranteed term expressions.

- Agreement Term Reasoner: Requires a series term reasoner, input type as array of `BOOLEAN` and output as type `BOOLEAN`.
- Series (array of `BOOLEAN`) term reasoner example: Since array of `BOOLEAN` does not currently exist in the SLA model, this is declared locally in the monitoring features. An example construction of this type of reasoner is as follows:

```
FeaturesFactoryImpl ffi = new FeaturesFactoryImpl();
Function fimpl = ffi.createFunction();
fimpl.setName(core.$series);
fimpl.setDescription("Reasoner for a series of
BOOLEAN");
Basic fInput1 = ffi.createPrimitive();
fInput1.setName("param1");

fInput1.setType("http://www.slaatsoi.org/types#array_of_BOOLEAN");

fInput[0] = fInput1;
fimpl.setInput(fInput);
```

¹¹ `org.slasoi.gslam.monitoring.manager.MonitoringManagerBuilder`

¹² `org.slasoi.monitoring.common.features.ComponentMonitoringFeatures`

```
Basic fOutput = ffi.createPrimitive();
fOutput.setName("output1");
fOutput.setType(meta.$BOOLEAN);
fimpl.setOutput(fOutput);
```

4.7.2 SLA monitoring using the ASTRO engine

One of the two monitoring engines, that are used within the ORC to monitor the KPIs and Guarantee Terms defined in the agreed SLA, is the ASTRO monitoring engine, developed by FBK.

Within the SLA@SOI platform, the ASTRO monitoring engine is embedded inside an instance of a Reasoning Component Gateway (RCG). The role of the RCG is that of mediating the interactions that take place between the framework components and the monitoring engine (through a Java API) and between the external world services and the monitoring (through an event bus). The RCG API provides a set of methods for configuring, starting, and stopping the monitoring engine. The module that implements the RCG that is coupled with the ASTRO monitoring engine is named *fbk-rcg*.

In order to use the monitoring engine, one has to create a client bundle from the where the monitoring engine services, exposed by the *fbk-rcg* component, are injected and started. For the ORC, this has been implemented within the integration-test module by the development of the monitoring scenario.

The following steps were performed to use the ASTRO monitoring engine within the ORC.

Monitoring Service Injection

The OSGi service named *FbkRcgService* has been injected into the platform within the client bundle represented by the class *SpringDMClient*. This class can be found inside the integration test module and the approach used for the injection is the *Service Tracker* one:

```
tracker = new ServiceTracker(context,
FbkRCGService.class.getName(), null);
tracker.open();
rcgFBKService = (FbkRCGService) tracker.waitForService(2000);
```

A possible alternative for injecting the service that might be used is the so called *Service Reference* approach:

```
@ServiceReference
public void setFBKRcgServices( FbkRCGService rcgFBKService );
```

Monitoring Configuration

The monitoring engine uses an event bus as a communication channel. The configuration required for allowing the interaction of the monitoring engine with the framework and with the services can be found within the files *interactioneventbus.properties* and *monitorresulteventbus.properties* inside the *\$SLASOI_HOME/fbk-rcg/properties* folder. Both files refer to the AMQP and to the XMPP protocol since both are supported by the SLA@SOI framework. The *interactioneventbus* file configures the channel where the monitoring engine subscribes for interaction events and the *monitorresulteventbus* configures the

channel where the monitoring engine publishes the monitor results. Sample configuration files are reported below:

Interactioneventbus.properties:

```
xmpp_username=interaction-reader
xmpp_password=slasoi
xmpp_host=slasoi
xmpp_port=5222
xmpp_pubsubservice=pubsub.naruto
messaging=xmpp
pubsub=xmpp
xmpp_service=slasoi
xmpp_resource=test
```

Monitoringresulteventbus.properties:

```
xmpp_username=result-sender
xmpp_password=slasoi
xmpp_host=slasoi
xmpp_port=5222
xmpp_pubsubservice=pubsub.naruto
messaging=xmpp
pubsub=xmpp
xmpp_service=slasoi
xmpp_resource=test
```

Of course here it is assumed that an instance of an event bus is actually running at the configured locations.

Monitoring Start

The startup of the monitoring takes place when the *startMonitoring* operation of the *fbk-rcg* component is invoked.

The operation takes in input a string representing a configuration ID, which in principle should be used to track different configuration instances, but that is not used in current implementation and, thus, can be any string.

Using the following code snippet one can start the monitoring engine:

```
String configurationId = "gggg:ggggg:gggg:gggg";
fbkrcgService.startMonitoring(configurationId);
```

Add SLA to Monitoring Configuration.

When the monitoring engine has been started, it waits for an SLA to be monitored. A set of invocations are needed to start monitoring a given SLA.

The SLA must be set within a *ReasonerConfiguration* object, which then in turn is added to the configuration of the *fbk-rcg*.

Within the ORC scenario, the monitoring engine is started after the negotiation/provisioning scenarios execution, when the SLA that was agreed between the customer and provider has been added to the SLARegistry. The SLA at this point is retrieved from the SLA registry and passed to the monitoring engine. The following code snippet demonstrates the invocation of this operation that takes place within the *RunTimeMonitoringScenario* class of the integration test module:

```
String slaUuid = "Uuid of provisioned SLA";
org.slasoi.slamodel.primitives.UUID[] slaUUID = new
org.slasoi.slamodel.primitives.UUID[] {new
org.slasoi.slamodel.primitives.UUID( slaUuid) };
SLA[] slas =
sslamContext.getSLARegistry().getIQuery().getSLA(slaUUID);
ConfigurationFactory cf = new ConfigurationFactoryImpl();
ReasonerConfiguration rc = cf.createReasonerConfiguration();
rc.setSpecification(slas[0]);
fbkRcg.addConfiguration(rc);
```

Monitoring Execution

When the monitoring engine starts monitoring the SLA, it is actually waiting for events on the bus to come. As soon as the events generated by the ORC services arrive, the evaluation process of the monitoring engine starts.

Inside the ORC, the ASTRO monitoring engine is used to monitor the agreement terms related to the ORCPayment service. The interaction events for this service can be pushed onto the bus both by executing the ORC application or using an external event generator application developed within the SLA@SOI project. With this application, it is possible to configure different service simulation parameters like number of process interaction, response time threshold, error frequency etc.

After publishing the set of events to the interaction channel of the messaging bus, the monitoring engine reacts to it and performs evaluation. During the evaluation, the monitoring engine will place the results (terms violations, KPI values) on the result channel of the bus in the guise of events. In this way, such events are made available to the other framework components (e.g., reporting) for successive elaboration.

A sample of a monitoring result event is shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Event xmlns="http://www.slaatsoi.org/eventschema">
  <EventID>
    <ID>53</ID>
    <EventTypeID>ORCPaymentServiceStop</EventTypeID>
  </EventID>
  <EventContext>
    <Time>
      <Timestamp>1309271700636</Timestamp>
    </Time>
    <Notifier>
      <Port>0</Port>
    </Notifier>
    <Source/>
  </EventContext>
  <EventPayload>
    <MonitoringResultEvent>
      <SLAInfo assessmentResult="satisfaction"
        slaURI="http://www.slasoi.org/ORCSLAT1.xml"
        slaUUID="f954590d-6c3e-4b0b-a320-722190edc86a">
        <AgreementTerm assessmentResult="satisfaction">
          <GuaranteedState>
            <QoSName>ORCResponseTimePaymentState</QoSName>
            <QoSValue>true</QoSValue>
          </GuaranteedState>
        </AgreementTerm>
      </SLAInfo>
```

```
<MonitoringInfo/>
<ExtraProperties/>
</MonitoringResultEvent>
</EventPayload>
</Event>
```

Monitoring Stop

The *fbk-rcg* component guarantees the possibility of stopping the monitoring through the *stopMonitoring* operation.

The operation takes in input a string representing a configuration id, which in principle should be used to track different configuration instances, but that is not used in current implementation and, thus, can be any string.

The following code snippet demonstrate the invocation of this operation:

```
String configurationId = "gggg:ggggg:gggg:gggg";
fbkrcgService.stopMonitoring(configurationId);
```

4.7.3 Infrastructure Monitoring

The Infrastructure Monitoring Agent (IMA) is an entity responsible for infrastructure layer monitoring and verifying the compliance of infrastructure services with the SLA. It collects metrics data from external monitoring software (e.g. Ganglia), processes it, stores metrics history, computes QoS terms and verifies their compliance with the SLA. In case any violations are found IMA notifies higher-level components (the PAC) and stores violations history. If any QoS term is nearing the violation threshold IMA emits warning of potential SLA violation. IMA also provides an interface for other components to get monitoring data.

Registering new infrastructure service

To register new infrastructure service and initiate its monitoring a RegisterService request has to be sent to the IMA through publish/subscribe messaging protocol, to the configuration channel.

A infrastructure service is a collection of resources (i.e. virtual machines) and corresponds to the infrastructure SLA. RegisterService request is sent by the Infrastructure Service Manager when it receives new infrastructure SLA. The request message contains information about the infrastructure service, list of virtual machines corresponding to the service and guaranteed QoS terms from the SLA for both the service and all VMs. Guaranteed terms can be specified in two ways:

- as a list of AgreementTerm objects. Each AgreementTerm specifies the name of the guaranteed QoS term, violation constraint, the constraint unit in case of numerical terms and optionally warning constraint.
- as a instance of MonitoringSystemConfiguration which is generated by the Monitoring Manager based on the SLA and monitoring features of the IMA. The IMA parses the MonitoringSystemConfiguration object and extracts QoS terms constraint expressions.

A whole code sample for generating RegisterService request and sending it to the IMA can be seen in the unit test helper class [ServiceRegistrationHelper.java](#). The procedure is as follows:

First subscribe to IMA configuration channel and register message listener to receive response to the RegisterService request:

```
pubSubManager.subscribe("INF_MON_AGENT_CONFIG");
messageListener = new MessageListener() {
    public void processMessage(MessageEvent messageEvent) {
        String payload = messageEvent.getMessage().getPayload();
        try {
            PubSubResponse response =
PubSubResponse.fromJson(payload);
            responses.add(response);
        }
        catch (Exception e) {
            fail("Invalid PubSubResponse message.");
        }
    }
};
pubSubManager.addMessageListener(messageListener);
```

Create RegisterServiceRequest object and set unique message ID, recipient, service name and unique service URI:

```
RegisterServiceRequest request = new
RegisterServiceRequest();
request.setMessageId(UUID.randomUUID().toString());

request.setRecipient(InfrastructureMonitoringAgent.APPLICATION
_NAME);
request.setServiceName("PayrollSystem");

request.setServiceUrl("slasoi://company.com/Service/PayrollSys
tem");
```

Specify service level QoS terms. In this sample QoS terms are given explicitly as AgreementTerm objects. Constructor parameters are QoS term name, violation constraint (threshold for numerical terms), warning constraint (threshold) if applicable and unit. Service Availability term can be specified as follows:

```
request.addAgreementTerm(new
RegisterServiceRequest.AgreementTerm(
    MetricTypeEnum.SERVICE_AVAILABILITY, "99.9", "99.93",
"%"));
```

Define VMs, add them to the infrastructure service and specify VM level QoS terms, for example:

```
RegisterServiceRequest.Vm vm1 = new
RegisterServiceRequest.Vm
("payrollsystem-vm1", "payrollsystem-vm1.openlab.com");
vm1.setClusterName("tashi.openlab.com");
request.putVm(vm1);
vm1.addAgreementTerm(new
RegisterServiceRequest.AgreementTerm(
    MetricTypeEnum.VM_CORES, "2", null, ""));
```

Create PubSubMessage containing the RegisterServiceRequest object serialized to JSON format and publish it to the IMA configuration channel:

```
PubSubMessage message = new
PubSubMessage("INF_MON_AGENT_CONFIG",
    request.toJson());
pubSubManager.publish(message);
```

A sample RegisterService request message serialized to JSON:

```
{
  "serviceUrl" : "slasoi://company.com/Service/PayrollSystem",
  "recipient" : "InfrastructureMonitoringAgent",
  "vmList" : [{
    "fqdn" : "payrollsystem-vm1.openlab.com",
    "agreementTerms" : [{
      "violationThreshold" : "99",
      "guaranteedTerm" : "VM_AVAILABILITY",
      "unit" : "%"
    }, {
      "violationThreshold" : "1",
      "guaranteedTerm" : "VM_CORES",
      "unit" : ""
    }, {
      "violationThreshold" : "1.5",
      "guaranteedTerm" : "VM_CPU_SPEED",
      "unit" : "GHz"
    }, {
      "violationThreshold" : "false",
      "guaranteedTerm" : "VM_DATA_ENCRYPTION",
      "unit" : ""
    }, {
      "violationThreshold" : "STANDARD",
      "guaranteedTerm" : "VM_DISK_THROUGHPUT",
      "unit" : ""
    }, {
      "violationThreshold" : "tashil.img",
      "guaranteedTerm" : "VM_IMAGE",
      "unit" : ""
    }, {
      "violationThreshold" : "STANDARD",
      "guaranteedTerm" : "VM_NET_THROUGHPUT",
      "unit" : ""
    }, {
      "violationThreshold" : "256",
      "guaranteedTerm" : "VM_MEMORY_SIZE",
      "unit" : "MB"
    }, {

```

```

        "violationThreshold" : "2048",
        "guaranteedTerm" : "VM_MEMORY_SPEED",
        "unit" : "MHz"
    }, {
        "violationThreshold" : "false",
        "guaranteedTerm" : "VM_PERSISTENCE",
        "unit" : ""
    }, {
        "violationThreshold" : "SI",
        "guaranteedTerm" : "VM_LOCATION",
        "unit" : ""
    }
    ],
    "clusterName" : "tashi.openlab.com"
}, {
    "fqdn" : "payrollsystem-vm2.openlab.com",
    "agreementTerms" : [
        ...
    ],
    "clusterName" : "tashi.openlab.com"
}
],
"requestType" : "RegisterServiceRequest",
"messageId" : "9D2C54B0-5A0C-AB1C-2F99-B754F5C38152",
"agreementTerms" : [{
    "violationThreshold" : "true",
    "guaranteedTerm" : "SERVICE_AUDITABILITY",
    "unit" : ""
}, {
    "violationThreshold" : "WORKING_DAYS",
    "guaranteedTerm" : "SERVICE_AVAILABILITY_RESTRICTIONS",
    "unit" : ""
}, {
    "violationThreshold" : "99.9",
    "guaranteedTerm" : "SERVICE_AVAILABILITY",
    "unit" : "%"
}, {
    "violationThreshold" : "1",
    "guaranteedTerm" : "ACCEPTABLE_SERVICE_VIOLATIONS",
    "unit" : ""
}, {
    "violationThreshold" : "PUBLIC",
    "guaranteedTerm" : "SERVICE_DATA_CLASSIFICATION",
    "unit" : ""
}, {

```

```

        "violationThreshold" : "STANDARD",
        "guaranteedTerm" : "SERVICE_HARDWARE_REDUNDANCY_LEVEL",
        "unit" : ""
    }, {
        "violationThreshold" : "NOT_IMPORTANT",
        "guaranteedTerm" : "SERVICE_ISOLATION",
        "unit" : ""
    }, {
        "violationThreshold" : "24",
        "guaranteedTerm" : "SERVICE_MTTT",
        "unit" : "hours"
    }, {
        "violationThreshold" : "15",
        "guaranteedTerm" : "SERVICE_MTTR",
        "unit" : "minutes"
    }, {
        "violationThreshold" : "12",
        "guaranteedTerm" : "SERVICE_MTTV",
        "unit" : "hours"
    }, {
        "violationThreshold" : "DAY",
        "guaranteedTerm" : "SERVICE_REPORTING_INTERVAL",
        "unit" : ""
    }, {
        "violationThreshold" : "NOT_IMPORTANT",
        "guaranteedTerm" : "SERVICE_SAS70_COMPLIANCE",
        "unit" : ""
    }, {
        "violationThreshold" : "CCR_REGION_ALLOWED",
        "guaranteedTerm" : "SERVICE_CCR",
        "unit" : ""
    }, {
        "violationThreshold" : 2,
        "unit" : "",
        "guaranteedTerm" : "VM_QUANTITY"
    }
    ],
    "serviceName" : "TravelSystem"
}

```

Wait on the response from the IMA. Response message can be paired with the request message by the message ID:

```

long startTime = System.currentTimeMillis();
while (responses.size() < 1 &&
    System.currentTimeMillis() - startTime < 30000) {

```

```

    Thread.sleep(50);
}
assertEquals(responses.size(), 1);
PubSubResponse response = responses.get(0);
assertEquals(response.getInReplyTo(),
request.getMessageId());

```

Response status should be OK:

```

assertEquals(response.getStatus(),
PubSubResponse.Status.OK);

```

When the IMA receives a RegisterService request it registers the new infrastructure service and starts monitoring it.

Retrieving monitoring data from the IMA

Monitoring data can be retrieved from the IMA through a publish/subscribe messaging protocol. This can be done by generating request message and sending it to the monitoring data channel. The IMA receives the request, generates the appropriate response message and returns it via the same channel.

Several operations are available:

- **GetHostsInfo:** returns a list of all host machines with certain host metrics and a list of all guest machines (VMs) for each host with certain VM metrics
- **GetMetricHistory:** returns metric value history for specified metric and time interval of specified VM or infrastructure service. The density of values is limited by the maximum number of points.
- **GetSchedulerEvents:** returns scheduler events history
- **GetServiceEvents:** returns cluster manager audit records history related to given infrastructure service or user
- **GetServicesInfo:** returns a list of all infrastructure services, for each service state, some details and statistics and a list of resources. For each resource certain metrics are given.
- **GetServiceSLASummary:** returns SLA summary for given infrastructure service which contains a list of all service QoS terms and a list of all VM QoS terms for each resource. For each QoS term a current value, violation threshold and state (compliance with the SLA) is given.
- **GetServiceViolationsFrequency:** returns SLA violations and warnings frequency for given infrastructure service per reporting periods
- **GetServiceViolations:** returns a detailed list of SLA violations and warnings for given infrastructure service for given time period
- **GetInfrastructureInfo:** returns monitored cluster infrastructure info
- **GetSysInfo:** returns details and metric values for given host or VM

The following code snippets represent retrieving history values of specified metric. The whole sample is available in the [GetMetricHistoryTest.java](#).

Subscribe to the IMA monitoring data channel and register message listener to listen for the response message:

```

pubSubManager.subscribe("FRAMEWORK_COMMAND_CHANNEL");
pubSubManager.addMessageListener(new MessageListener() {

```

```

public void processMessage(MessageEvent messageEvent) {
    String payload = messageEvent.getMessage().getPayload();
    try {
        PubSubResponse resp =
PubSubResponse.fromJson(payload);
        if
(resp.getResponse().equals("GetMetricHistoryResponse")) {
            response =
GetMetricHistoryResponse.fromJson(payload);
        }
    }
    catch (Exception e) {
        fail("Invalid GetMetricHistoryResponse message.");
    }
}
});

```

Create `GetMetricHistoryRequest` object, set message ID, service URI and metric for which you want to receive the history. In case the metric/QoS term applies to a VM you have to set also the fully qualified name of the VM. A time interval (fromDate and toDate) can be set for which you want to retrieve the metric history. If interval is not given the IMA returns the whole metric history from the service creation. You can also set a `maxNumberOfValues` property to limit the number of historical values.

```

GetMetricHistoryRequest request = new
GetMetricHistoryRequest();
request.setMessageId(UUID.randomUUID().toString());

request.setServiceUri("slasoi://company.com/Service/PayrollSystem");
request.setMetricType(MetricTypeEnum.SERVICE_AVAILABILITY);
request.setMaxNumberOfValues(50);

```

Create `PubSubMessage` containing the `GetMetricHistory` request serialized to JSON format and publish it to the monitoring data channel:

```

PubSubMessage pubSubMessage = new
PubSubMessage("FRAMEWORK_COMMAND_CHANNEL",
request.toJson());
pubSubManager.publish(pubSubMessage);

```

A sample `GetMetricHistory` request message serialized to JSON:

```

{"metricType":"SERVICE_AVAILABILITY","serviceUri":"slasoi://company.com/Service/PayrollSystem","maxNumberOfValues":"50","messageId":"B26EBC61-2EFF-C09C-5CEB-D63735BAA74D","requestType":"GetMetricHistoryRequest"}

```

Wait for the response:

```

long startTime = System.currentTimeMillis();
while (response != null &&
System.currentTimeMillis() - startTime < 15000) {

```

```
Thread.sleep(50);  
}
```

Response message can be paired with the request message by the message ID. Shortened response message serialized in JSON format for the above request:

```
{  
  "serviceUri" : "slasoi://company.com/Service/PayrollSystem",  
  "metricType" : "SERVICE_AVAILABILITY",  
  "metricUnit" : "%",  
  "violationThreshold" : "99.9",  
  "metricValues" : [{  
    "value" : "99.92148",  
    "timestamp" : "2011-06-14T16:51:57.000+0200"  
  }, {  
    "value" : "99.93072",  
    "timestamp" : "2011-06-15T09:32:43.200+0200"  
  }, {  
    "value" : "99.93814",  
    "timestamp" : "2011-06-16T02:13:29.400+0200"  
  },  
  ...  
],  
  "inReplyTo" : "B26EBC61-2EFF-C09C-5CEB-D63735BAA74D",  
  "responseType" : "GetMetricHistoryResponse",  
  "timestamp" : "2011-06-28T14:26:41.619+0200",  
  "originator" : "InfrastructureMonitoringAgent"  
}
```

4.8 Adjustment

4.8.1 BPEL Adjustment

BPEL adjustment deals with structural properties of the BPEL file which encodes an underlying business-process. Namely, BPEL adjustment can parallelize some sequential parts of the process. Of course, parallelized parts should be independent. This kind of modifications to the BPEL structure can be stipulated by the need to accelerate process execution and to avoid some faulty situations that cannot be comprised in a sequential process.

BPEL adjustment module is provided as a module separated from the SLA@SOI platform. It will soon be available on a dedicated SourceForge project named "BPEL Management Toolkit" along with other service management utility modules. No usage of the BPEL adjustment feature had been planned for the ORC. Instead, BPEL adjustment has been used within the SLA@SOI Use Case on E-Government.

BPEL adjustment is provided as a jar file and has no dependencies either on the other modules or on external libraries.

The module that is going to utilize BPEL adjustment should contain the following lines:

```

import eu.fbk.soa.stradj.BpelAdjustment;
BpelAdjustment bpel =
    new BpelAdjustment(infile, restrictions_file);
bpel.adjust();
bpel.printGraph(outfile);

```

Where

- `infile` is a path to the original BPEL-file. Usually it is deployed on BPEL-engine (e.g., ActiveBPEL) which runs on application server (e.g., Tomcat). Therefore, the location can be obtained via standard Java-resources, e.g. `getResource()` of `java.lang.ClassLoader`.
- `outfile` is a path to the resulting BPEL file which will contain a BPEL-code of a parallelized process.
- `restrictions_file` is a special file that contains parallelization restrictions. The latter can be stipulated by the business needs of the business process, and they are defined by a human (e.g., process designer or business analyst). Restriction defines the order of execution of activities even if they can be executed in parallel. In other words, execution of one activity **must** be executed before the other one. Namely, it is executed via synchronization links.

File of restrictions have the following syntax:

```

<restrictions>
  (<restriction name="NCName">
    <activity-source>CName</activity-source>
    <activity-target>CName</activity-target>
  </restriction>)+
</restrictions>

```

For example, if we consider B6 scenario, we may require to book treatment options (`updateAgenda` action) before booking mobility options (`bookMobilityOptions` action). This is required to avoid the situation when treatment option booking is failed for some reason, and thus we have to roll back a mobility booking as well. Since mobility booking can be seen as an optional feature of the process, we prefer to deal with it in a secondary turn. An example of such a requirement is given below:

```

<restrictions>
  (<restriction name="treatment-before-mobility">
    <activity-source>updateAgenda</activity-source>
    <activity-target>bookTreatmentOptions</activity-
target>
  </restriction>)+
</restrictions>

```

Where

- attribute `name` of tag `restriction` defines a name of a link that will be created to enforce a parallelization restriction;
- `activity-source` and `activity-target` specify, respectively, `activity1` that should precede `activity2`. Both activities (e.g., `invoke`, `receive`,

reply, assign) are identified by their BPEL names that **must** be defined in attribute name in the original BPEL file.

4.8.2 Infrastructure Adjustment

SLA@SOI version of Apache Tashi contains a set of functions for on the fly adaptations of the virtual machines. These functions are: adaptCpuShare, adaptMemoryLimit, adaptNetworkBandwidth and adaptDiskBandwidth. Infrastructure POC component triggers these functions when it gets the violation messages from the Infrastructure Monitoring. These actions are triggered via ISM calling UpdateComputeCommand inside [tashi_commands.py](#). In order to extend the virtual machines management functionality and in particular runtime adaptations of the virtual machines, the [VmControlInterface](#) has to be extended and new functions have to be implemented inside [qemu.py](#) or [xenpv.py](#), depends on which virtual machines manager you are using. For example adaptCpuShare function is given below:

```
def adaptCpuShare(self, vmId, share):
    status = 0
    if os.path.exists("/dev/cpu/tasks"):
        subprocess.call("echo %s | sudo tee
/dev/cpu/qemu%s/cpu.shares" % (share, vmId), shell=True)
        log.info("Cpu share %s given to the Cgroup
qemu%s" % (share, vmId))
    else:
        status = 1
        log.error("CPU Cgroup is not mounted")
    return status
```

4.9 Template construction

Domain specific SLAT talks and is synchronized with its neighbour SLAT(s). In order to set up a software SLAT, software service provider has to translate the corresponding metrics in software SLAT to be infrastructure requirements. For instance, how many VMs should be used for hosting requested software from customer? Therefore, software service provider will start the negotiation with infrastructure service provider for getting infrastructure environment for the software. In the end, there could be several dependencies with infrastructure SLA in one software SLA.

4.9.1 Software SLAT

The ORC Software SLAT (SSLAT) consists of different sections, the general SLAT information, the interface definition, and finally the AgreementTerms. More details on the structure of an SLAT can be found in [16]. As the complete SSLAT has more than 700 lines of XML tags, we use only small excerpts of each section for illustration. The complete SSLAT can be found in the file [ORC Software-SLAT.xml](#).

```
<slasoi:SLATemplate
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:slasoi="http://www.slaatsoi.eu/slamodel"
xsi:schemaLocation="http://www.slaatsoi.eu/slamodel
slasoi.xsd">
```

```

<slasoi:Text/>
<slasoi:Properties/>
<slasoi:UUID>ORC_SoftwareSLAT</slasoi:UUID>
<slasoi:ModelVersion>1</slasoi:ModelVersion>
<slasoi:Party>
  <slasoi:Text/>
  <slasoi:Properties>
    <slasoi:Entry>
      <slasoi:Key>http://www.slaatsoi.org/slamodel#gslam_epr
      </slasoi:Key>

<slasoi:Value>http://localhost:8080/services/SWNegotiation?wsd
1
  </slasoi:Value>
</slasoi:Entry>
</slasoi:Properties>
<slasoi:ID>ORCProvider</slasoi:ID>
<slasoi:Role>http://www.slaatsoi.org/slamodel#provider
</slasoi:Role>
</slasoi:Party>

```

The general information section of the SSLAT defines a UUID for this SSLAT, which is followed by a party tag. Some XML-tags are empty, as the SLA model requires their existence however their content is not required in the ORC scenario. The party tag includes a reference to the running instance of the Software SLAM. Additionally, the party has an ID and the role is set to provider.

The ORC provides several interfaces. For the sake of brevity, we now focus on the section describing the PaymentService. The description of other services follows the same schema. Again, empty tags are required by schema file, however they are not needed in the ORC scenario.

```

<slasoi:InterfaceDeclr>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:ID>ORCPaymentService</slasoi:ID>
  <slasoi:ProviderRef>ORCProvider</slasoi:ProviderRef>
  <slasoi:Interface>
    <slasoi:InterfaceSpec>
      <slasoi:Text></slasoi:Text>
      <slasoi:Properties></slasoi:Properties>
      <slasoi:Name>PaymentService</slasoi:Name>
      <slasoi:Operation>
        <slasoi:Text></slasoi:Text>
        <slasoi:Properties></slasoi:Properties>
        <slasoi:Name>PaymentServiceOperation</slasoi:Name>
        <slasoi:Input>
          <slasoi:Text></slasoi:Text>
          <slasoi:Properties></slasoi:Properties>

```

```

        <slasoi:Name>cardInformation</slasoi:Name>
        <slasoi:Auxiliary>>false</slasoi:Auxiliary>
    </slasoi:Input>
    <slasoi:Input>
        <slasoi:Text></slasoi:Text>
        <slasoi:Properties></slasoi:Properties>
        <slasoi:Name>cardNumber</slasoi:Name>
        <slasoi:Auxiliary>>false</slasoi:Auxiliary>
    </slasoi:Input>
    <slasoi:Output>
        <slasoi:Text></slasoi:Text>
        <slasoi:Properties></slasoi:Properties>
        <slasoi:Name>PaymentServiceResponse</slasoi:Name>
        <slasoi:Auxiliary>>false</slasoi:Auxiliary>
    </slasoi:Output>
</slasoi:Operation>
</slasoi:InterfaceSpec>
</slasoi:Interface>
</slasoi:InterfaceDeclar>

```

The interface declaration starts with an ID followed by a reference to the Provider of this service using the IDs defined in the general section of the SSLAT. Each interface includes an interface specification, which has a name and lists all operations included in this interface. Each operation has a name and some in- and output parameter. For each Parameter a name has to be set and it has to be defined if the parameter is auxiliary or not. More details can be found in [Ref to SLA model description].

The last and most important section of the SSLAT is the definition of Agreement terms. In the ORC scenario we use three different quality attributes. First the maximum load on the system that the customer is allowed to initiate on the system is limited. Based on this knowledge the provider can realize an appropriate sizing of the required infrastructure sizing. Additionally, the SSLAT includes AgreementTerms considering the completion time of a service and its reliability. In the following we again focus on the PaymentService. As the structure of the AgreementTerms is quite similar we list and describe only the customer constraint limiting the requests per second in this document.

In the ORC scenario we use variables to define ranges of selectable values. Each AgreementTerm can be split up into a VariableDeclar and a Guaranteed tag. The following listing only include the declaration of a variable, the specification of a Guaranteed tag is described later.

```

<slasoi:AgreementTerm>
    <slasoi:Text/>
    <slasoi:Properties/>
    <slasoi:ID>ORC_CustomerConstraintPayment</slasoi:ID>
    <slasoi:VariableDeclar>
        <slasoi:Text/>
        <slasoi:Properties/>
        <slasoi:Customisable>
            <slasoi:Var>Var_CustomerConstraintPayment</slasoi:Var>

```

```

    <slasoi:Value>
      <slasoi:Value>70</slasoi:Value>
      <slasoi:Datatype>http://www.slaatsoi.org/coremodel/
units#tx_per_s
    </slasoi:Datatype>
  </slasoi:Value>
  <slasoi:Expr>
    <slasoi:CompoundDomainExpr>
      <slasoi:Subexpression>
        <slasoi:SimpleDomainExpr>
          <slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#greater
_than</slasoi:ComparisonOp>
            <slasoi:Value>
              <slasoi:CONST>
                <slasoi:Value>50</slasoi:Value>
            </slasoi:Value>
          </slasoi:CONST>
        </slasoi:Value>
      </slasoi:SimpleDomainExpr>
    </slasoi:Subexpression>
    <slasoi:Subexpression>
      <slasoi:SimpleDomainExpr>
        <slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#less_th
an_or_equals</slasoi:ComparisonOp>
          <slasoi:Value>
            <slasoi:CONST>
              <slasoi:Value>100</slasoi:Value>
          </slasoi:Value>
        </slasoi:CONST>
      </slasoi:Value>
    </slasoi:SimpleDomainExpr>
  </slasoi:Subexpression>
</slasoi:LogicalOp>http://www.slaatsoi.org/coremodel#and</slaso
i:LogicalOp>
  </slasoi:CompoundDomainExpr>
</slasoi:Expr>
</slasoi:Customisable>
</slasoi:VariableDeclr>
<slasoi:Guaranteed>
...
</slasoi:Guaranteed>

```

```
</slasoi:AgreementTerm>
```

A VariableDeclr section includes a Customizable tag. In addition to the variable name (Var tag), the actually selected value (Value tag) including its unit, and an expression (Expr tag) is defined. With the expression the range of selectable values is defined. More details on the expression language and the different range definitions can be found in [16]. The interpretation of the variable definition listed above is:

var = 70 1/s; allowed value range 50 1/s < var <= 100 1/s

The definition of the variable is followed by the definition of the guaranteed state.

```
<slasoi:AgreementTerm>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:ID>ORC_CustomerConstraintPayment</slasoi:ID>
  <slasoi:VariableDeclr>
...
</slasoi:VariableDeclr>
<slasoi:Guaranteed>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:State>
    <slasoi:ID>ORCThroughputConstraintPayment</slasoi:ID>
    <slasoi:Priority xsi:nil="true" />
    <slasoi:Constraint>
      <slasoi:TypeConstraintExpr>
        <slasoi:Value>
          <slasoi:FuncExpr>
            <slasoi:Text>
            </slasoi:Text>
            <slasoi:Properties>
            </slasoi:Properties>

<slasoi:Operator>http://www.slaatsoi.org/commonTerms#arrival_r
ate</slasoi:Operator>
      <slasoi:Parameter>

<slasoi:ID>ORCPaymentService/PaymentServiceOperation
      </slasoi:ID>
      </slasoi:Parameter>
      </slasoi:FuncExpr>
    </slasoi:Value>
    <slasoi:Domain>
      <slasoi:SimpleDomainExpr>

<slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#less_th
an</slasoi:ComparisonOp>
      <slasoi:Value>
```

```

<slasoi:ID>Var_CustomerConstraintPayment</slasoi:ID>
    </slasoi:Value>
    </slasoi:SimpleDomainExpr>
    </slasoi:Domain>
    </slasoi:TypeConstraintExpr>
    </slasoi:Constraint>
  </slasoi:State>
</slasoi:Guaranteed>
</slasoi:AgreementTerm>

```

The specification of a state starts with an ID followed by a Priority tag. If priorities are not used, the tag must include `xsi:nil="true"` to allow a proper parsing of the XML file. In the ORC we only use type constraint expressions. Such an expression consists of a value that can be measured and a domain, which defines the guaranteed value range. We use the `http://www.slaatsoi.org/commonTerms-#arrival_rate` operation to measure the throughput respectively the invocations per second of the service operation with ID `ORCPaymentService/-PaymentServiceOperation`. In the domain, we specify, that the value should be less than the value defined within the variable `Var_CustomerConstraintPayment`.

The Interpretation of the SSLAT parts described above is:

"The customer is not allowed to invoke the operation provided by the PaymentService offered by the ORCProvider more than 70 times per second. The threshold value (actually 70 1/s) can be varied between 50 and 100 1/s"

4.9.2 Infrastructure SLAT

The ORC Infrastructure SLAT (ISLAT) consists of different sections, the general SLAT information, the interface definition, and finally the AgreementTerms. More details on the structure of an SLAT can be found on [SourceForge](#) [16]. As the complete ISLAT has more than 600 lines of XML tags, we use only small excerpts of each section for illustration. The complete SSLAT can be found in the file [ORCInfrastructure SLA template.xml](#).

In infrastructure SLAT, the following features can be specified:

- Involved parties, where the detailed information of service customer and service provider are defined. As below:

```

<slasoi:SLATemplate
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:slasoi="http://www.slaatsoi.eu/slamodel"
xsi:schemaLocation="http://www.slaatsoi.eu/slamodel
slasoi.xsd">
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:UUID>ORC_SoftwareSLAT</slasoi:UUID>
  <slasoi:ModelVersion>1</slasoi:ModelVersion>
  <slasoi:Party>
    <slasoi:Text/>

```

```

    <slasoi:Properties>
      <slasoi:Entry>
        <slasoi:Key>http://www.slaatsoi.org/slamodel#gslam_epr
        </slasoi:Key>

<slasoi:Value>http://localhost:8080/services/ISNegotiation?wsd
1
      </slasoi:Value>
    </slasoi:Entry>
  </slasoi:Properties>
  <slasoi:ID>ORCProvider</slasoi:ID>
  <slasoi:Role>http://www.slaatsoi.org/slamodel#provider
  </slasoi:Role>
</slasoi:Party>

```

- Functional properties, the detailed configuration of virtual machines (VMs), like the CPU core, speed, memory, hard disk, bandwidth, service image, the number of VMs and so on. We define these attributes into "VariableDeclr", which is customisable by both parties during the SLA negotiation. When all the "VariableDeclr" attributes are established and agreed, they will further be referenced by "Guaranteed".

```

<slasoi:VariableDeclr>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:Customisable>
  <slasoi:Var>VM_CORES_VAR</slasoi:Var>
  <slasoi:Value>
    <slasoi:Value>1</slasoi:Value>

<slasoi:Datatype>http://www.w3.org/2001/XMLSchema#integer
  </slasoi:Datatype>
  </slasoi:Value>
  <slasoi:Expr>
    <slasoi:CompoundDomainExpr>
      <slasoi:Subexpression>
        <slasoi:SimpleDomainExpr>
<slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#greater\_than</
slasoi:ComparisonOp>
          <slasoi:Value>
            <slasoi:CONST>
              <slasoi:Value>0</slasoi:Value>

<slasoi:Datatype>http://www.w3.org/2001/XMLSchema#integer</slasoi:Dat
atype>
          </slasoi:CONST>
          </slasoi:Value>
        </slasoi:SimpleDomainExpr>
      </slasoi:Subexpression>
    </slasoi:Subexpression>
  </slasoi:SimpleDomainExpr>

```

```

<slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#less_than_or_e
quals</slasoi:ComparisonOp>
    <slasoi:Value>
    <slasoi:CONST>
    <slasoi:Value>16</slasoi:Value>

<slasoi:Datatype>http://www.w3.org/2001/XMLSchema#integer</slasoi:Dat
atype>
    </slasoi:CONST>
    </slasoi:Value>
    </slasoi:SimpleDomainExpr>
    </slasoi:Subexpression>

<slasoi:LogicalOp>http://www.slaatsoi.org/coremodel#and</slasoi:Logic
alOp>
    </slasoi:CompoundDomainExpr>
    </slasoi:Expr>
    </slasoi:Customisable>
</slasoi:VariableDeclr>

<slasoi:Guaranteed>
    <slasoi:Text/>
    <slasoi:Properties/>
    <slasoi:State>
    <slasoi:ID>VM_CORES</slasoi:ID>
    <slasoi:Priority xsi:nil="true"/>
    <slasoi:Constraint>
    <slasoi:TypeConstraintExpr>
    <slasoi:Value>
    <slasoi:FuncExpr>
    <slasoi:Text/>
    <slasoi:Properties/>

<slasoi:Operator>http://www.slaatsoi.org/resources#vm_cores</slasoi:O
perator>
    <slasoi:Parameter>
    <slasoi:ID>VM_X</slasoi:ID>
    </slasoi:Parameter>
    </slasoi:FuncExpr>
    </slasoi:Value>
    <slasoi:Domain>
    <slasoi:SimpleDomainExpr>

<slasoi:ComparisonOp>http://www.slaatsoi.org/coremodel#equals</slasoi
:ComparisonOp>
    <slasoi:Value>
    <slasoi:ID>VM_CORES_VAR</slasoi:ID>
    </slasoi:Value>
    </slasoi:SimpleDomainExpr>
    </slasoi:Domain>

```

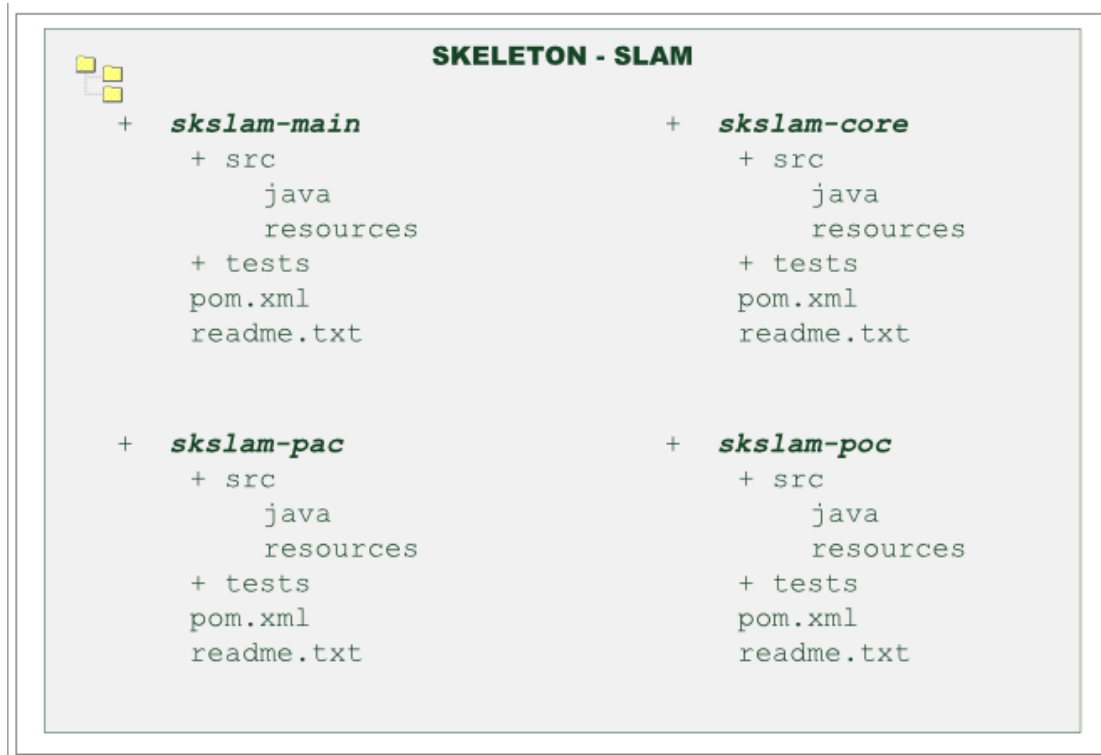
```
</slasoi:TypeConstraintExpr>
  </slasoi:Constraint>
</slasoi:State>
</slasoi:Guaranteed>
Quality of Service (QoS) terms, like availability,
reliability, pricing, penalties. The definition of QoS terms
and references are the same as Functional properties.
```

5 Extending Skeleton SLAM and the SLA@SOI Studio

5.1 Skeleton SLAM

One of the most important features of G-SLAM is its reusability by new SLA managers with minimal effort during implementation. The Skeleton SLAM aims to define 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. Note that this design allows the independence across components within the SLAM and enables the plugging-in and plugging-off of new entities.



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 following figure:

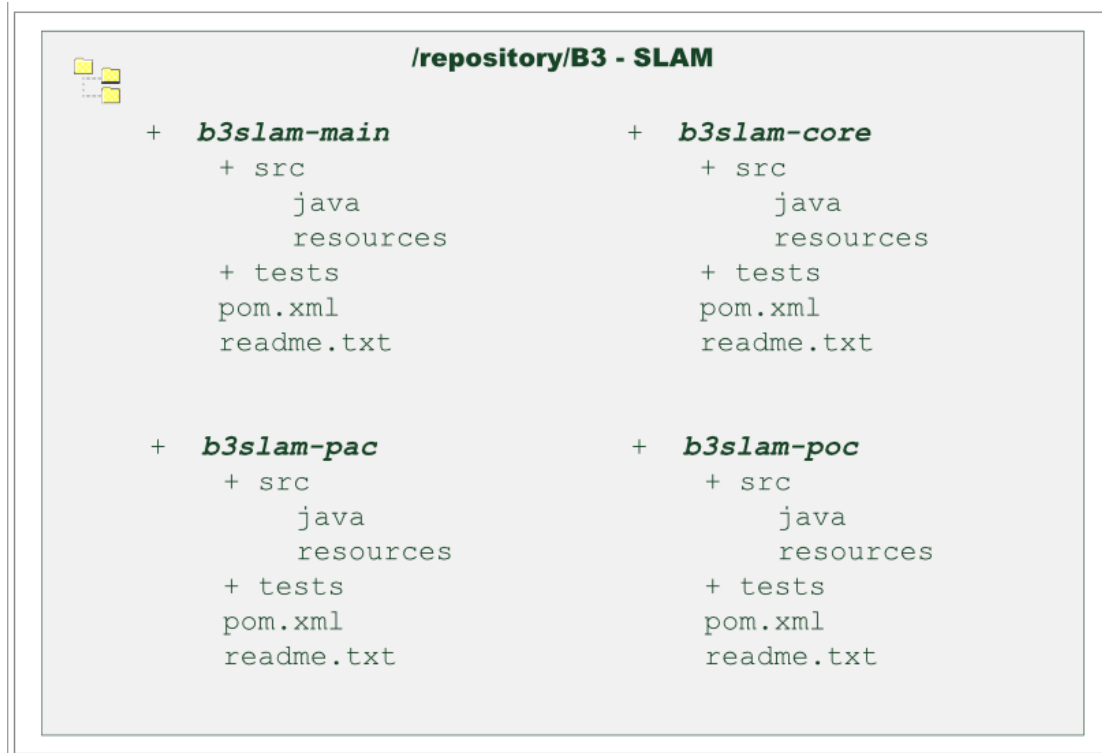
```

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

mvn %PLUGIN%:%PARAM0% %PARAM1% %PARAM2% %PARAM3%

```

After running this script the domain specific SLAM will contain the following directories:



Now the sub-modules `b3slam-pac` and `b3slam-poc` will contain the implementation of the domain specific SLAM. 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`.

5.2 SLA@SOI Studio

The SLA@SOI Studio aims to help to adapt and configure the SLA@SOI platform for new use cases. The Studio adds a new project type to Eclipse, named *SLA@SOI Deployment Project*. This is the placeholder for most of the configuration files and settings of the platform as well as for the use-case specific files, such as the Service Construction Model, SLA templates for service dependencies etc. GUI editors for most of these files are included with the Studio.

The Studio can also *run* such as project, i.e. start the platform in pax runner using the above mentioned configuration files and settings. The effect of different settings, changes in your custom SLAM code etc can thus be quickly tested live.

For more detailed description of the features and instructions to use the Studio, please refer to [9].

5.3 SLA Manager Development with Studio

The Studio also aims to help with development of your custom SLA managers. It contains a wizard front-end to the maven plugin for skeleton SLAM generation and also has the option to automatically import the modules of your new SLAM into Eclipse.

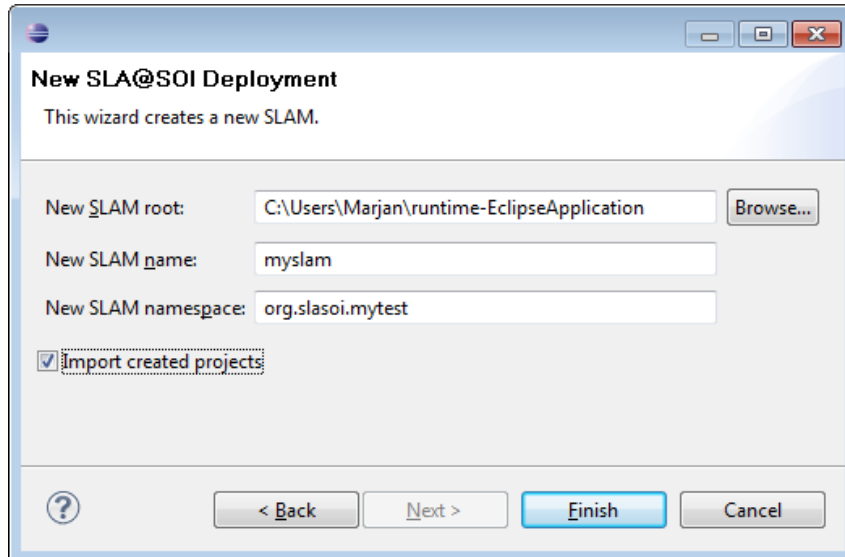


Figure 21: Studio wizard for generation of a new (skeleton) SLAM

5.4 Filling-in the Skeleton SLAM for a Specific Use-Case

Here we describe how to adapt/extend/fill-in the skeleton SLAM that you have just generated in order to get a working SLAM for a specific use-case. As the different kinds of SLAMs and their relationships have already been described above, our description will be simplified to a single SLAM in order to make it shorter and easier to grasp.

5.4.1 Implementing POC

The POC is in charge of negotiating SLATs and, if a SLAT acceptable to all the parties is found, creating an agreement, registering the SLA, and finally devising a plan that will be used by the PAC to provision the service. All but the latter step are done by implementing the corresponding methods of the inner class *DomainSpecAssessmentAndCustomize*, the template of which is already present within the *PlanningOptimizationImpl* class of your POC. The methods that need to be implemented are:

- *SLATemplate[] negotiate(String id, SLATemplate template)*

This method can either:

- return an array containing just the given SLAT, thereby accepting it as it is,
 - return an empty array, thereby rejecting the given SLAT, or
 - return an array of one or more alternatives, e.g. modified copies of the given SLAT.
- *SLA createAgreement(String arg0, SLATemplate arg1)*

This method should create a SLA from the given SLAT (that is, set the properties *UUID*, *effectiveFrom*, *effectiveUntil*, and *agreedAt*) and register it in the SLA registry. If the registration is done with the state *SLAState.OBSERVED*, provisioning of a service instance will be automatically triggered by the framework.

- *boolean terminate(UUID arg0, List<TerminationReason> arg1)*

This method should either reject the termination (returning false) or accept it (returning true). In the latter case it must of course also de-register the SLA and trigger tearing down the provisioned service instance.

Devising the plan to provision a service instance is done by writing a new class that implements the *INotification* interface (*org.slasoi.gslam.core.poc.PlanningOptimization.INotification*), which contains just the method

```
void activate(final SLA sla)
```

This method must prepare a provisioning plan (*org.slasoi.gslam.commons.plan.Plan*) and finally call the PAC's method *executePlan* on it.

5.4.2 Implementing PAC

At the very least, you need to implement the method *ProvisioningAdjustmentImpl.executePlan(Plan plan)*, which must execute all the steps of the provided service instance provisioning plan. Once you have implemented and tested this provisioning, you can proceed with implementing cancellation and modification of the plan.

5.4.3 Workarounds for Common Problems

In order for the Drools library to start correctly, we must ensure that the PAC implementation exists in the context of the PAC bundle rather than that of the main bundle. This can be done by creating the *ProvisioningAdjustmentImpl* object in the constructor of the class *ProvisioningAdjustmentServices*, storing the reference and then merely returning it in the *create()* method. Also, add the following to the *<instructions>* tag of POC's and main's *pom.xml* files:

```
<Require-Bundle>${slam.required.bundles}, drools4osgi,
gslam-protocol-engine</Require-Bundle>
```

If your SLAM will not use Drools then you can skip this modification.

If you encounter problems with the *configurationFile* property in *pac-context.xml*, you can simply remove this property.

To ensure that GSLAM can find syntax converter, which it depends on, add it to *pom.xml* of the POC project.

Please ensure that the class *SkeletonSLAMBean* in the main bundle is exposed as a bundle activator. This can be done by adding the following to the *main-context.xml* file:

```
<bean id="orc-mockupBean"
class="org.slasoi.orcmockup.main.beans.SkeletonSLAMBean"
init-method="start"
destroy-method="stop" />
```

References

- [1] Open Reference Demonstrator source code. URL: <https://sla-at-soi.svn.sourceforge.net/svnroot/sla-at-soi/ord/gui/trunk>
- [2] Apache Axis 2. URL: <http://axis.apache.org/axis2/java/core/>
- [3] SLA@SOI integration tests documentation. URL: <http://sourceforge.net/apps/trac/sla-at-soi/wiki/DocumentationIntegrationTest>
- [4] SLA@SOI Source Forge project. URL: <http://sourceforge.net/apps/trac/sla-at-soi/wiki>
- [5] Open Reference Case source code. URL: <https://sla-at-soi.svn.sourceforge.net/svnroot/sla-at-soi/ord/orc/trunk/>
- [6] SLA@SOI ORC Installation Guide. URL: http://sourceforge.net/apps/trac/sla-at-soi/browser/ord/doc/ORC_Deployment_Guide.pdf
- [7] Infrastructure POC. URL: <http://sourceforge.net/apps/trac/sla-at-soi/wiki/InfrastructurePoc>
- [8] SLA@SOI Deliverable DB2.a
- [9] SLA@SOI Studio User Guide. URL: http://sourceforge.net/apps/trac/sla-at-soi/browser/platform/trunk/doc/Studio-User_Guide.doc
- [10] Open Cloud Computing Interface. URL: <http://www.occi-wg.org/>
- [11] OCCI HTTP rendering. URL: <http://ogf.org/documents/GFD.185.pdf>
- [12] Infrastructure Service Manager. URL: <https://sourceforge.net/apps/trac/sla-at-soi/wiki/InfrastructureServiceManager>
- [13] Apache Tashi. URL: <http://incubator.apache.org/tashi/>
- [14] Service Construction Model Editor. URL: <http://ftp-slasoi.xlab.si/scmEditor.zip>
- [15] Monitoring Manager. URL: <http://sourceforge.net/apps/trac/sla-at-soi/wiki/MonitoringManager>
- [16] SLA Model. URL: <http://sourceforge.net/apps/trac/sla-at-soi/wiki/SlaModel>

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 asterisk *.

Agreement Initiator	An agreement initiator is a party to a <i>service level agreement</i> . 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 <i>agreement initiator</i> to <i>agreement responder</i> during agreement creation, indicating the relationship which the initiator would like to form.
Agreement Responder	The agreement responder is a party to a <i>service level agreement</i> . 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 <i>agreement responder</i> to advertise the types of offers it is willing to accept.
Agreement Term	Agreement terms define the content of a <i>service level agreement</i> .
Business Service	A business service is exposed/invoked via at least some non IT elements.
Business Manager	A specialization of <i>service provider</i> : 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 <i>service provider</i> : 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 <i>infrastructure provider</i> : 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 <i>infrastructure services</i> .

Infrastructure Service	An infrastructure service is a specific <i>IT service</i> which exposes resource/hardware-centric capabilities.
Internal Service	Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.
IT Service	An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.
Offered Service	An abstract service (more precisely: service type) which is offered by a specific <i>Service Provider</i> to its <i>Service Customers</i> .
Operation Level Agreements	A specification of the conditions under which an <i>internal service</i> or a component is to be used by its "customer".
Service	A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. See also <i>service interface type, service concreteness, service exposure</i>
Service Concreteness	The stage a service reaches over time from a fully abstract type to actually instantiated. See also <i>service type, offered service, service implementation, service instance</i>
Service Consumer	Person(s) who actually consume/use the provided services. Typically they belong to the <i>service customer</i> .
Service Customer	Someone (person or group) who orders/buys services and defines and agrees the service level targets.
Service Description Term	Service Description Terms describe the functionality that will be delivered under the <i>service level agreement</i> . The agreement description may include also other non-functional items referring to the service description terms.
Service Exposure	Services can be exposed either internally (within the same administrative domain) or externally. See also <i>internal service, external service</i>
Service Implementation	A service implementation is a possible concrete realization of a given <i>service type</i> .
Service Instance	A concrete realization of an <i>offered service</i> which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given <i>service implementation</i> .
Service Interface Type	Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface. See also <i>business service, IT service, hybrid service</i>
Service Level Consequence	An action that takes place in the event that a service level objective is not met.
Service Level Agreement	An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or

	availability. Entities can dynamically establish and manage agreements via Web service interfaces.
Service Level Objective	Service Level Objective represents the quality of service aspect of the <i>agreement</i> . Syntactically, it is an assertion over the agreement <i>terms</i> of the agreement as well as such qualities as date and time.
Service Provider	An organization supplying services to one or more internal customers or external customers.
SLA Manager	A specialization of <i>service provider</i> : person/system that is responsible for managing SLATs and SLA relationships.
Software Designer	A specialization of <i>software provider</i> : person that designs/develops the architecture and components of a specific SLA based application.
Software Manager	A specialization of <i>service provider</i> : person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.
Software Provider	An organization producing <i>software components</i> which might be used by a <i>service provider</i> to assemble actual <i>services</i> .
Software Service	A software service is a specific <i>IT service</i> which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.
Software Component	Software components are the entities produced at design-time by a <i>software provider</i> .
Service Type	A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.

Appendix B: Abbreviations

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