contributing to NESSI

Empowering the Service Economy with SLA-aware Infrastructures

European Commission
Information Society and Media

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

Deliverable D.B2a

Adhoc demonstrator

Keywords:
Web Service, Open Reference Case, CoCoME, Adhoc Demonstrator, Service Level Agreement, Service-Oriented Infrastructure

Due date of deliverable: 31st May 2009
Actual submission to EC date: 30th June 2009

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

Lead contractor for this deliverable: XLAB
Revision: V.0.11 (24th June 2009)

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
PU	Public	Yes

Document Status	
Deliverable Lead	Gregor Berginc, XLAB
Reviewer 1	Costas Kotsokalis, UDO
Reviewer 2	Paolo Zampognaro, ENG
PMT Reviewer	Wolfgang Theilmann, SAP
Complete version submitted to reviewers	20th April, 2009
Comments of reviewer 1 received	29th April, 2009
Comments of reviewer 2 received	30th April, 2009
Revised version submitted to reviewers and PCC	14th May, 2009
Final approval from reviewer 1 received	20th May, 2009
Final approval from reviewer 2 received	20th May, 2009
Deliverable submitted to PMT	25th May, 2009
PMT Approval	8th June, 2009

Contributors	
Partner	Contributors
XLAB	Miha Stopar, Žiga Mlinar
FZI	Christoph Rathfelder, Christof Momm

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

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

Document History			
Version	Date	Author	Changes
V0.1	03/31/2009	Gregor Berginc (XLAB)	Initial structure of the document.
V0.2	04/15/2009	Christoph Rathfelder (FZI)	ORC specifications and deployment guide appendix.
V0.3	04/16/2009	Miha Stopar, Gregor Berginc (XLAB)	Integrated GUI sections with the main deliverable text.
V0.4	04/16/2009	Žiga Mlinar, Gregor Berginc (XLAB)	Simulation environment.
V0.5	04/19/2009	Christoph Rathfelder (FZI), Gregor Berginc (XLAB)	Modified CoCoME and ORC description, integrated int the main document.
V0.6	04/20/2009	Christoph Rathfelder (FZI),Gregor Berginc (XLAB)	Ready for first round of internal reviews.

V0.7	5/25/2009	Miha Stopar	Integrated comments from internal review.
V0.8	6/15/2009	Gregor Berginc	Extended the summary and introduction section.
V0.9	6/15/2009	Christoph Rathfelder (FZI)	Updated the evaluation results section.
V0.10	6/22/2009	Gregor Berginc, Christoph Rathfelder, Miha Stopar	Extended executive summary and intro sections as suggested by PMT review.
V0.11	6/24/2009	Miha Stopar, Christoph Rathfelder	Conclusions added. Final cross-check.

Executive Summary

The integrated European research project SLA@SOI aims at supporting service-oriented infrastructures with support for holistic management of services according to service level agreements (SLAs). The project researches for an SLA management framework that can be applied to a large variety of use cases. Within the first project year, the project developed an initial SLA management framework, the so-called adhoc framework.

This document presents the first demonstrator of SLA@SOI that applies the adhoc SLA management framework for an SLA-driven management of a service-oriented application. The document provides details on the actual service-oriented reference application, the way how the SLA management framework has been applied and how both together have been integrated and evaluated within the overarching adhoc demonstrator. The demonstrator is both public and open-source and thus available to the widest possible range of stakeholders, interested in the implementation of a Service Level Agreement aware Service Oriented Infrastructure. These include but are not limited to service providers, scientific community and developers.

The open reference case constitutes the service-oriented reference application in the context of the adhoc demonstrator. It provides software functionality for supporting the sales process in a retail-chain. Within the demonstrator, the open reference case is considered as a software package that is made available to customers following a software-as-a-service model. A couple of enhancements were done in order to turn the original retail application into a service-oriented one which suits the envisaged on-demand model. First, original functional interfaces were turned into Web Services. Then various deployment options and levels of separations have been realized in order to introduce flexibility for the actual software provisioning. Last, several software hooks that facilitated run-time monitoring and SLA-awareness of the application have been added as necessary glue to the SLA-management framework.

The adhoc demonstrator itself allows for simulating customers who are interested in consuming the retail application. To that extent it covers both, the SLA-related negotiation and management activities taking place between customers and a service provider but also the simulation of the actual usage of the service-oriented application, i.e. the retail functionality. Furthermore, the demonstrator provides a rich view on all aspects of the SLA-management and the managed application, namely customer's perspective, service provider's perspective and developer's perspective. Customer's view allows browsing a directory of service offerings along with their Service Level Agreement Templates, and negotiating the agreement. Hierarchical views of Service Level Agreements enable the service provider to monitor the state of each and every service used to satisfy customer's needs. An additional panel provides a simulation environment where various usage profiles (simulating customer activity) can be experimented with. Developers are also presented with a specialised logging panel depicting detailed information about the status of all framework components.

The evaluation of the SLA management framework in the context of the adhoc demonstrator is done along two dimensions. First, there is the feasibility evaluation that shows as a proof of concept if and how the SLA management framework can be applied to a service-oriented application. Second, there is a more structured qualitative and quantitative evaluation which follows the well established Goal/Question/Metric (GQM) approach.

The adhoc demonstrator was successfully presented at the Collaboration meeting for FP6 and FP7 projects that was held as part of the Internet of Services 2009 meeting in Brussels.

Table of Contents

1 Introduction 11
1.1 Document structure 12
2 Open Reference Case Specification 13
2.1 Starting Point: The Common Component Modelling Example (CoCoME) 13
2.2 Adaption and Extension: Service oriented Open Reference Case 14
2.2.1 Business Process 17
2.2.2 Service-oriented "Retail Chain" Scenario & Architecture 18
2.2.3 Web Services 21
2.3 Specification of deployment options 24
2.4 Integration of the ORC with the SLA framework 25
2.4.1 Manageablity Configuration and Instrumentation of the ORC 25
2.4.2 ORC Prediction Model 28
2.4.3 SLA Hierarchy 28
3 Adhoc Demonstrator 30
3.1 Simulation Environment 30
3.1.1 Introduction 30
3.1.2 The sales process 32
3.1.3 Configuration options 34
3.2 Graphical User Interface 36
3.2.1 Customer creation view 38
3.2.2 Service provider view (SLA hierarchy and SLA violation visualisation) 45
3.2.3 Workload manager console 48
3.2.4 Framework logging 53
4 Architecture of the Adhoc Demonstrator 55
5 Evaluation plan 61
5.1 The Goal/Question/Metric Appraoch 61
5.2 GQM-based Evaluation Plan 62
6 Results of evaluation (FZI/XLAB) 67
6.1 ORC Service lifecycle supported by the SLA framework 67
6.1.1 Support to offer new or modified software components 67
6.1.2 Support for provisioning the ORC to a new customer 68
6.1.3 Support in offering, selecting and negotiating SLAs. 69
6.1.4 Support in operating the ORC offered by the framework 70
6.1.5 Support for the infrastructure provider 71
6.2 Extensibility of the Demonstrator 72
7 Conclusions 77
7.1 Summary 77
7.1.1 CoCoME Adaptation 77
7.1.2 Usage Simulation Environment 77
7.1.3 Adhoc Demonstrator Graphical User Interface 77
7.1.4 Evaluation 77
7.2 Outlook on Future Work 78
7.2.1 ORC Development 78
7.2.2 Usage Simulation Environment Development 78
7.2.3 Adhoc Demonstrator Graphical User Interface Development 78
7.2.4 Evaluation Plan 78
References 79
Appendix A : Glossary 80
Appendix B : Abbreviations 81
Appendix C : ORC Deployment Guide 82
C.1 Short Description 82
C.2 Prerequisite 82

C.3 Architecture Overview 82
C.4 Deployment Procedure 83
C.5 Automated Starting of the ORC 84
C.6 Testing the ORC 85
C.6.1 PaymentService 86
C.6.2 InventoryService 87
Appendix D : Simulation Environment Implementation 88
D.1 WorkloadGenerator.java 88
D.2 AdhocJobManager.java 89
D.2.1 XOSDCONSOLE Example 89
D.3 Agent.java 89
D.4 Generating Java code from WSDL 90
D.4.1 JAX-WS 90
D.4.2 Axis 90
Appendix E : Adhoc Demonstrator Development Requirements 92

Table of Figures

Figure 1: Common Component Modelling Example (CoCoME) components........ 14
Figure 2: Open Reference Case Stakeholder. 15
Figure 3: Service-oriented Open Reference Case 15
Figure 4: Provisioning lifecycle. 16
Figure 5: Provisioning lifecycle – continued... 17
Figure 6: Sales Process. 18
Figure 7: Current status of implemented system. 19
Figure 8: Layered architecture. 20
Figure 9: CoCoME component application. 21
Figure 10: Inventory Service Interface... 21
Figure 11: Store Information Service Interface. 22
Figure 12: Order Service Interface. 22
Figure 13: Payment Debit Service Interface... 22
Figure 14: Card Validation Service Interface. 23
Figure 15: Payment Service Composition. 23
Figure 16: PaymentService BPEL Process. 24
Figure 17: ORC deployment options... 25
Figure 18: Overview to Manageable ORC 26
Figure 19: SLA Hierarchy 29
Figure 20: Schematic overview of the adhoc demonstrator. 30
Figure 21: The architecture of the simulation environment. 31
Figure 22: The sales process workflow. Normal service calls are displayed in light blue. Dark blue rectangles are used for delays. Decision symbols are depicted in orange colour with the name of the probability used. 33
Figure 23: Navigation between tabs in adhoc demonstrator. 37
Figure 24: Basic information about adhoc demonstrator. 37
Figure 25: Adhoc demonstrator window management. 38
Figure 26: Creating new customer. 39
Figure 27: Product diagram. 40
Figure 28: Customer view. 40
Figure 29: Service view. 41
Figure 30: Product view. 42
Figure 31: Operation view. 43
Figure 32: SLAT Browser. 45
Figure 33: Retail Chain Sales Process SLA Hierarchy. 46
Figure 34: SLA Hierarchy view. 47
Figure 35: Connection settings for communication over message bus. 48
Figure 36: Connection settings for communication between adhoc demonstrator and Simulation Environment over message bus. 49
Figure 37: Simulation configuration. 50
Figure 38: Simulation panel after workload generator configuration................. 52
Figure 39: Cash desk control view. 52
Figure 40: Framework Logging view. 53
Figure 41: Connection settings for communication over message bus. 54
Figure 42: The component diagram showing the adhoc architecture. 57
Figure 43: Sequence diagram during the negotiation phase. 58
Figure 44: Sequence diagram of the provisioning phase. 58
Figure 45: Sequence diagram representing details of the monitoring and the adjustment component. 60
Figure 46: Relations between goals, questions, and metrics........................... 61
Figure 47: Deployment Options of ORC Demo on CentOS. 82
Figure 48: Installed Services 85
Figure 49: PaymentService deployed in ActiveBPEL. 86

Figure 50: Test PaymentService in SOAPUI. 87

List of Tables

Table 1: An example of the JSON formatted workload configuration. 34
Table 2: Description of global configuration properties. 35
Table 3: Agent configuration properties.. 35
Table 4: Workflow properties defining probabilities of various branching conditions. Within the workflow, these properties are used for more realistic simulation of the sales process. Values in tables only denote extreme values, values between minimum and maximum are also applicable (e.g., a probability of 0.5 specifies equal chance of both branches in the workflow)......................... 36
Table 5: Description of workflow delays. Each delay is described in terms of the normal distribution, thus there are two arrays used (delay and randomStandardFactor). 36
Table 6: Pre-defined template values... 43
Table 7: Formulas for operations parameters... 43
Table 8: Goals, Questions, and Metrics. 62
Table 9: Support to offer new or modified software 67
Table 10: Support for provisioning the ORC to a new customer 68
Table 11: Support in offering, selecting and negotiating SLAs 69
Table 12: Support in operating the ORC 70
Table 13: Support for the infrastructure provider 71
Table 14: Extensibility of the ORC 72
Table 15: Main classes and packages constituting the adhoc Simulation Environment.. 88
Table 16: Example of manager.properties configuration. 88
Table 17: Example of a function, annotated with the XOSDCONSOLE annotation. 89
Table 18: JAX-WS pom.xml configuration.. 90
Table 19: Axis pom.xml configuration. 90

1 Introduction

The purpose of the work package B2 of the European Research project SLA@SOI is to provide a reference service-oriented application (Open Reference Case) serving as a starting point for conducting experiments with the integrated SLA management framework. Because the results of this work package are provided as open-source software libraries and tools it is also an important dissemination facility demonstrating the results of the entire project. Opposed to the remaining industrial use cases, the Open Reference Case comes with well known public specifications allowing everyone to understand both the application and results.

Public deliverable D.B2a documents the first public demonstrator, called the adhoc demonstrator and main results achieved within the first year of the project. The adhoc demonstrator comprises three major contributions, namely a service-oriented application, a simulation environment and a graphical frontend.

The service-oriented application (Open Reference Case) is based on a well known Common Component Modelling Example (CoCoME) [1], a legacy application supporting the sales process in a retail chain. For the purposes of the adhoc demonstrator, the application was first extended with additional Web Service layer on top of CoCoME components. Diverse set of deployment options was achieved by complete separation of the application logic from the data model resulting in the following two options:

- all-in-one package consisting of the application logic and the model in a single package,
- separation of the application logic from the model, resulting in two packages, and

Although not tested with the adhoc demonstrator, a separation of composed from atomic services into different virtual machines is also possible.

Finally, the application was transformed into a manageable and SLA-aware application ready to be deployed with the SLA@SOI framework. This step included an instrumentation of services that monitors the behaviour of service invocations and a design-time prediction model used during the negotiation and resource planning phase.

The purpose of the simulation environment is to simulate a workflow of a real-world sales process, thus generating a workload on a deployed application. It is used to showcase the behaviour of the SLA@SOI framework under different conditions. The workload is configured with an usage profile described in terms of number of concurrent cashbox and patterns practiced by store's customers. The latter consists of an expected number of purchased items, delays simulating human actions, and probabilities of certain actions within the workflow.

The demonstrator's graphical user interface (GUI) facilitates execution of experiments by providing different views on the overall system allowing stakeholders like customers, service providers, and developers to concentrate on their role in the system only. It allows users to browse through the product catalogues, gather information about respective Service Level Agreement Templates, suggested values and their relation to the price of the offer. It further enables customisation of SLA parameters and sending of an SLA offer to appropriate service provider.

Service provider is equipped with a view on the overall SLA hierarchy for individual customers. Starting with the acceptance of the SLA offer sent by the customer the hierarchy is both visualised and described for better understanding of the current status. During run-time, the service provider is immediately informed of any SLA violation and corrective actions. Additional panel can be used to interact with the simulation environment.

Researchers and developers, interested in understanding the core of the framework are presented with additional logging panel that displays detailed information about the status of the framework, actions that took place during execution, and possible errors. The panel supports rich and fine-granular filtering options.

Evaluation of the adhoc demonstrator and, in particular the first version of the SLA@SOI framework, is based on a well-known Goal/ Question/Metric approach. We developed a series of goals aligned with requirements from business, service provider and developer perspective. To evaluate goals, further questions and metrics were defined.

1.1 Document structure

Section 2 introduces service oriented retail chain solution supporting the sales process. Part of the section also describes various deployment options and modifications required for the integration with the SLA management framework.

Third section is dedicated to the adhoc simulation environment and the graphical user interface allowing interested stakeholders to conduct experiments. The latter is described from different perspectives.

The adhoc architecture of the initial version of the SLA@SOI framework, including interactions between the framework, the adhoc demonstrator and the Open Reference Case application are described in Section 4.

Evaluation plan prepared for the analysis of the initial version of the SLA@SOI framework is presented in Section 5, while Section 6 describes preliminary results of the evaluation.

The deliverable concludes with a list of known limitations and directions for future work, in particular toward the reference demonstrator.

Finally, appendices to this document present additional, mostly technical, information about the implementation and deployment of Open Reference Case and adhoc demonstrator.

2 Open Reference Case Specification

This section presents the Open Reference Case (ORC) developed in WP B2. The purpose of having an open-source application besides other industrial use cases is to have a reference application with a well-known and extensible specification that can be tailored to specific needs and requirements of different industrial use cases (service providers). As it is distributed under flexible open-source licence it is available for everybody to download and experiment with.

ORC is based on an existing component-based solution realizing a point of sale in a supermarket. It transforms it into a service-oriented application by providing corresponding web service interfaces as well as various deployment options, enabling separation of different components. ORC is thus ready to be delivered as Software-as-a-Service (SaaS) in a multi-customer environment.

The ORC is integrated into the SLA framework to perform an early feasibility study of the framework. Furthermore; the ORC scenario forms the basis for a number of requirements on the SLA framework. The evaluation of the adhoc demonstrator considering these requierments is described in section 6.

We first describe the component-based Common Component Modelling Example (CoCoME)[1] use case, which was the starting point for the development of the ORC. Second, we present the required adaptations and extensions of CoCoME to the projects needs.

2.1 Starting Point: The Common Component Modelling Example (CoCoME)

The ORC is a modification of the CoCoME example. CoCoME 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. Furthermore, the trading system deals with ordering goods from wholesalers, and generating various kinds of reports. It is used as Open Reference Use-Case in the SLA@SOI Project.

[1] Modelling Contest: Common Component Modelling Example (CoCoME) - GI-Dagstuhl Research Seminar, http://www.cocome.org, 2006. Last called on 18.07.2008.

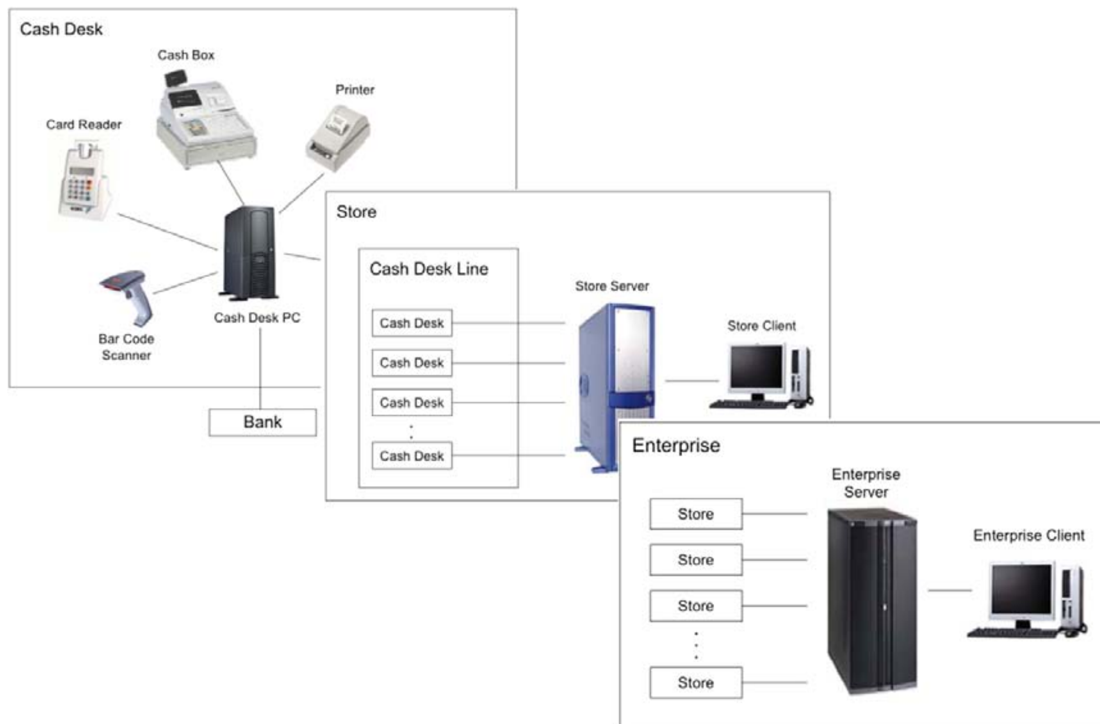

Figure 1: Common Component Modelling Example (CoCoME) components.

Figure 1 depicts the different components of the CoCoME scenario. An `Enterprise` consists of several `Stores`. Each `Enterprise` has an `Enterprise Server` to which all `Stores` are connected and an `Enterprise Client` enabling the Enterprise Manager to generate several kinds of reports. A retail store operates a certain number of `Cash Desks` in order to support and realise the sales process. Each `Store` has its own central `Store Server` connected to each `Cash Desk` of the Store as well as to the Enterprise Server. The `Cash Desk` is the place where the cashier scans the goods the customer wants to buy and where the paying (either by credit card or cash) is executed. A number of hardware components are associated with a single `Cash Desk` (Cash Box, Bar Code Scanner, ...). The central unit of each `Cash Desk` is the `Cash Desk PC` which wires all other components with each other and calls the services provided by the retail solution provider.

2.2 Adaption and Extension: Service oriented Open Reference Case

The adapted example describes a service-oriented retail solution which can be used in a trading system as it can be observed in a supermarket handling sales. It includes IT support for retail chains in general, covering enterprise headquarter (central management issues), stores (local management) and cash desks. Considering the approach of the SLA@SOI project we assume the following slightly adapted scenario:
Several supermarkets are connected to a single service provider, 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. This service provider is connected to several external providers such as bank, wholesale centre, CRM supplier etc. The services run on top of an IT infrastructure offered by a further provider. Involved stakeholders and their bindings are shown in following Figure 2:

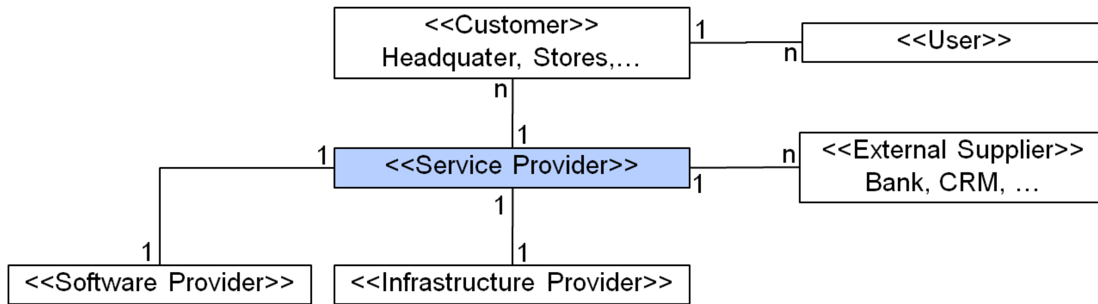

Figure 2: Open Reference Case Stakeholder.

The Service Provider manages the inventory and the application logic of the corresponding Store respectively of the corresponding Enterprise. If the Software Provider can't implement all services by itself, the Service Provider is able to access external services, as shown by the link between Service Provider and Bank in Figure 3.

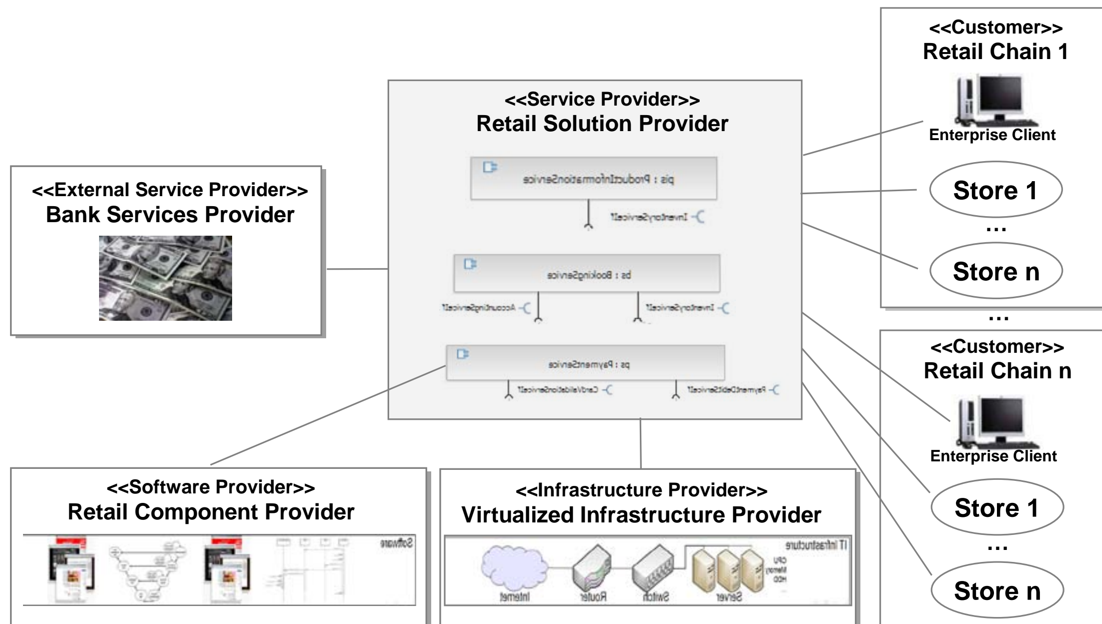

Figure 3: Service-oriented Open Reference Case

The service lifecycle intended by the ORC is depicted in Figure 4 and Figure 5 and comprehends the following stages:

1. Software Provider develops components and specifies non-functional requirements.
2. Infrastructure Provider specifies the offered infrastructure options and parameters.
3. Service Provider analyses SW components, possible deployments and service offerings based on offered Infrastructure.
4. Customer defines service requirements.
5. Service Provider plans possible service offerings.
6. Service Provider and Client negotiate agreement offerings.
7. Infrastructure Provider and Service Provider allocate physical resource and instantiate the offered services (including the setup of monitoring and accounting environment).

8. Service Provider operates the service for the Customer and collects usage information.
9. Service Provider performs monitoring, prediction and service adjustments.

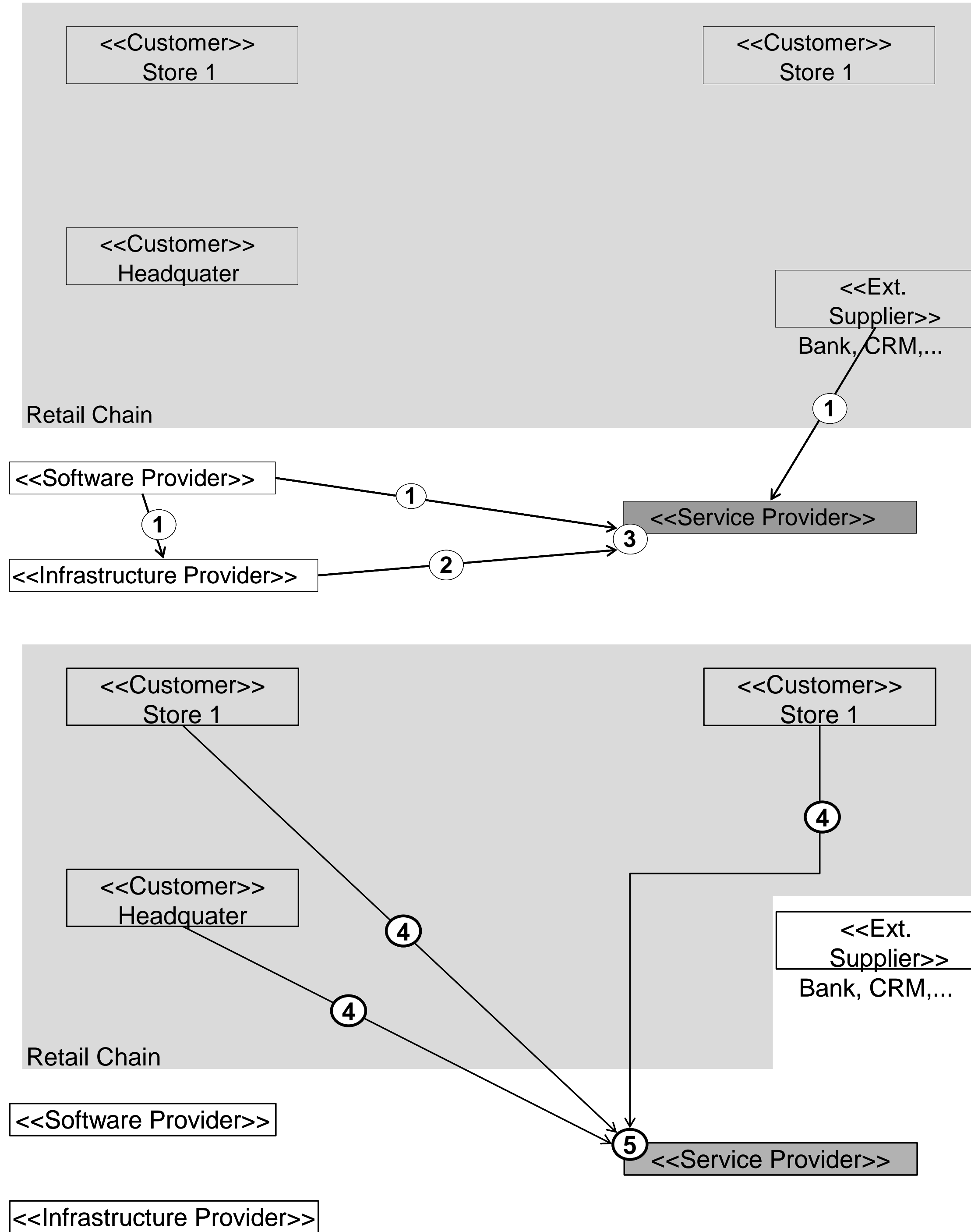

Figure 4: Provisioning lifecycle.

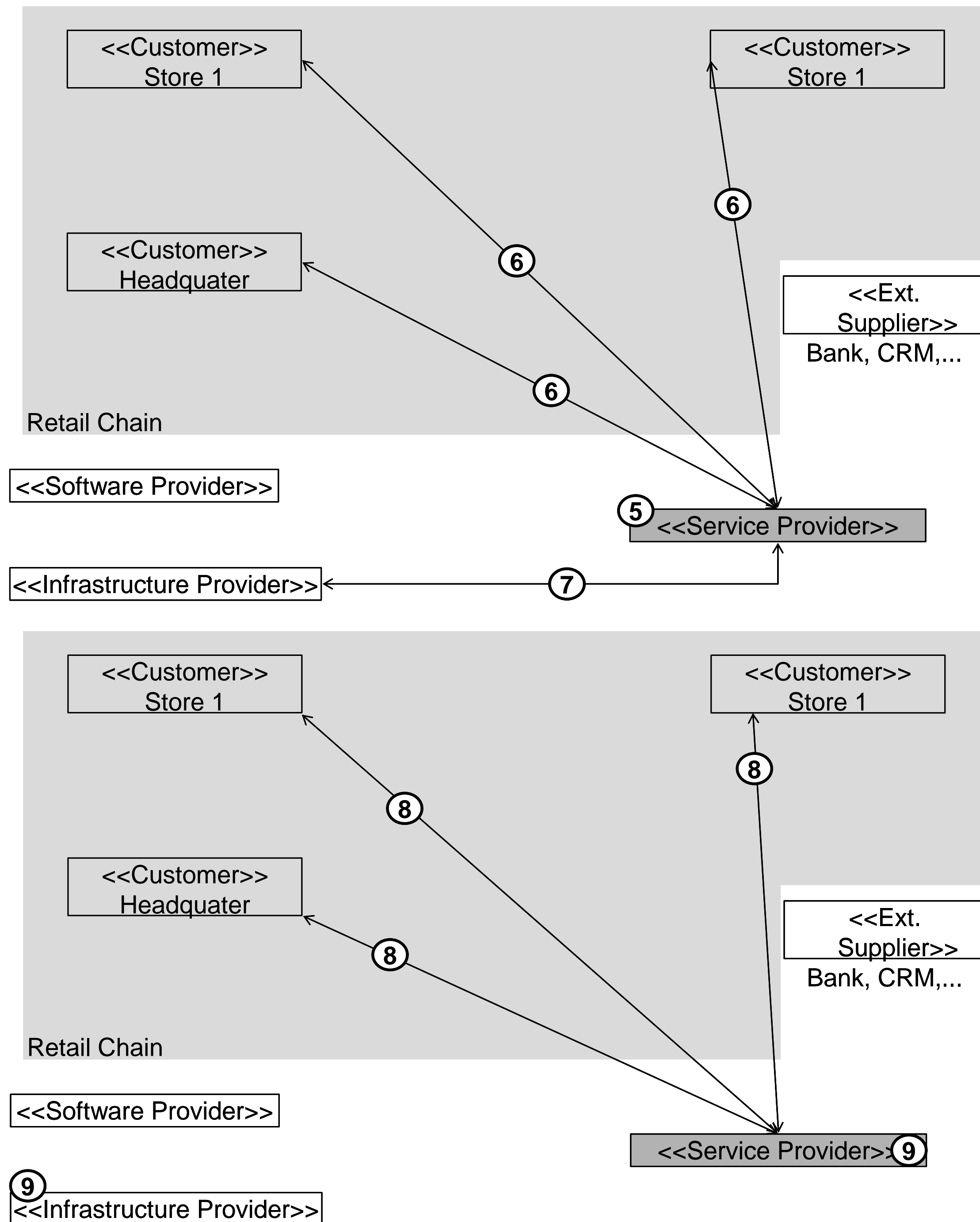

Figure 5: Provisioning lifecycle – continued.

2.2.1 Business Process

Sales Process

Figure 6 shows a sales process as it can be observed in a supermarket. This includes the processes at a single cash desk like scanning products using a bar code scanner, paying by credit card or cash as well as the booking of the purchase in the store inventory.

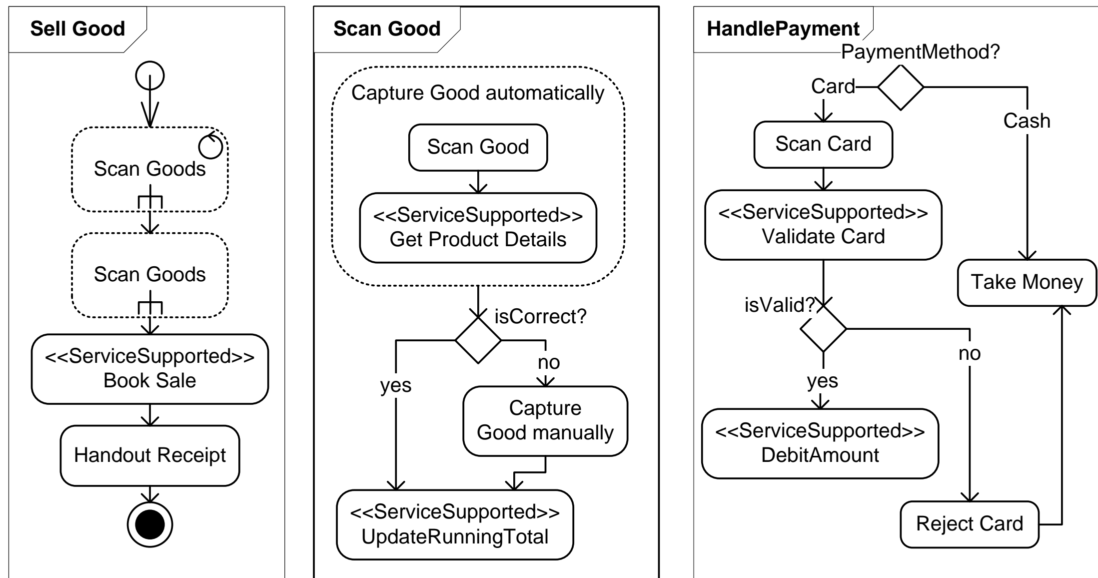

Figure 6: Sales Process.

The business process is thereby supported by different service compositions, defined in the preceding section.

2.2.2 Service-oriented "Retail Chain" Scenario & Architecture

In this section, the architecture of the service-oriented retail solution is described in more detail utilising UML 2.0. Therefore a structural view on the system is given. Figure 7 shows the current status of the implemented system, containing the legacy CoCoME components and the web service components. It contains a super-component `ServiceOrientedRetailSolution`, consisting of five components and the component `External` which is used by the `ServiceOrientedRetailSolution` component.

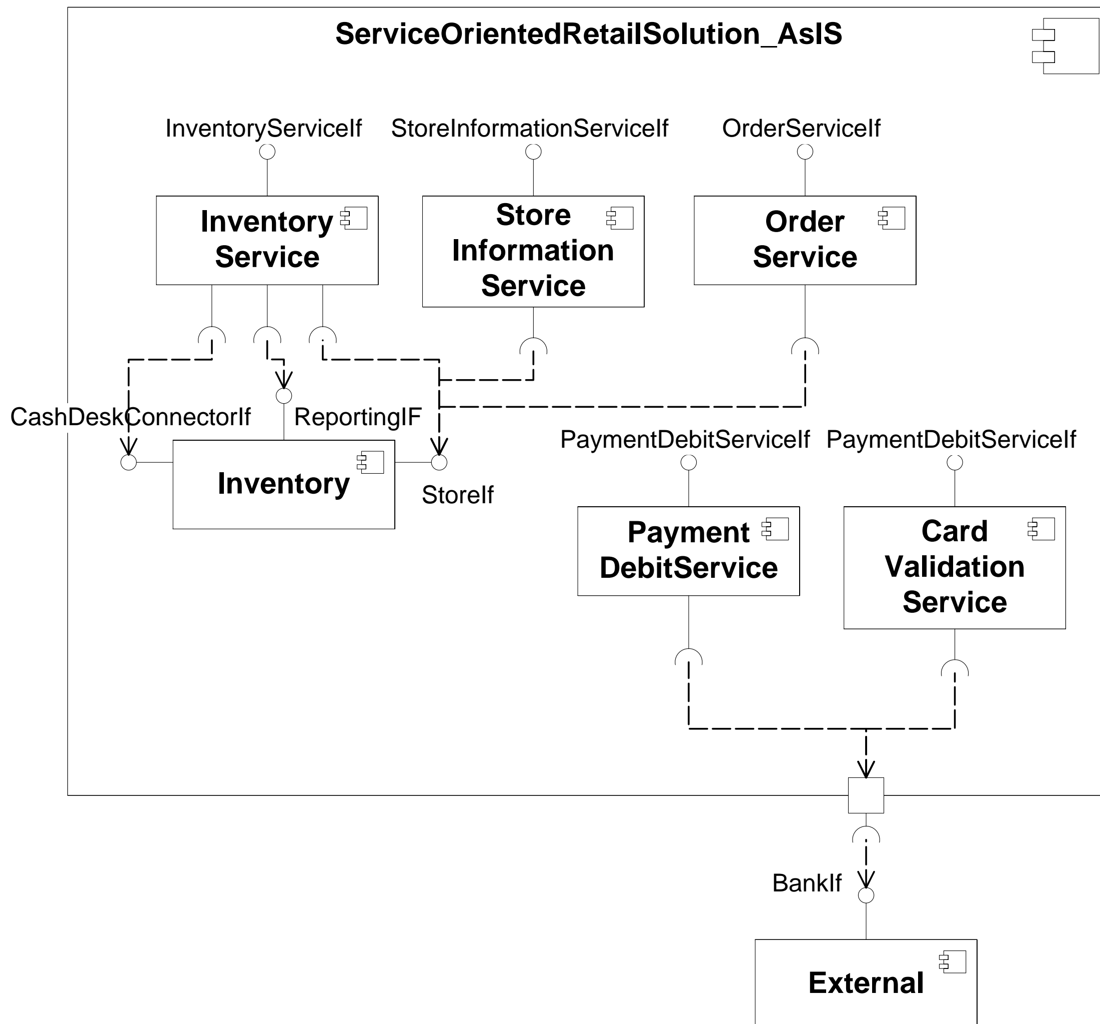

Figure 7: Current status of implemented system.

The five components `InventoryService`, `StoreInformationService`, `OrderService`, `PaymentDebitService` and `CardValidationService` implement the web service interfaces. `InventoryService`, `StoreInformationService`, and `OrderService` require the functionality provided by the legacy CoCoME component `Inventory`. The components `PaymentDebitService` and `CardValidationService` use the functionality provided by the `External` component, which is also a legacy CoCoME component. The Inventory component provides three interfaces. The interface `CashDeskConnectorIf` defines a method for getting product information like description and price using the product bar code. The interfaces `StoreIf` and `ReportingIf` deliver results of database queries. `External` provides the interface `BankIf` to handle the card payment. For each store instance in the trading system exists one instance of `Inventory` and `External`.

Figure 8 depicts a detailed view on the customized legacy CoCoME component `Inventory` including a layered architecture, `Application`, `Data` and `Database`.

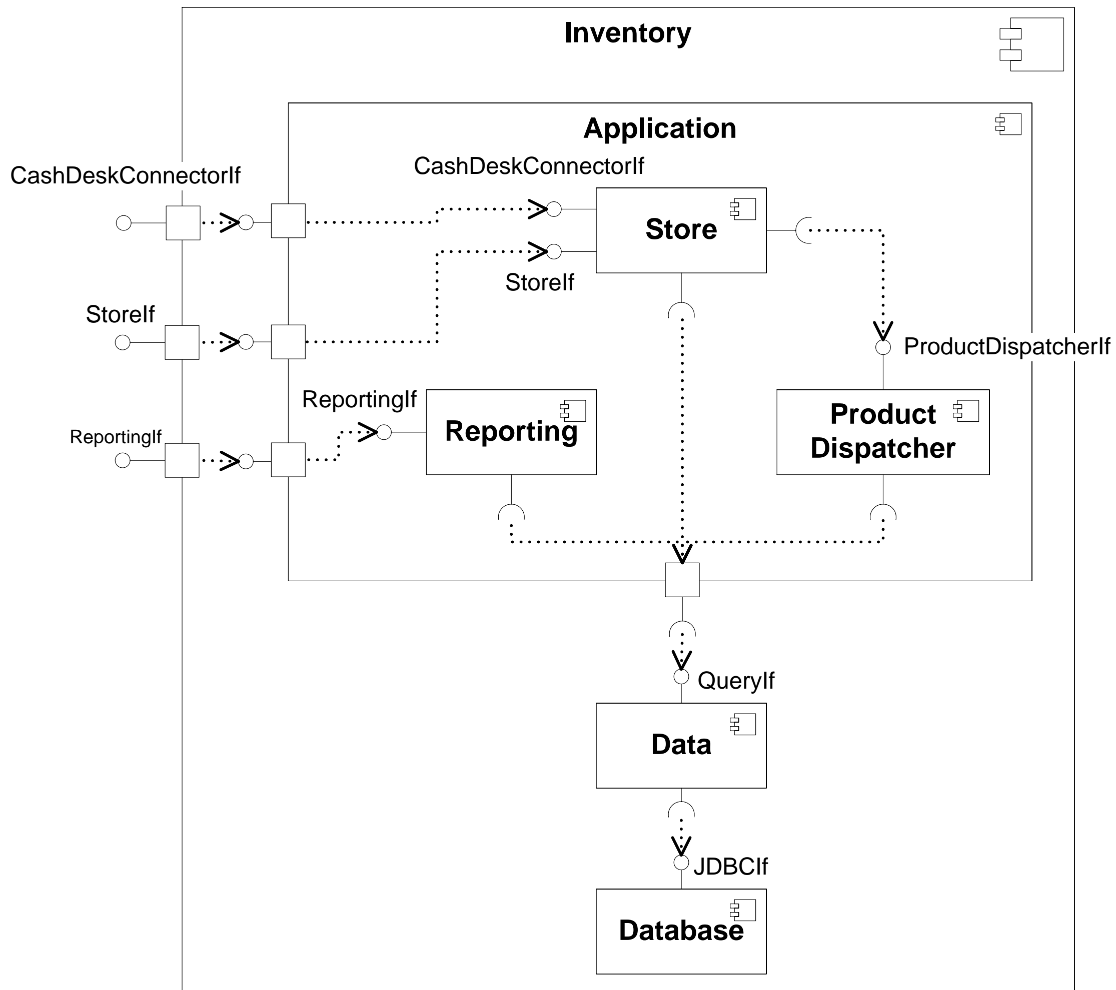

Figure 8: Layered architecture.

As shown in Figure 8 the `Application` is a sub component of `Inventory`. For each instance of `Inventory` exists one instance of the component `Database` where the data is stored. The component `Data` representing the data layer of a classical three-layer-architecture hides details of the database and provides data access to the application layer represented by the component `Application`. The communication between the components `Database` and `Data` is managed by Java Database Connectivity (JDBC) [3]in connection with Hibernate [4], an implementation of the Java Persistence API.
The component `Application` (Figure 9) represents the application logic consisting of three components `Reporting`, `Store` and `ProductDispatcher`. The component `Reporting` implements the interface `ReportingIf` whereas the component `Store` implements the interfaces `CashDeskConnectorIf` and `StoreIf`. It also requires the interface `ProductDispatcherIf`. The latter defines a method for the Enterprise Client to search for a product at another Store. While the communication between `Data` and `Application` is realized by passing references of persistent objects to the `Application`, the `Application` uses PoJOs (Plain old Java Objects) or Transfer Objects (TO) [5]to pass information.

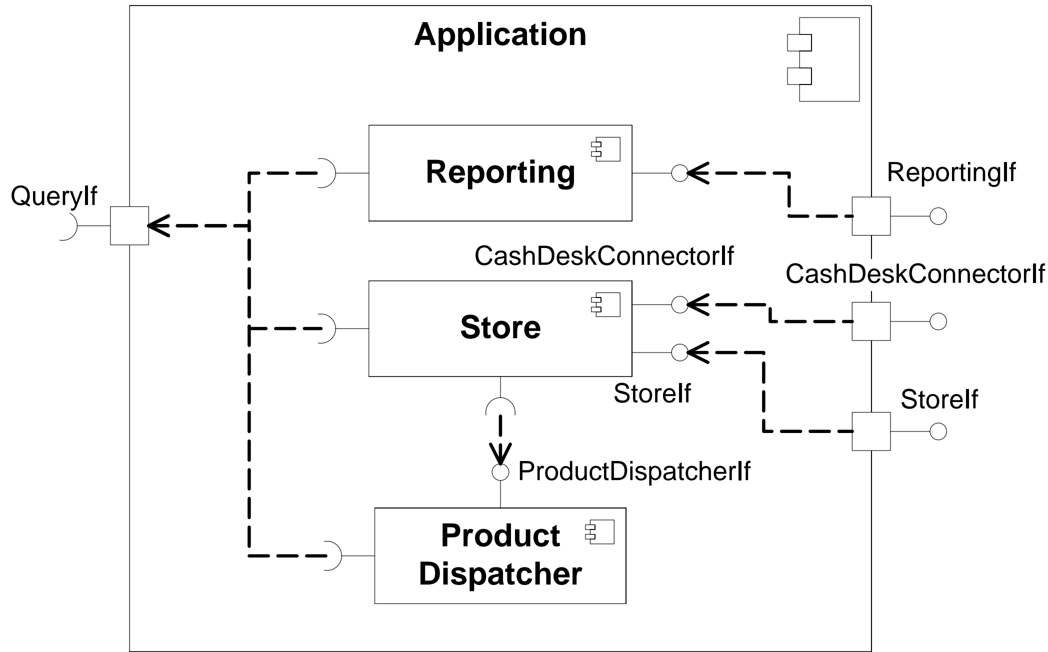

Figure 9: CoCoME component application.

2.2.3 Web Services

In this section, the web services and their interfaces, which form the base of ORC scenario are described in more detail.

Inventory Service

The `InventoryServiceInterface` (Figure 10) defines methods for getting product information and changing prices.

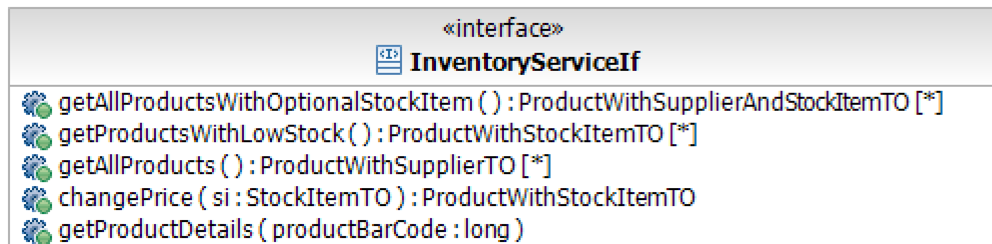

Figure 10: Inventory Service Interface

The Interface consists of five methods:

- `getAllProductsWithOptionalStockItem` determines all products of the portfolio of a given store and the supplier for each of them. Additionally the corresponding stock items are queried. It returns a list of products, their suppliers and the corresponding stock item if they have any.
- `getProductsWithLowStock` determines products and stock items which are nearly out of stock, meaning amount is lower than 10% of maximal stock. It returns a list of products and their stock item in the given store.
- `getAllProducts` determines all products of the portfolio of a given store and the supplier for each of them. I.e. it returns a list of products and their suppliers
- `changePrice` updates the sales price of a stock item. Needs as parameter the stock item with the new price and returns an instance of `ProductWithStockItemTO` which holds product information and updated

price information for the stock item identified by the parameter StockItemTO.

- getProductDetails determines the product and the item in the stock of the store by the given barcode. It returns a ProductWithStockItemTO instance which contains the identified product which is linked to the stock item of the store.

Store Information Service

The StoreInformationServiceInterface (Figure 11) provides a method for getting information about a store.

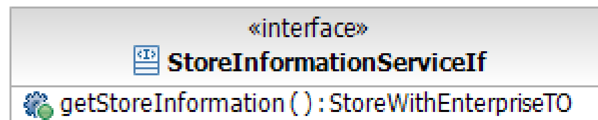

Figure 11: Store Information Service Interface.

The Interface consists of one method:

- getStoreInformation gets the transfer object with information of the store and his enterprise. This information is retrieved by the component during configuration and initialization.

Order Service

The OrderServiceInterface (Figure 12) defines methods for ordering products and updating the stock after order delivery.

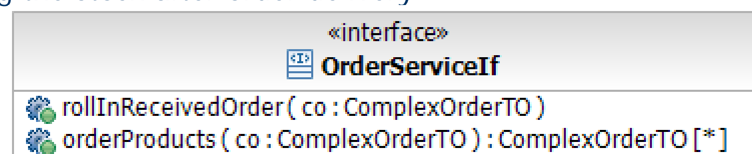

Figure 12: Order Service Interface.

The Interface consists of two methods:

- rollInReceivedOrder updates stocks after order delivery. Therefore it adds the amount of ordered items to the stock items of the store and sets the delivery date to the date of method execution. Requires an instance of ComplexOrderTO, which contains the order that is rolled in, as parameter.
- orderProducts creates a list of orders for different suppliers for an initial list of products to be ordered by a store. The product order is persisted and the ordering date is set to the date of method execution. The method requires an instance of ComplexOrderTO which contains all products to be ordered and returns a list of orders, one for each supplier that is affected.

Payment Debit Service

The PaymentDebitServiceInterface (Figure 13) provides a method for the debiting of payments.

Figure 13: Payment Debit Service Interface.

The Interface consists of one method:

- debitCard is used to debit a bank account. Possible return values are OK, NOT_ENOUGH_MONEY and TRANSACTION_ID_NOT_VALID. Requires a TransactionID which is issued with a valid PIN.

Card Validation Service

The `CardValidationServiceInterface` (Figure 14) defines a method for validating a credit card.

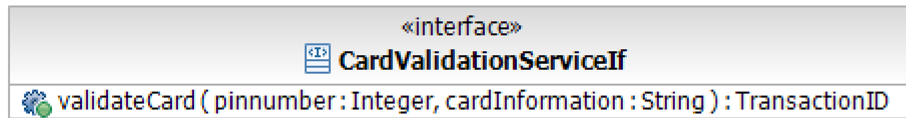

Figure 14: Card Validation Service Interface.

The Interface consists of one method: `validateCard` is used to validate a credit card. It requires the PIN and some information about the card. The method returns a transaction id which can be used for debiting the payment.

Payment Service as Service Composition

The `PaymentService` in Figure 15 orchestrates the web services `CardValidationService` and `PaymentDebitService` to handle the payments.

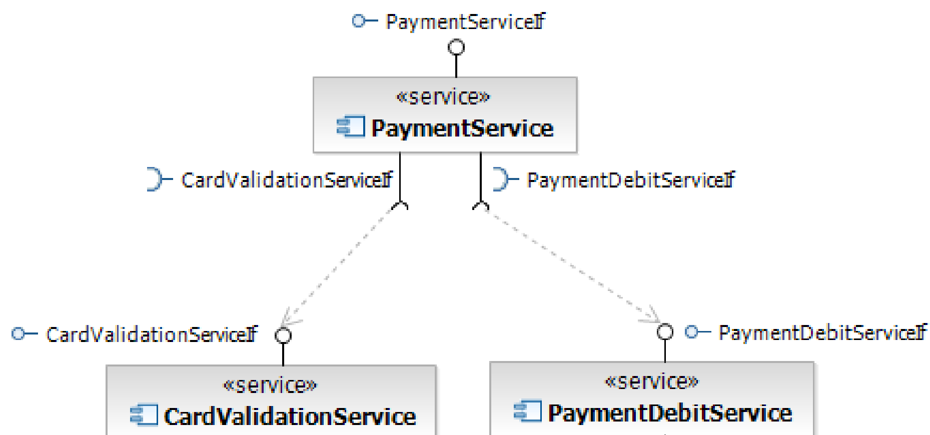

Figure 15: Payment Service Composition.

The PaymentServices is implemented as WS-BPEL [6] process using the development tool ActiveBPEL Designer [7]. It is deployed on the ActiveBPEL Engine as the run-time web service composition middleware. The sub process `PaymentService` includes card validation and debit payment and is required to support the whole Sale Process. Since the corresponding web services `CardValidationService` and `PaymentDebitService` are already available as basic web services the PaymentService can be built up by consuming these two services.

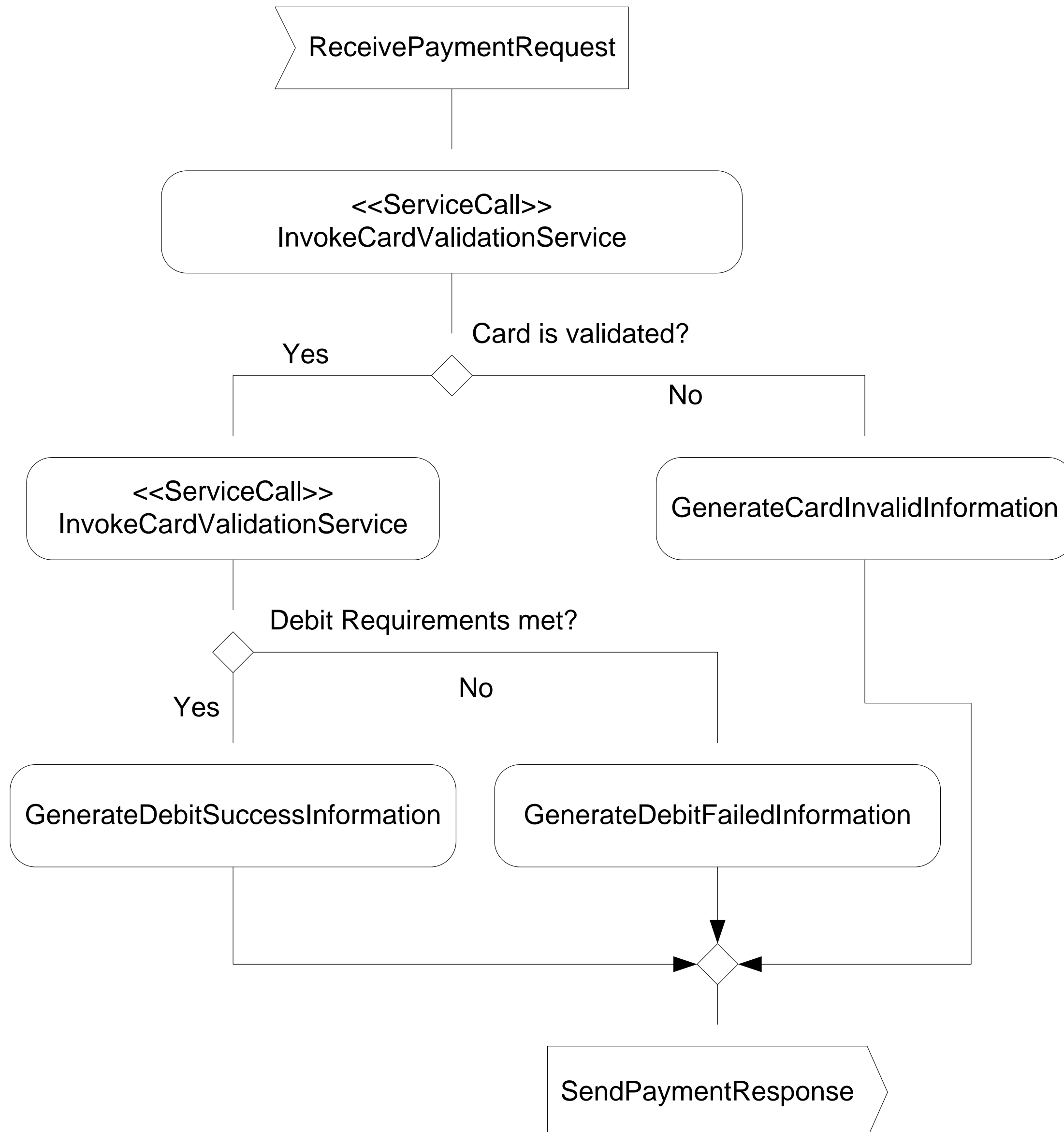

Figure 16: PaymentService BPEL Process.

As depicted in Figure 16, as soon as the payment request is received, the provided card information is validated. In case the card validation fails, the `CardInvalidInformation` is generated. If the card is validated, the second step is to validate if the debit requirement on this card is fulfilled. If the requirment is fulfilled, the `PaymentIsSuccessful` information is generated. Otherwise, the `DebitFailed` information is generated. Last but not least, all the generated information is sent back to the cashier as the `PaymentResponse`. In such a way, the card payment request can be automatically processed.

2.3 Specification of deployment options

The Service Provider offers several deployment options in a form of virtual appliances deployable on an internal infrastructure provided by Intel. Description of the testbed can be found in D.A4a [8]. Initially, the service provider will prepare two pre-built bundles (see Figure 17) containing all offered services supporting the retail chain.

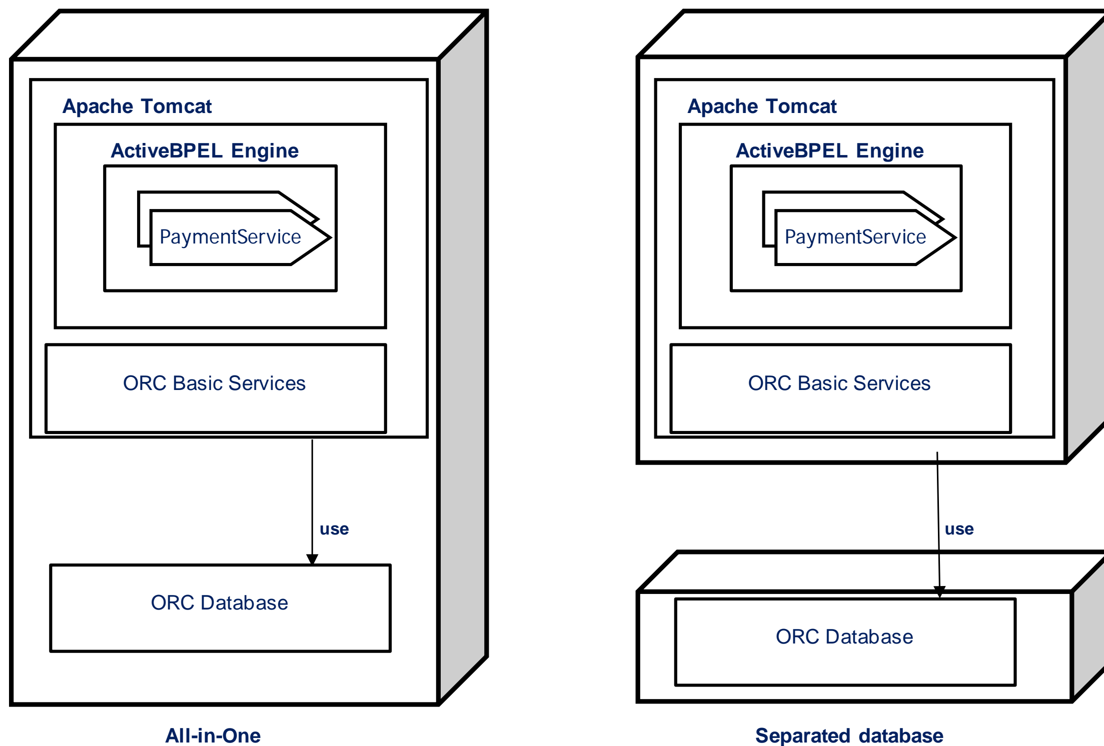

Figure 17: ORC deployment options.

- **All-in-one:** This deployment option consists of only one single virtual machine image. This image contains the database as well as the basic services and the composite services which include the application logic of the ORC.
- **Separated database:** The second bundle is a virtual appliance consisting of two virtual machine images, one providing an application server with all services deployed, and the other hosting only the database.

A detailed description on how to get these two deployment options running is provided in Appendix C.

2.4 Integration of the ORC with the SLA framework

To use the SLA framework in the ORC scenario, some ORC specific configurations, adaptions, and extensions are required. To enable the management of the ORC a manageability interface has to be designed and developed. This ORC specific management extensions include an instrumentation of the services that monitors the start and stop of service calls. Additionally, the design-time performance prediction requires an ORC specific model that reflects the structure and behaviour of the ORC. Furthermore, the abstract SLA hierarchy needs to be instantiated in the ORC scenario to reflect the included services and there relations.

2.4.1 Manageablity Configuration and Instrumentation of the ORC

To create a manageable version of the ORC we followed the engineering methodology presented in deliverable D.A6a section 4.3 [17]. Accordingly, we

first created a manageability model for the PaymentService and the referenced atomic services on basis of the existing ORC component model. For meeting requirements from SLA adjustment (see deliverable D.A5a, section 4.4 [16]) this model on the one hand defines, that management information of all provided and referenced operation calls should be available. On the other hand the model specifies that it should be possible to correlate the operation calls of the atomic services with the operation calls of the composite PaymentService. Further details can be found in section 4.5.2.

Based on this ORC-specific manageability model we had to perform two subsequent steps according to the engineering methodology. First, we had to create an instrumented functional design, which corresponds to the manageability model. Second, we had to configure the manageability infrastructure with a corresponding manageability data model. For this purpose, we manually created an XML-based configuration file. This configuration file is used by the manageability infrastructure to create the static database entries for the corresponding manageability data model (for details see D.A6a [17], section 4.5.3 and D.A3a [15], section 5.1.6). Furthermore, we had to define the concrete architecture of the manageable solution, which for instance specifies the number of management agents and their deployment. [8] provides an overview to the architecture of our manageable ORC solution.

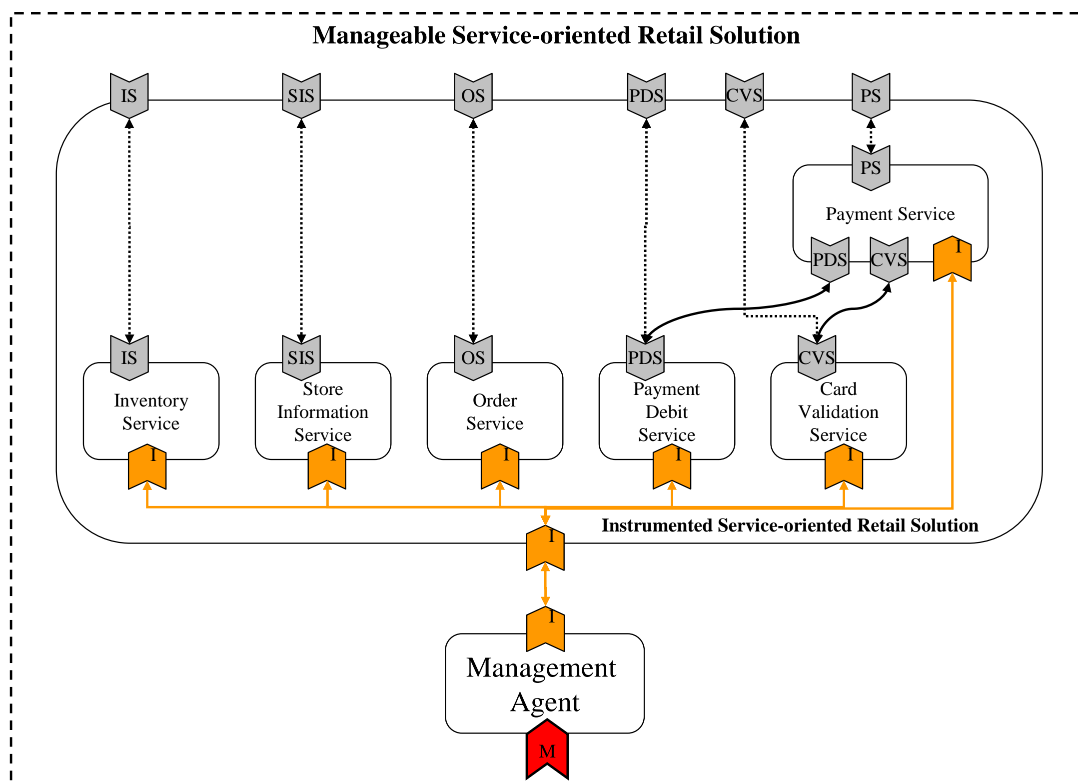

Figure 18: Overview to Manageable ORC

Thus, there is only one management agent component which is responsible for the all ORC components. This agent communicates with the instrumentation interfaces we added to each functional component. In the following, we provide some more details on this ORC-specific instrumentation, which took most effort and time as the open source technology we used only offers few management functionality out-of-the-box.

According to the manageability model, instrumentation is performed on all the services in the ORC scenario. The instrumentation of atomic services causes each atomic service to send two events per invocation. The first event regards the incoming Request message, and contains "who" is sending it, to what remote method it is directed, a timestamp indicating when the message was received, etc. The second event regards the outgoing Response message, and contains to "whom" it is directed, what exposed web method produced it, a timestamp indicating when it was produced, etc.

Since the atomic services are deployed in the ORC as an AXIS webapp, the event creations are activated by modifying the AXIS webapp's server-config.wsdl file. In this file we specify two additional handlers for the service we want to instrument. For example, if we look at the instrumentation of service InventoryService we have:

```
<service name="InventoryService" provider="java:RPC">
<requestFlow>
<handler type =
          "java:slasoi.instrumentation.axis.InstrumentationHandlerRequest"/>
</requestFlow>
<responseFlow>
<handler type
     ="java:it.slasoi.instrumentation.axis.InstrumentationHandlerResponse"/>
</responseFlow>
<operation name="changePrice" ...
</operation>
</service>
```

All we have to do is add the code in light blue. This activates the events creation by executing "InstrumentationHandlerRequest" on the incoming request, and "InstrumentationHandlerResponse" on the outgoing response.

The instrumentation of the Composite Service (for example, "PaymentService") is slightly more complex. For each invocation of the composite service, the instrumentation produces two primary events, one when the process starts and one when the process terminates. These events are obviously time-stamped. In the first case, the event contains "who" sent the message, and to what web method exposed by the composite service it was sent. In the second case, it contains to whom the response message is sent.

When instrumenting a composite service, it is also possible to request that the instrumentation produce secondary events. In particular, we can ask for events signalling a communication between the process and the external world. Indeed, in the ORC scenario we want to receive events every time the process makes a call to either service "CardValidationService" or service "PaymentDebitService", and every time it receives a response message from one or the other. The total number of messages produced by an invocation of the composite service is therefore 6: one for the process start, one for the process termination, two for communication with "CardValidationService", and two for communication with "PaymentDebitService".

To request these secondary events we need to provide a BPEL Process Instrumentation specification (under the form of an XML file), and place it in a special folder on the ORC deployment called "bpi". For example, for the "CardValidationService" we need to write:

```
<workflow>
<processID value="PaymentService"/>
<operation
   value="/process/flow/scope/invoke[@name=InvokeCardValidationService]"/>
```

```
<status value="input"/>
</workflow>

<workflow>
<processID value="PaymentService"/>
<operation
   value="/process/flow/scope/invoke[@name=InvokeCardValidationService]"/>
<status value="output"/>
</workflow>
```

The first workflow node states that we want to create an event when the process sends a message to "CardValidationService". The second workflow node states that we want another event when the process receives the response message from "CardValidationService".

A more general and detailed description of the instrumentation can be found in deliverable D.A3a, section 4.1 [15].

2.4.2 ORC Prediction Model

A software model of the ORC supports design-time prediction of the performance of the ORC services (Deliverable D.A6a [17], Section 2.7). The ORC model contains a specification of the service components that are involved in the sales process, as well as the software components of the underlying legacy application (the trading system). The component specifications include provided and required interfaces, behavioural specifications, and composition of components.

The ORC model in its current state comprises 15 components – 8 service components and 7 legacy components. The service components include basic services (such as the inventory service), and composed services (such as the payment service). Access to the inventory database is modelled through a database component. Furthermore, the repository contains all interfaces that are provided and required by the components.
In addition to the provided and required service interfaces, the components contain Resource Demanding Service Effect Specifications (RDSEFF) for each provided service operation. RDSEFFs abstract from a component's implementation and model only the performance relevant behaviour (see also Deliverable D.A6a [17], Section 2.5.3).

The ORC infrastructure model describes the resource environment to which service and legacy components are allocated to. For QoS prediction in SLA@SOI, the model is not fixed in advance, but used by the design-time prediction service as a template. The allocation of service and legacy components to resource containers allows calculation of concrete time demands from abstract resource demands specified in component RDSEFFs. To enable a prediction and comparison of the different deployment options, design-time prediction requires a separate allocation model for each deployment option.

A more detailed description of the prediction and the required software model can be found in Deliverable A6a [17], Section 2.7.

2.4.3 SLA Hierarchy

For each service in the ORC scenario an individual SLA needs to be defined. As described in Deliverable A5a [16], Section 2.4, the SLAs can be in relation with other SLAs. This abstract SLA Hierarchy is instantiated in the ORC scenario. This

concrete SLA hierarchy, which includes SLA for all services of the ORC as well the complete SalesProcess and the underlying hardware is sketched in Figure 19.

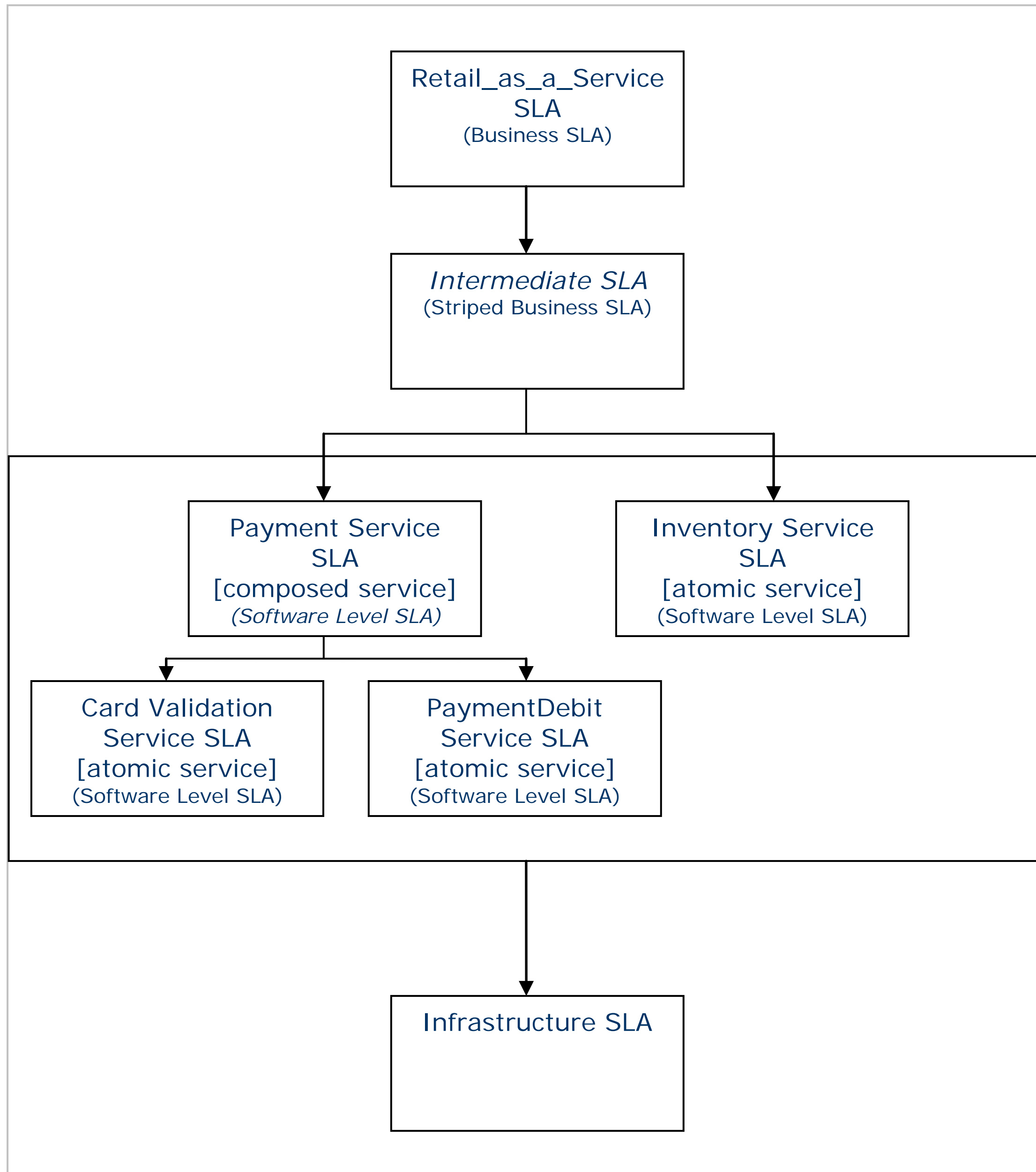

Figure 19: SLA Hierarchy

A more detailed description of the SLAs, their relations and the abstract SLA hierarchy can be found in Deliverable A5a [16], Section 2.4.

3 Adhoc Demonstrator

The adhoc demonstrator, depicted in Figure 20, consists of a graphical user interface and a simulation environment. The demonstrator can be deployed on arbitrary infrastructure supporting the SLA@SOI framework. The graphical user interface provides four different perspectives on the execution environment:

- **Service Customer Interface** provides a simple service and SLA template browser. It also allows users to start negotiation with the service provider based on selected Service Level Objectives as well as monitoring of active SLAs.
- **Service Provider Interface** provides service providers with an overview of active services, SLAs, relationships, infrastructure, violations, etc.
- **Logging/Visualization** interface can be used by researchers and technical staff to get a deeper understanding of the SLA@SOI Management Framework.
- **Workload Control Panel** provides means of controlling the actual usage profile of the customer, independent of the profile that was used during the negotiation phase. The purpose of this panel is to control the behaviour of the SLA@SOI framework.

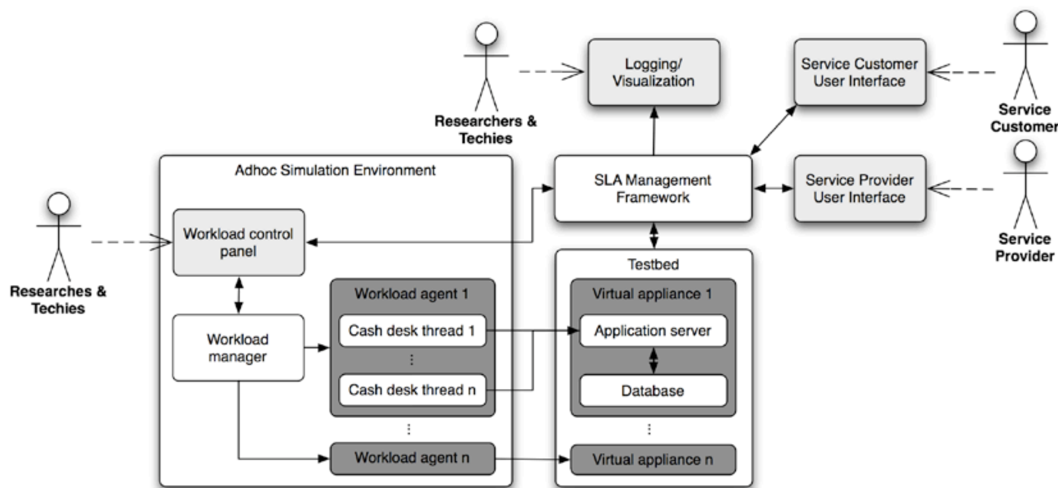

Figure 20: Schematic overview of the adhoc demonstrator.

3.1 Simulation Environment

3.1.1 Introduction

The simulation environment provides a tool for simulating a real-world sales process on multiple cash desks running in parallel. It consists of the workload generator manager responsible for the distribution of workload and a distributed and multithreaded network of workload agents invoking services offered by the service-oriented application. Although the sales process is hard-coded in every workload agent it provides configuration options for every step in the process allowing users to tailor the workload to their needs and/or actual usage profile of the customer.

The workload manager and agents are loosely coupled with the remaining components of the adhoc demonstrator by using the message bus interface. Detailed description of the message bus can be found in Section Infrastructure Messaging in D.A4a [8]. The message bus supports full-duplex communication, i.e., it is used to transfer commands from the workload manager panel/console to the manager as well as for reporting the status of each workload agent which is then displayed in the graphical user interface.

The workload agent is an abstraction that defines the sales process independently of the underlying WebService access used. Current implementation supports two client types, namely JAX-WS [9] integrated with Sun Java 6, and Apache Axis [10] (see Appendix D for more information).

The architecture of the simulation environment is depicted in Figure 21. When the workload manager is initialized it starts listening to a pre-configured message bus channel for commands and configurations. It also starts sending its own heart-beat message to advertise its ability to receive commands. These messages contains ID of the manager to enable adhoc demonstrator to differ between them. Heart-beat messages are sent in regular intervals (1 second by default) and are used by the user interface to detect irresponsive workload managers. Initially, the manager does not start any workload agent.

Once the workload configuration is sent from the adhoc GUI over the message bus all workload managers receive the message but only the one whose ID matches the ID in the request processes the message and initialises agents (threads). Initially, all threads are in *idle* state, waiting for further instructions. Once they are started they are reporting their status over the message bus which is, eventually, displayed in the GUI panel.

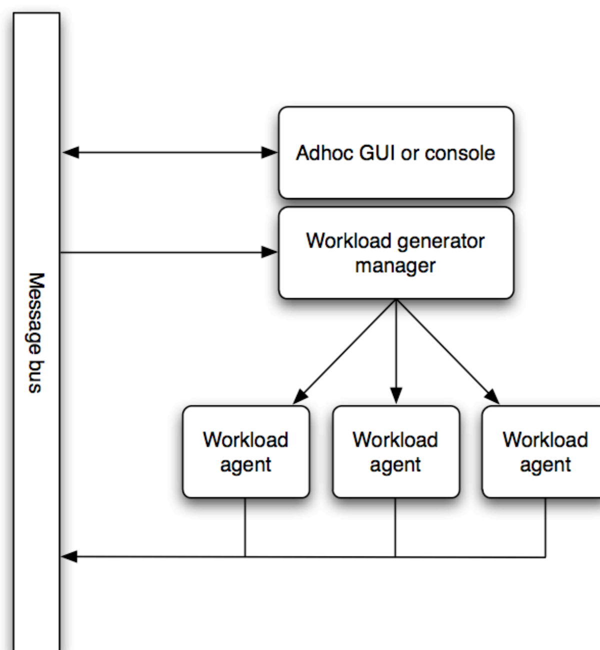

Figure 21: The architecture of the simulation environment.

Besides heart-beat and configuration messages, there are three additional message types sent over the message bus: commands, command responses, and free text messages.

Using commands it is possible to control the behaviour of the manager and agents by starting, pausing, resuming and stopping any one of them. The following list describes all available commands:

- **start** starts all workload agents that were already configured
- **startInstance <agent> <instance>** starts certain instance of the given workload agent
- **startAllInstances <agent>** starts all instances of an agent
- **stop** stops the workload generator (all it's agents and it's instances)
- **stopInstance <agent> <instance>** stops certain instance of the given workload agent
- **stopAllInstances <agent>** stops all instances of an agent
- **pause** temporarily pauses all workload agents
- **pauseInstance <agent> <instance>** pauses certain instance of the given workload agent
- **pauseAllInstances <agent>** pauses all instances of an agent
- **resume** resumes all workload agents
- **resumeInstance <agent> <instance>** resumes certain instance of the given workload agent
- **resumeAllInstances <agent>** resumes all instances of the given agent type
- **kill** kills all workload agents that were already configured
- **killInstance <agent> <instance>** kills certain instance of the given workload agent type
- **killAllInstances <agent>** kills all instances of the given agent type
- **listAgents** returns the list of all agents with their statuses
- **quit** terminates all workload agents and quits the workload generator manager.

Command response format depends on the underlying command: it can be free text or JSON [11] / XML serialised object. Free text messages are also used by workload agents reporting their status, e.g., waiting for the Web Service operation to return.

3.1.2 The sales process

Every workload agent is imitating the sales process depicted in Figure 22, starting with the selection of products that are used to update the running total of the purchase. Once all products are processed, the workflow simulates the payment and completes the flow with a call to book the sale and update the stock. Real-world workload is achieved with additional delays and various probabilities that guide the workflow. Below is a list of most important steps in the workflow:

1. **Preparation.** This step is omitted from the main workflow. Its only purpose is to retrieve available products from the InventoryService so that successive steps work on valid products. Only barcodes are stored at this stage.
2. **Init customer.** This step initialises the sale. Products are randomly chosen from the list of all available products. This step simulates buyer's shopping cart.
3. **Get product details** service call is used to retrieve all the details of the current product. Although all the details are retrieved, only the price is used to update the running total.
4. **Is data recognised?** Based on pre-configured probability, it is possible that the product is not recognised. In this case, additional delay is used to simulate manual capturing of product's price.

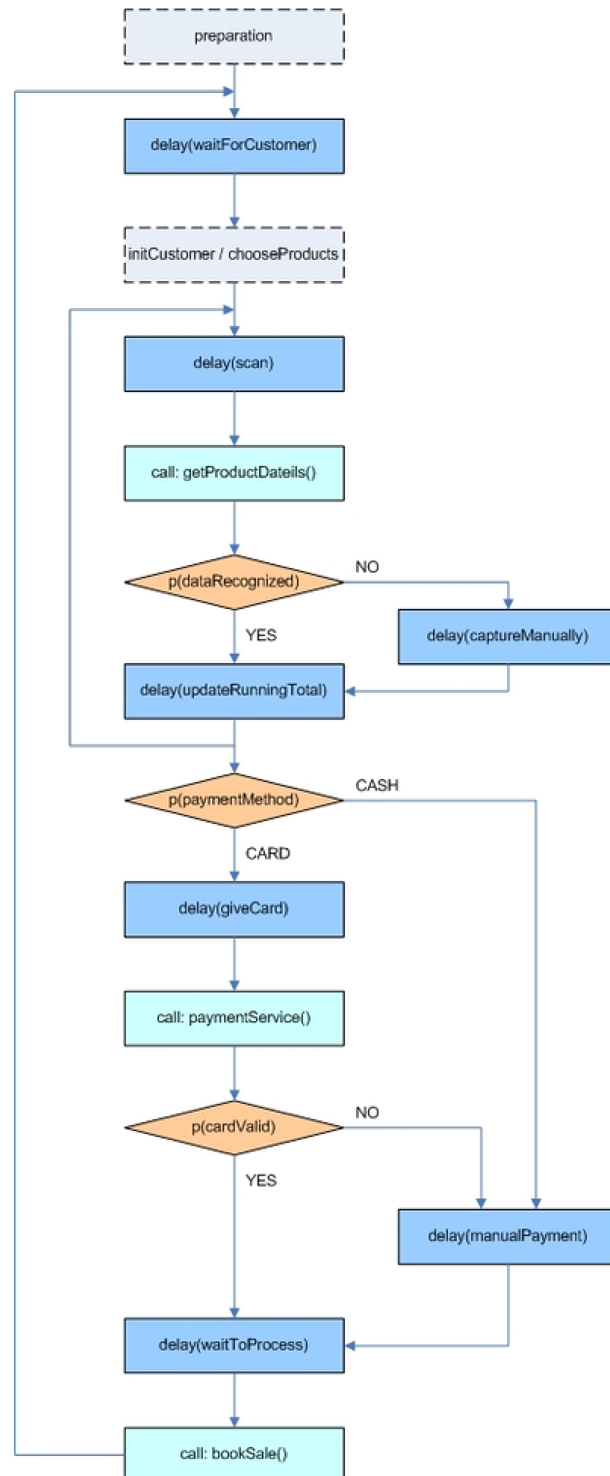

Figure 22: The sales process workflow. Normal service calls are displayed in light blue. Dark blue rectangles are used for delays. Decision symbols are depicted in orange colour with the name of the probability used.

5. **Payment method selection** controls the amount of credit card payments and cash payments.
6. **Credit card payment** calls the Payment service to validate the card. As we are not working with real data, we use another probability to decide whether to send correct PIN number or not. If the card is rejected, the manual payment is executed. Otherwise the Payment service calls the PaymentDebitService internally to debit the given credit card.

7. **Book Sale** is the last call in the workflow that sends the list of bought products and updates the stock.

Probabilities and delays are all part of the configuration that is sent to the workload manager. Normal (Gaussian) distribution is used to describe delays appearing in the workflow, resulting in two parameters for each delay, namely the base value and the standard factor (deviation). Alternatively, Poisson distribution is also supported by workload generators, but not by the GUI (workload generators can also be configured via means of a configuration file or by XConsole client [12]. These parameters are described in detail in the following section.

3.1.3 Configuration options

Configuration of workload generators is done via message bus. Message is formatted as JSON (JavaScript Object Notation) string consisting of two sections. The first section describes global configuration, reflecting common properties of all agents, while the second section describes actual workload agents. The latter supports the configuration of arbitrary number of workload agents in a single configuration string. An example of the JSON formatted configuration is shown in Table 1.

Table 1: An example of the JSON formatted workload configuration.

```
{
	// Global settings
	"id" :"test",
	"webClientType"	: 1,
	"agents": [

	// Configuration of the first agent
	{
		"id"		: 1,
		"probabilities"	: {
			"dataRecognized"	: 0.5,
			"paymentMethod"	: 0.5,
			"cardValid"		: 0.5
		},
		"numInstances"		: 3,
		"url"			:
"http://172.16.118.209:8081/Store1/",
		"webClientType"		: 1,
		"numberOfBoughtProducts" : 10,
		"randomSeed"		: 32453,
		"timeDistribution"	: 2,
		"delay" :
		{
			"waitForCustomer"	: 5000,
			"scan"			: 2000,
			"captureManually"	: 5000,
			"updateRunningTotal"	: 3000,
			"giveCard"		: 4000,
			"manualPayment"		: 5000,
			"waitToProcess"		: 7000
		}
		"randomStandardFactor"	:
		{
			"waitForCustomer"	: 2000,
```

```
                        "scan"                       : 1000,
                        "captureManually"            : 2000,
                        "updateRunningTotal"         : 1000,
                        "giveCard"                   : 3000,
                        "manualPayment"              : 3000,
                        "waitToProcess"              : 5000
                },
        },

        // Configuration of the second agent
        {
                "id"             : 2,
               // ...
        },

        // Additional agents
        // ...
        ]
}
```

Since the message bus channel is public all messages are sent to all workload managers connected to the channel. Each workload manager checks the ID of the configuration message and it executes the configuration, if the message ID corresponds to its own ID. Current implementation supports two types of clients, namely JAX-WS and Axis. Appropriate client is selected based on the *webClientType* property. Although this property is global it can be overridden by every agent. For further information about the global section refer to table Table 2.

Table 2: Description of global configuration properties.

Field	Type	Description	
id	String	the name of the workload manager	
webClientType	Integer	type of the interface used, currently there are 2 possibilities	
		1	JaxWs
		2	Axis
Agents	JSONArray	array of agents configurations (see description below)	

Agent configuration properties are described in tables below. The manager uses the *numInstances* property to initialise a given number of workload agents with the same configuration. To configure agents with different workload properties, specify separate agent descriptions.

Table 3: Agent configuration properties.

Field	Type	Description
id	String	id of the agent, usually numbered sequentially
probabilities	JSONArray	Workflow probabilities (refer to Table 4)
numInstances	Integer	Number of Instances of this agent to start
url	String	Url of the service, the agent is calling
webClientType	Integer	Client type (1 for JAX-WS, 2 for Axis)
numberOfBoughtProducts	Integer	Number of products to buy

randomSeed	Long	Random seed used by random number generator.
timeDistribution	Integer	Random distribution to use for delays.
delay	JSONArray	Base delays for workflow tasks (refer to Table 5)
randomStandardFactor	JSONArray	Standard factor for Gaussian distribution (Table 5)

Table 4: Workflow properties defining probabilities of various branching conditions. Within the workflow, these properties are used for more realistic simulation of the sales process. Values in tables only denote extreme values, values between minimum and maximum are also applicable (e.g., a probability of 0.5 specifies equal chance of both branches in the workflow).

Field	Type	
dataRecognized	Double	Probability that data on an article is recognized: 1 – always recognized 0 – never recognized
paymentMethod	Double	Probability that customer will pay with credit card: 1 – always pay with credit card 0 – always pay with cash
cardValid	Double	Probability that card is valid: 1 – card is always valid 0 – card is never valid

Table 5: Description of workflow delays. Each delay is described in terms of the normal distribution, thus there are two arrays used (delay and randomStandardFactor).

Field	Type	
waitForCustomer	Distribution	Time to wait for new customer
scan	Distribution	Time to wait for scanning of a product
captureManually	Distribution	Time needed to enter barcode manually
updateRunningTotal	Distribution	Time to update running total
giveCard	Distribution	Time needed for a customer to give the credit card
manualPayment	Distribution	Time needed for manual payment (cash)
waitToProcess	Distribution	Time needed to process the bill

3.2 Graphical User Interface

The graphical user interface for the adhoc demonstrator is a Java application based on Eclipse Rich Client Platform technology – an open platform that makes use of Eclipse framework GUI components.

For adhoc demonstrator development requirements please see Appendix E.

The graphical user interface of the adhoc demonstrator consists of four main panels. Each of them corresponds to one of four different perspectives on the execution environment, namely Customers panel, SLA Hierarchy panel, Framework Logging panel and ORC Simulation panel. These views are organised

in tabs that are always available to the user. The navigation between tabs is possible by clicking on tabs, by File menu shortcuts, and by keyboard shortcuts (Figure 23).

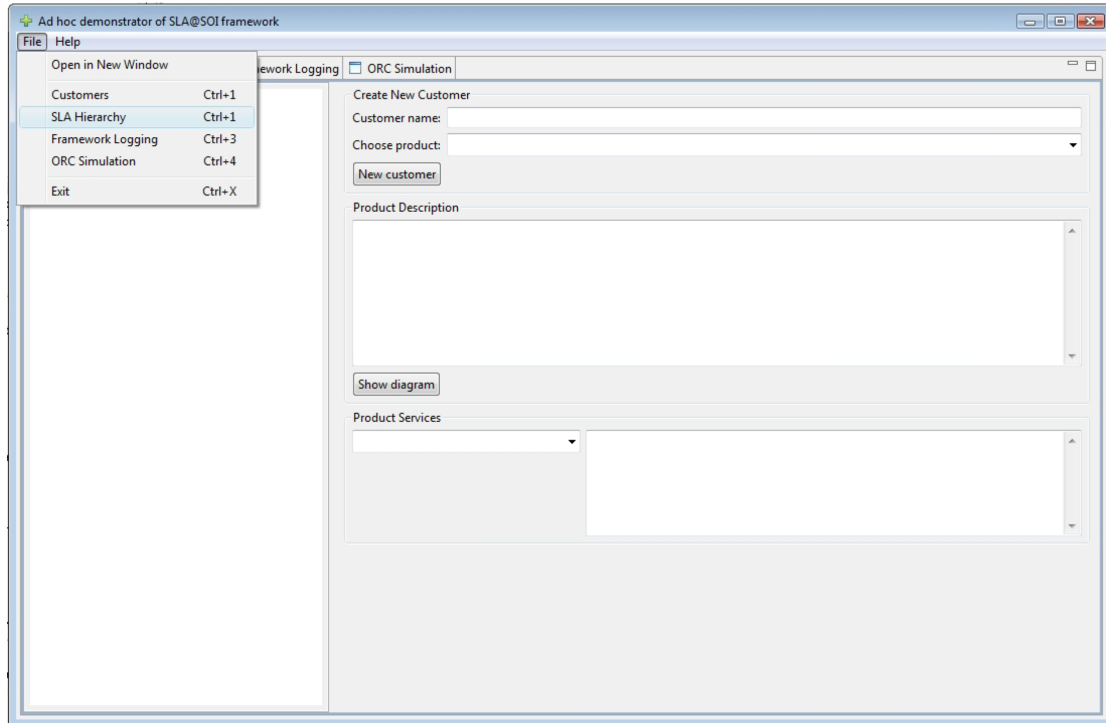

Figure 23: Navigation between tabs in adhoc demonstrator.

Alongside File menu item there is a Help menu item that displays basic information about the SLA@SOI project and the adhoc demonstrator (Figure 24).

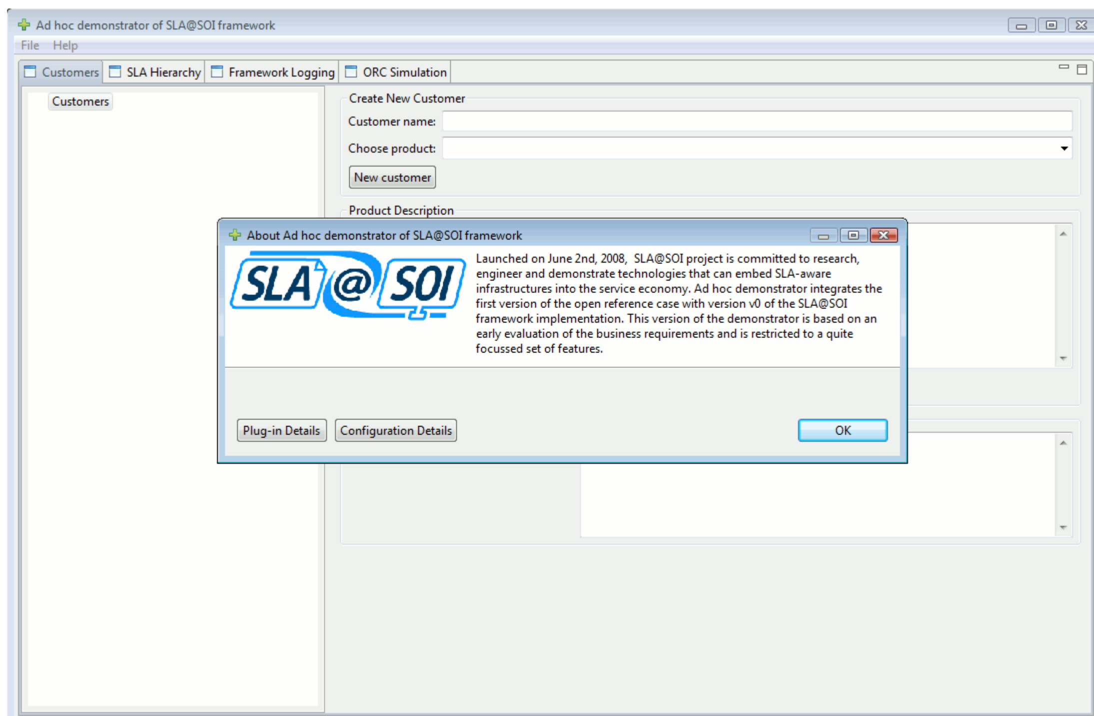

Figure 24: Basic information about adhoc demonstrator.

Being an Eclipse RCP application the adhoc demonstrator GUI also inherits basic window management features such as docking, pulling tabs out of the main window, sorting tabs in the arbitrary order, minimizing, maximizing, sizing, moving and some other, OS platform specific, functionalities (Figure 25).

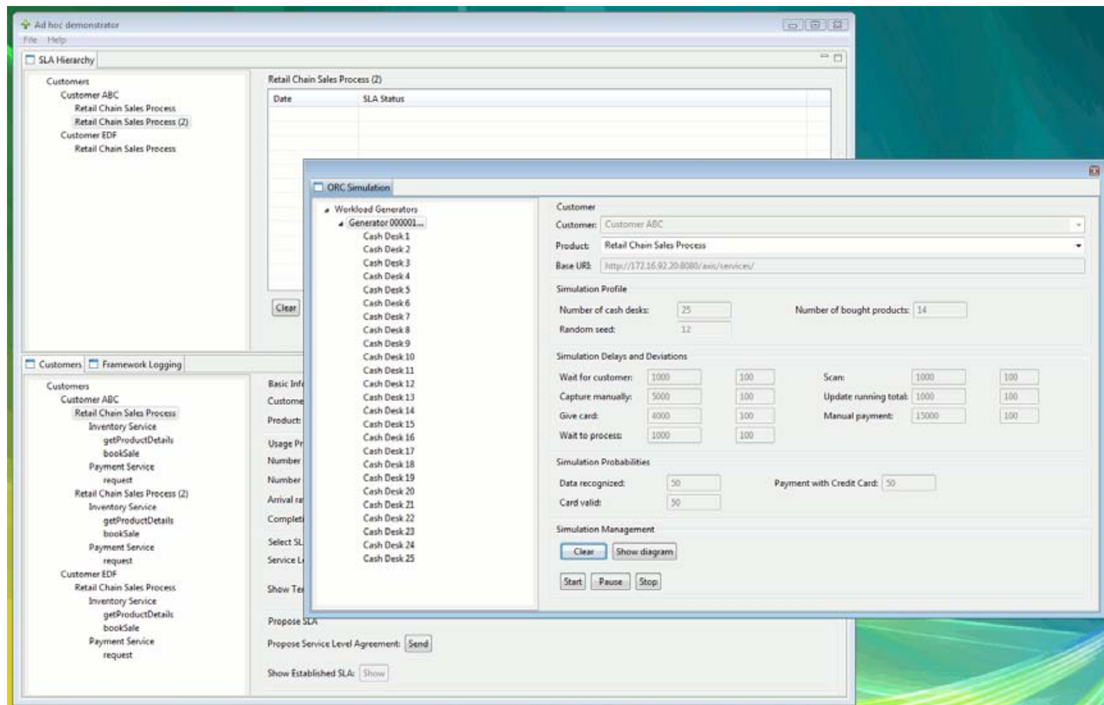

Figure 25: Adhoc demonstrator window management.

3.2.1 Customer creation view

The customer panel in the adhoc demonstrator is the starting point for the outside communication with SLA@SOI framework. It provides searching for products, their services and operations, although only Retail Chain Sales Process product based on Open Reference Case is available in year one. It represents a trading system dealing with 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 booking the sale.

When a new account in the adhoc demonstrator is created, customer can study and modify the Service Level Agreement Template with pre-defined values to create a Service Level Agreement. The complete SLA offer is sent to the E-Contracting module which is the only point of access to the SLA@SOI framework available to customers. Once the SLA offer is received it is delegated to the negotiation and provisioning modules ([14] [16]).

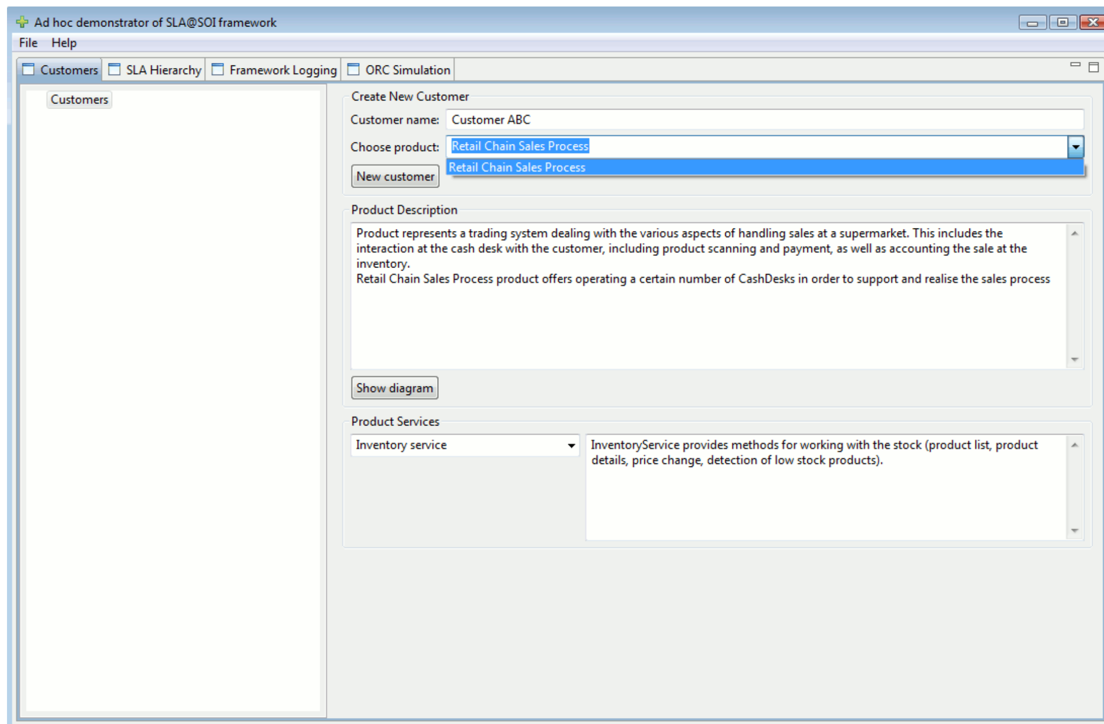

Figure 26: Creating new customer.

The Customers panel consists of two main views - customers tree view on the left side and content view on the right side (Figure 26). Before any customer account is created only the root item is available in the tree view with an associated Create New Customer functionality in the content view.

The view for creating a new customer account has three GUI parts. The first is Create New Customer group box, which has Customer name and Choose product fields. The fields in the second and third group box are filled immediately after customer name and product are specified. The second contains short product description and a button which opens a window dialog with an image as an additional explanation of a product (Figure 27). The last group box contains combo box, where descriptions of services can be obtained by selecting a specific product service.

Adhoc demonstrator supports an arbitrary number of customers and an arbitrary number of customer's products (Figure 28). After the first customer account is created the view for creating additional customers can be found at the root item of the tree.

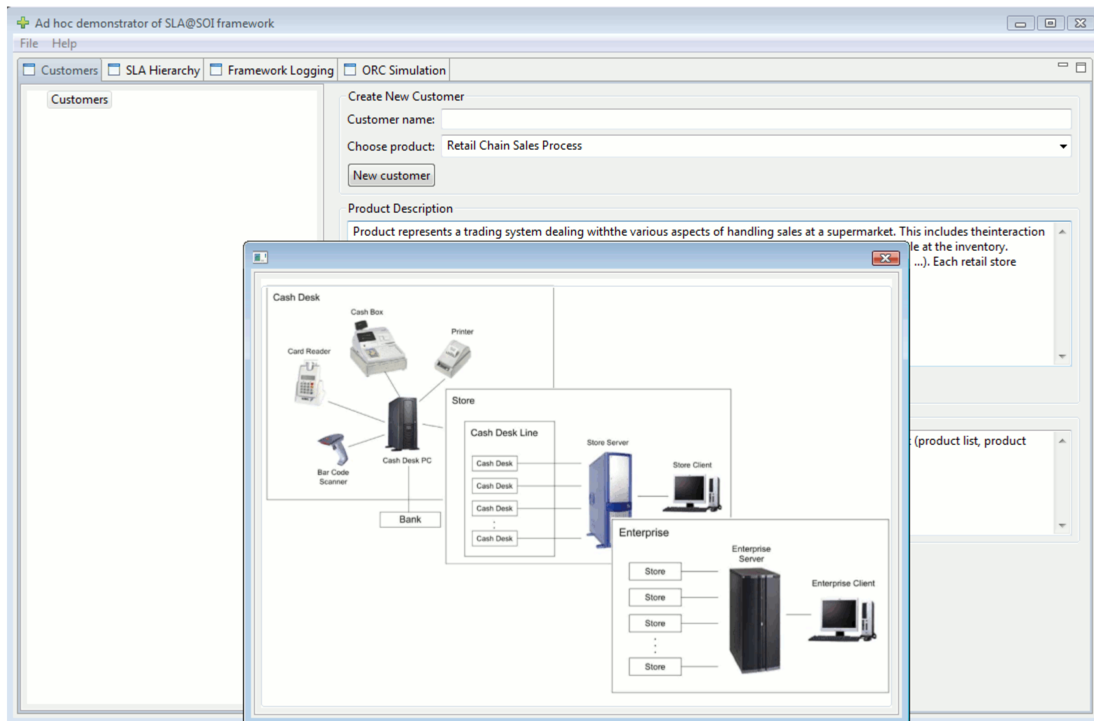

Figure 27: Product diagram.

The new customer account is created by clicking a New customer button, which causes the appearance of the customer's tree structure in the tree view on the left side of the panel. The tree view is used for navigating between customer's products, product services and service operations. For each one of them the associated view on the right side is available.

The root item of the customer's tree structure provides simply a view with a list of all customer's products and their descriptions. All customer's products are collected in the tree view as Customer's root item subitems (Figure 28).

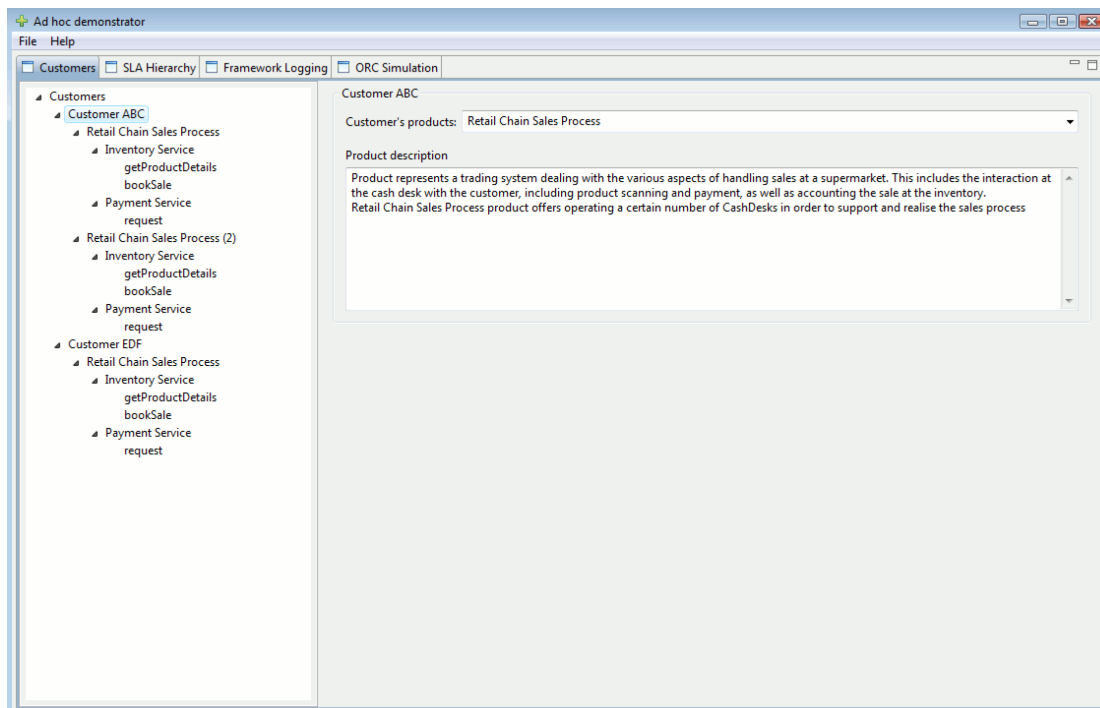

Figure 28: Customer view.

The depth level of the product tree structure depends on the granularity of product services. The tree structure includes product services, their subservices to the arbitrary level and service operations. Each service item view contains short service description (Figure 29).

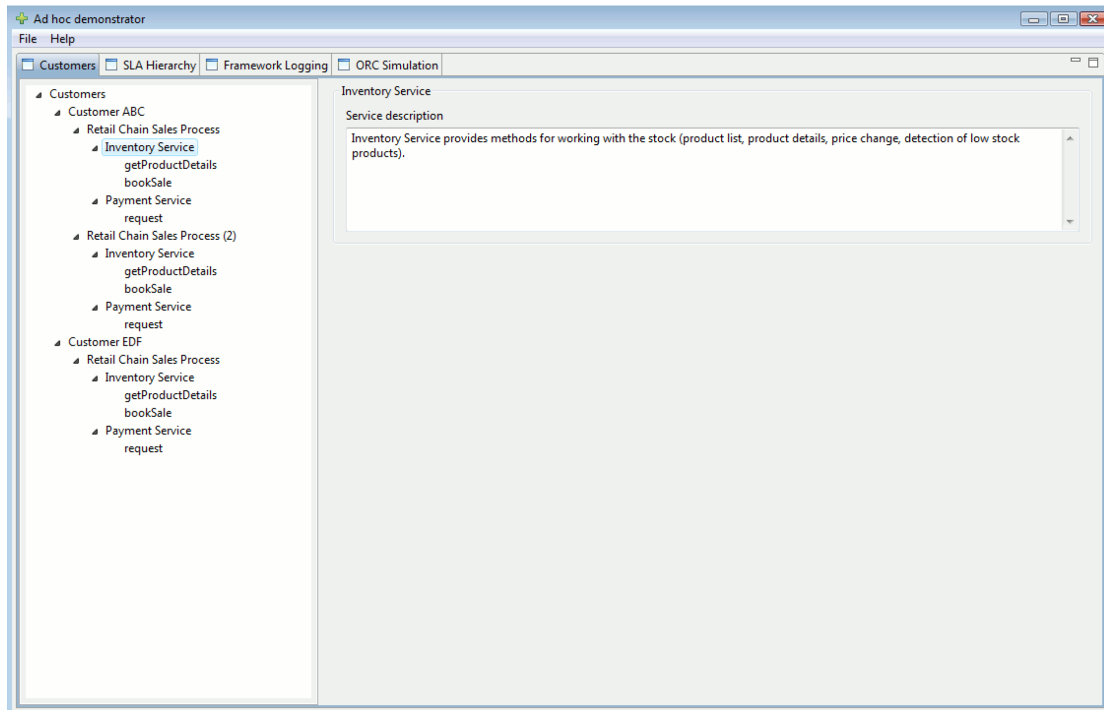

Figure 29: Service view.

By navigating between tree items and specifying various parameters customer creates an offer of a Service Level Agreement. The parameters to be defined can be found in the product root item view (Figure 30) and in the operations items view (Figure 31).

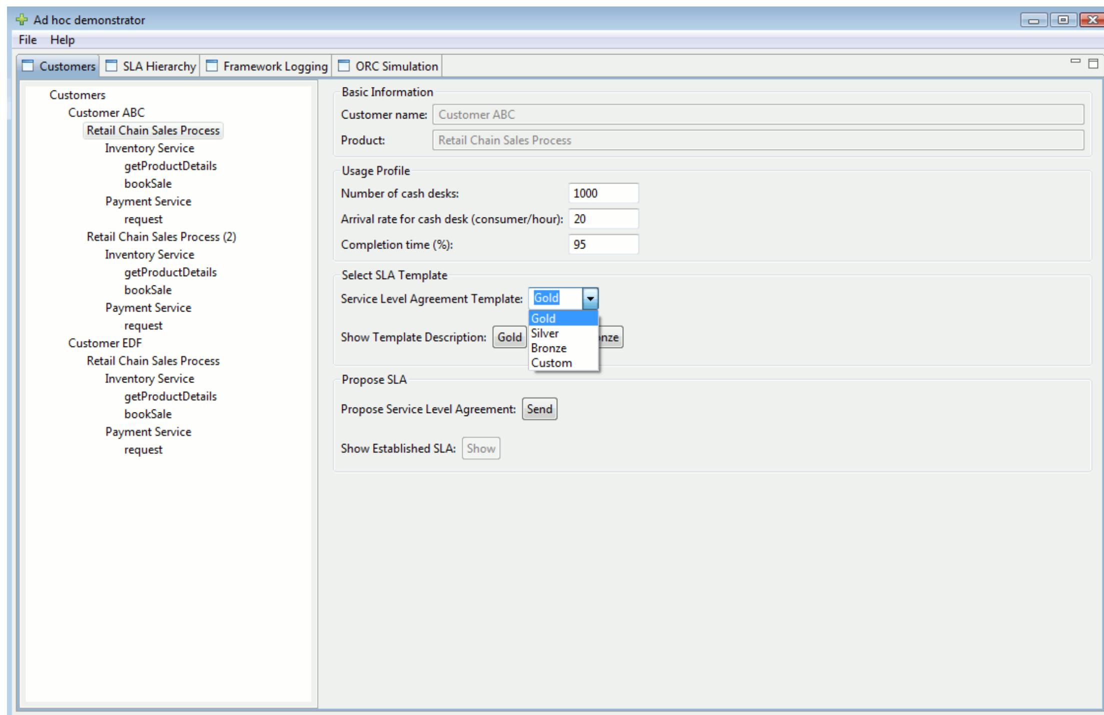

Figure 30: Product view.

The main SLAT view is the product root view. It consists of Basic Information, Usage Profile, Select SLA Template and Propose SLA group boxes. The first serves only as a customer and product identification.

By choosing one of the gold, silver or bronze options in the Select SLA Template group box, the parameters in the Usage Profile group box get pre-defined values. The parameters in Usage Profile group box are:

- number of cash desks
- arrival rate of consumers for cash desk (measured in consumer per hour)
- completion time (measured in percentage – e.g. at least for 90% of operations invocations is completion time less than 20ms)

Service Level Objective terms are then calculated for each operation from these parameters. Each operation has three terms (Figure 31):

- arrival rate (requests per second)
- completion time (percentage)
- upper limit for completion time (in milliseconds)

For example, the terms values for operation in Figure 31 mean that arrival rate must be less than 555.55 req/s and completion time must be less than 20ms in 95% of the invocations.

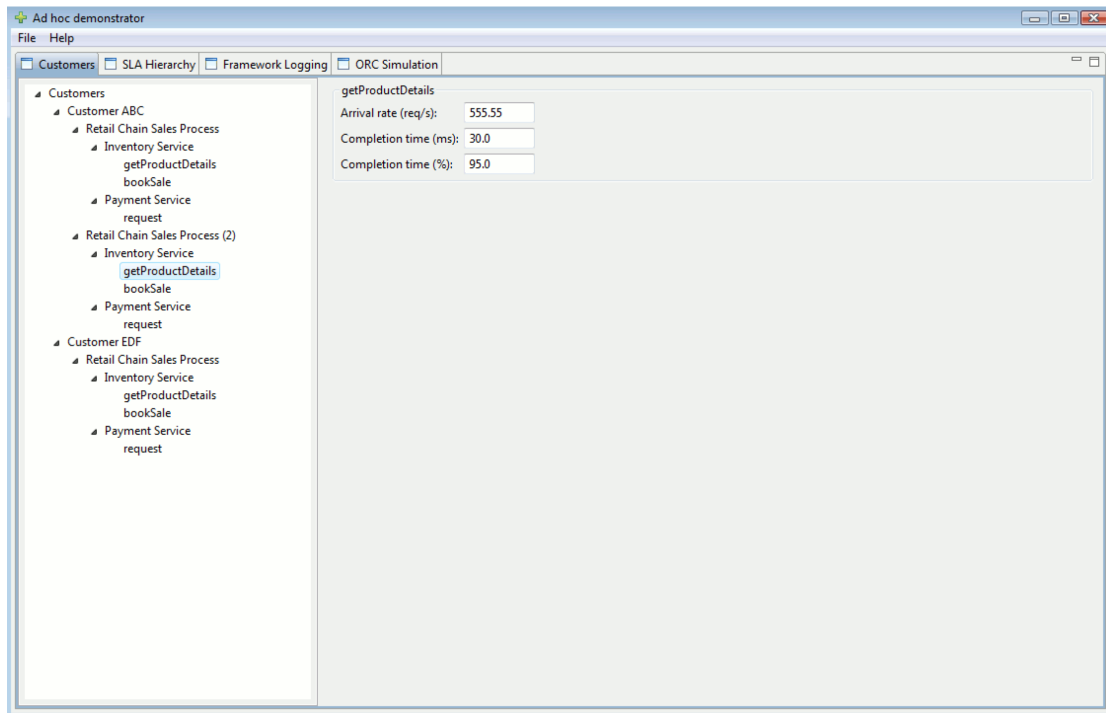

Figure 31: Operation view.

Customer is allowed to modify parameters for three operations: getProductDetails, bookSale (both under Inventory service) and request (under Payment service), which handles card validation and payment debit operations.

The actual SLA depends only on operations terms, i.e., it has no notion about number of cash desk or arrival rate for cash desk. Parameters from Usage Profile group box are just input data for calculating the actual SLA parameters.

The three pre-defined templates and corresponding SLA parameters are listed in Table 6:

Table 6: Pre-defined template values.

	Gold	Silver	Bronze
number of cash desks	1000	250	40
max. consumer arrival rate for cash desk	20	10	5
number of goods per consumer	10	10	10
completion time	95%	90%	85%

Operations parameters are calculated based on formulas in Table 7 (note that the completion time expressed in milliseconds is independent of Usage Profile parameters – there are pre-defined fixed values for gold, silver and bronze template).

Table 7: Formulas for operations parameters.

Operation	Arrival rate (requests per	Completion Time	Completion Time

	second)	(percentage)	(milliseconds)
getProductDetails	(number of cash desks * consumer arrival rate * number of goods) / 3600	completion time as defined in Usage Profile, but can be changed for each operation in arbitrary manner predefined values: 95 – Gold 90 – Silver 85 - Bronze	defined in arbitrary manner – does not depend on Usage Profile predefined values: 30 – Gold 70 – Silver 30 - Bronze
bookSale	(number of cash desks * consumer arrival rate) / 3600	completion time as defined in Usage Profile predefined values: 95 – Gold 90 – Silver 85 - Bronze	defined in arbitrary manner – does not depend on Usage Profile predefined values: 120 – Gold 180 – Silver 120 - Bronze
request	(number of cash desks * consumer arrival rate) / 3600	completion time as defined in Usage Profile predefined values: 95 – Gold 90 – Silver 85 - Bronze	defined in arbitrary manner – does not depend on Usage Profile predefined values: 70 – Gold 120 – Silver 70 - Bronze

In the Table oben *consumer arrival rate* refers to the *Arrival rate for cash desk* parameter that can be specified in the product root item view (Figure 30).

Independent of the three pre-defined options customer can modify parameters in an arbitrary manner. The purpose of pre-defined options is only to provide an initial guidance for reasonable parameters set. As soon as some modification is made in the Usage Profile parameters the Service Level Agreement template is set to Custom value and operations parameters are recalculated. When operations parameters are modified manually the Usage Profile parameters are cleared (the actual SLA is independent of them) and the template is set to Custom.

Customer can compare custom values with a predefined templates by browsing through wizard which can be opened by clicking one of the Gold, Silver or Bronze button in the Select SLA Template group box (Figure 32).

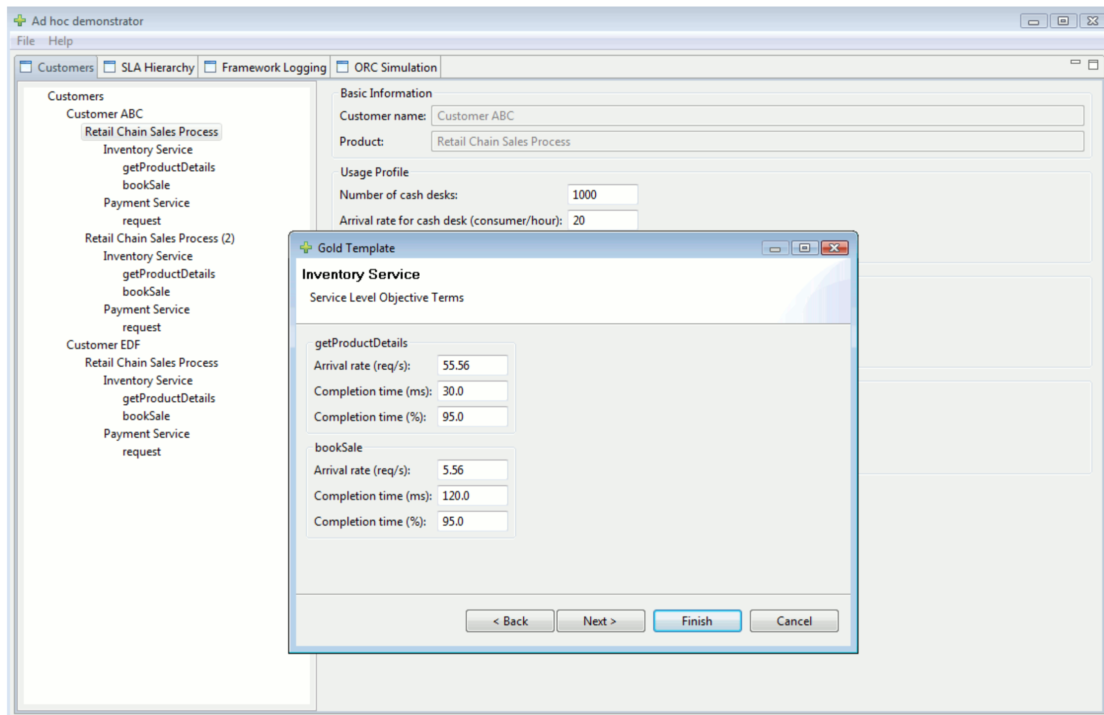

Figure 32: SLAT Browser.

Fully defined Service Level Agreement offer is sent to the framework by clicking on a Send button in a Propose SLA group box. This invokes proposeSLA function in eContracting module, which starts negotiation and provisioning phases. After these phases are concluded eContracting returns to adhoc demonstrator SLAResponseInfo object, which contains established Business SLA or the information why establishing of Business SLA failed.

Wizard which is similar to the one on the Figure 32 is opened when the Established SLA is returned to adhoc. It serves for browsing through the established SLA parameters, through price information and information about the date and time, when the customer can start using the product. In case the establishing Business SLA failed, only message box with an explanation is opened.

3.2.2 Service provider view (SLA hierarchy and SLA violation visualisation)

The purpose of SLA Hierarchy panel is to provide a view on a status of Service Level Agreements and their possible violations. Framework modules establish number of SLAs within processing of Business SLA offer, i.e. Business SLA, Intermediate SLA, Inventory SLA, Payment SLA, Validate SLA, Debit SLA and Infrastructure SLA in the case of processing SLA offer for Retail Chain Sales Process product (Figure 33). Adhoc demonstrator visualizes the status of each of them along with the text descriptions of status changes.

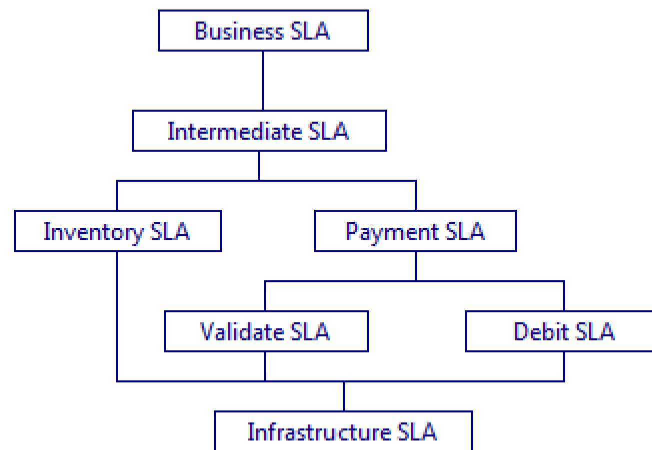

Figure 33: Retail Chain Sales Process SLA Hierarchy.

SLA Hierarchy panel has similar structure to Customers panel. Tree view on the left side provides navigation between customers and their products. For each customer's product exists a graphical interface for visualising the state of the Service Level Agreements on all layers (business layer, software service layer, infrastructure layer) and their relationship. The changes of state can be easily detected through framework SLA status messages and through hierarchy graph, which presents each SLA as a coloured node (Figure 34).

The semantics of node colors is:

- White: No SLA established yet
- Green blinking: SLA negotiation pending
- Green: SLA established and observed
- Red: SLA violated
- Yellow: at least one child element is violated

Infrastructure SLA can have only green or red color. When red, all Software SLAs switch to yellow.

Adhoc demonstrator retrieves most of the status messages from the framework over the message bus, used in the SLA@SOI framework. There is only one exception – since Business SLA is the main SLA in the hierarchy, business SLA violations require additional framework processing and confirming, which is done by the eContracting framework module after the violation events were triggered and sent to message bus. When business violation is confirmed it is sent to adhoc demonstrator using NotifyDeliveryStatus interface.

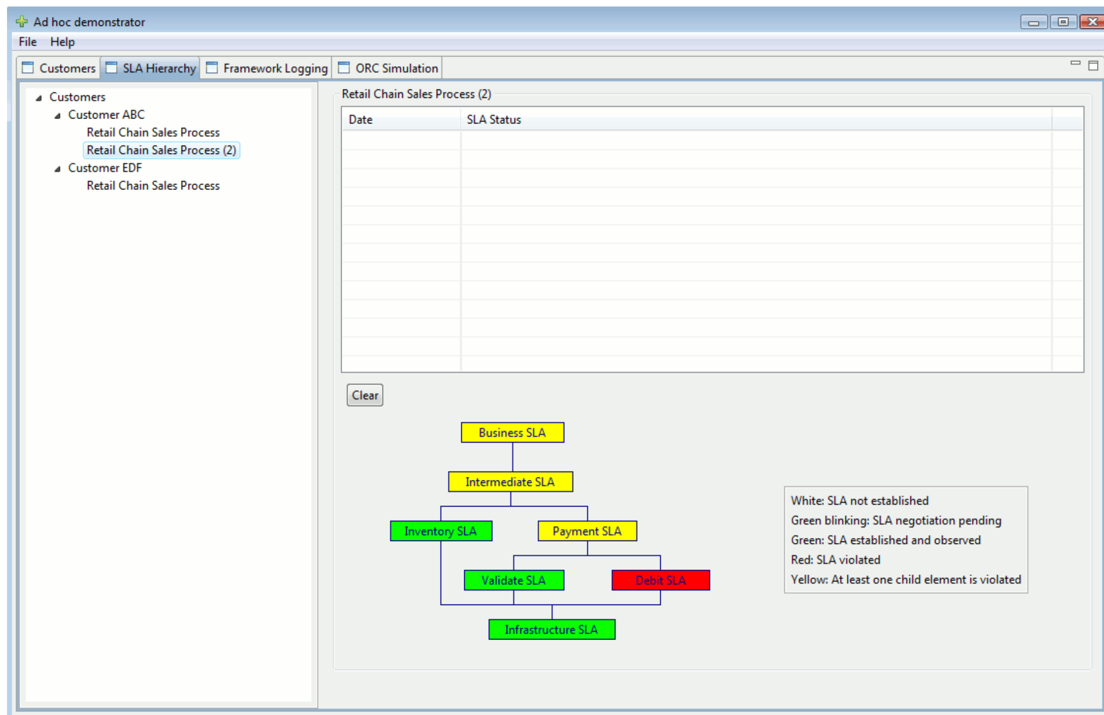

Figure 34: SLA Hierarchy view.

Other modules use SLAStatusMessage which contains a few simple parameters that has to be set before it can be sent to message bus. The parameters contain information about which SLA has a changed status, what is the new state, the date of the modification and short description about what happened.

The connection to message bus is configured by clicking Configure button at the root item of the tree view and setting the parameters in the Connection Details dialog window (Figure 35). The parameters are already offered to adhoc demonstrator user, but they can be changed. The parameters that have to be specified are:

- Channel: for this purpose channel with name ADHOC_SLA_MESSAGES is used
- Pubsub: xmpp (this and the following values are configured to use Intel testbed)
- Host: slasoi.dnsalias.net
- Port: 5222
- Service: slasoi.dnsalias.net
- Pubsub service: pubsub.slasoi.dnsalias.net

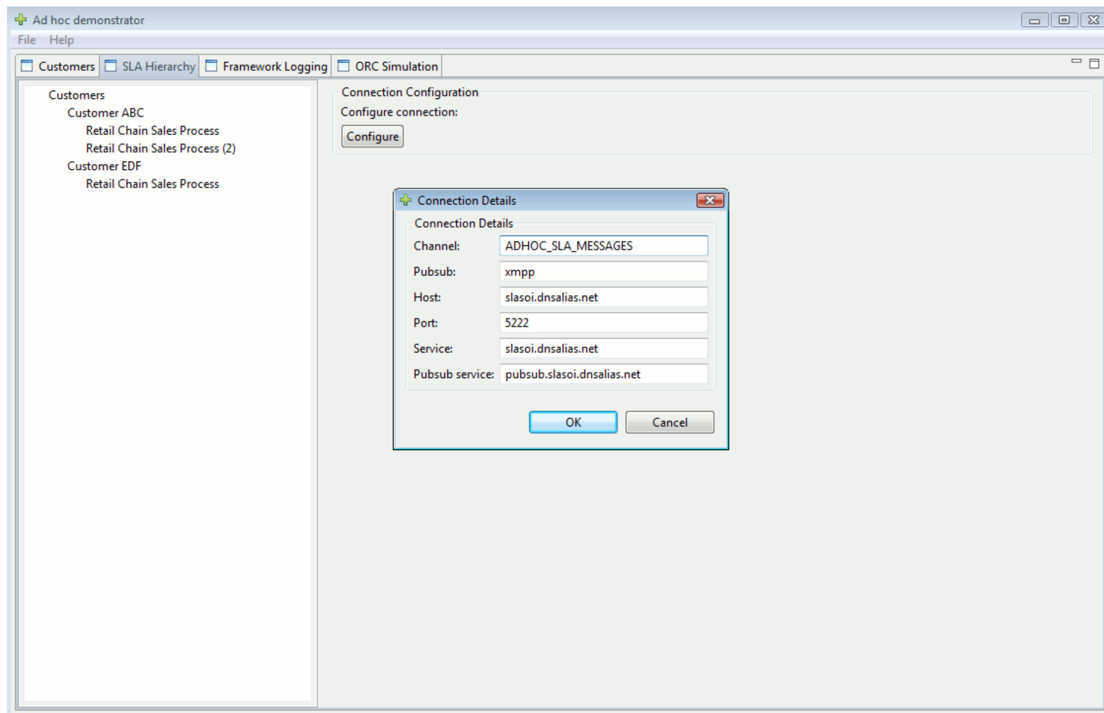

Figure 35: Connection settings for communication over message bus.

3.2.3 Workload manager console

ORC Simulation panel provides a console for simulating a running product. Retail Chain Sales Process product can be started, paused and stopped here. Different simulation profile parameters are available to be defined to enable simulating a wide range of different usage profiles.

First, the connection to message bus has to be configured. The Configure button which opens a Connection Details dialog window can be found at the root item of the tree on the left side of the panel (Figure 36). Connection Details dialog window is the same as in SLA Hierarchy panel and the parameters are offered too. The channel name for this purpose is ADHOC_WORKLOAD_CHANNEL.

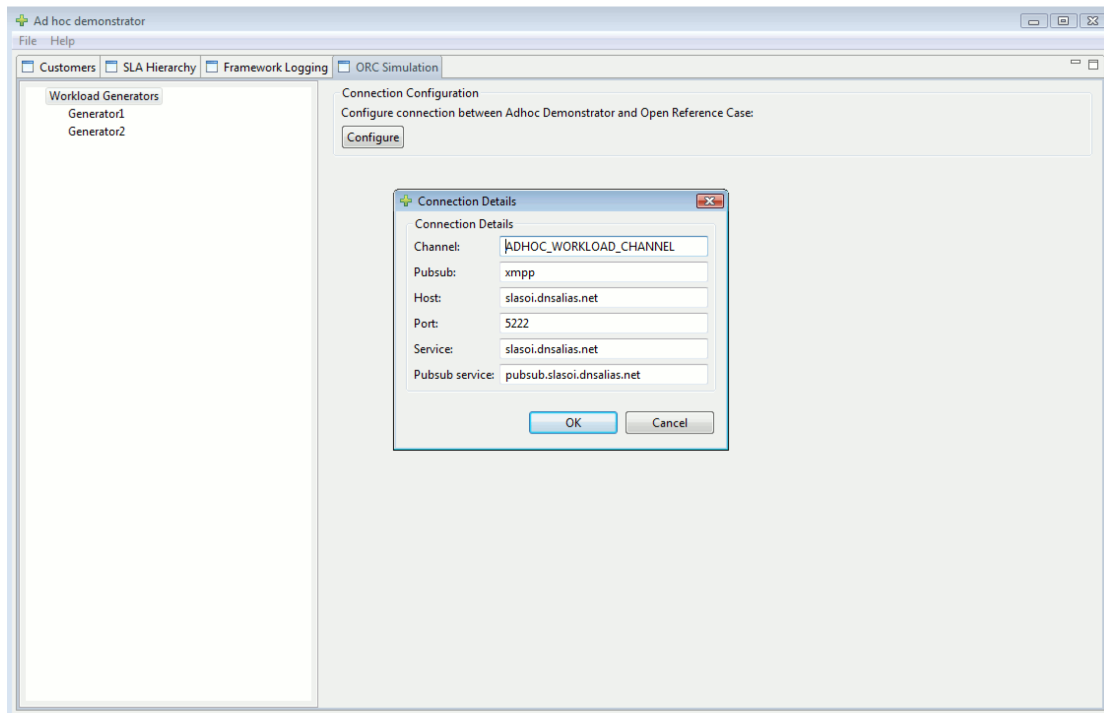

Figure 36: Connection settings for communication between adhoc demonstrator and Simulation Environment over message bus.

After the connection is set, an underlying daemon starts listening to heart-beat pings from workload generators. These messages are sent by workload generators each second to a message bus. Workload generator is a Java process which simulates a defined number of running cash desks and is waiting to be configured and started from the adhoc demonstrator.

Workload generator is added to Workload Generators tree on the left side of the panel immediately after the first heart-beat message is detected. By clicking on a generator item in the tree view the simulation profile configuration panel is provided on the right side of the panel (Figure 37). The customer and its product to be run have to be specified for a generator. Then, Base URI (which is a Retail Chain Sales Process product endpoint) value is set – it serves as an information where the workload generator can find web services to invoke.

The actual simulation parameters are to be specified in three group boxes: Simulation Profile, Simulation Delays and Deviations and Simulation Probabilities group box. Some of parameters are already configured with default values, which are meant to serve as initial help when configuring simulation profile and can be modified in an arbitrary manner. All simulation parameters are independent of customer's profile that was defined in the negotiation process between customer and SLA@SOI framework. This independence is important to enable simulating really frequent invocations of product services and causing SLA violations, which can be then traced in the SLA Hierarchy panel.

Figure 37: Simulation configuration.

In the Simulation Profile group box three parameters have to be specified: Number of cash desks, Number of bought products and Random seed. The first parameter is simply a number of concurrent threads that simulate operating cash desks (when workload generator is a Java process, the operating cash desk is a Java thread), the second is a number of products that are bought by each simulated cash desk consumer and the third is a number that is used for generating an array of random numbers from which the deviations of the simulation delays (Simulation Delays and Deviations group box) are calculated.

The next two group boxes contain configuration parameters for each cash desk thread.

In the Simulation Delays and Deviations group box several delays for different ORC function calls have to be specified. Actual delays are calculated in workload generator process and can differ from a specified value based on the deviation parameter value. Delays are calculated using random array that was mentioned before and parameters can be therefore exactly the same when reconfiguring or when configuring some other generator if using the same Random seed. This way the SLA violations can be reproduced and an in-depth analysis of the SLA@SOI framework is possible by SLA Hierarchy and Framework Logging panels.

The quick explanation of the delays (all delays and deviations have to be specified in milliseconds):

- Wait for customer delay is the delay before processing the next customer at the cash desk (after simulation is started the customers are coming to the cash desks in the infinite loop)
- Scan delay is the delay before product barcode is scanned and product details are retrieved
- Capture manually delay is the delay before the product details are retrieved manually in cases when barcode scanning fails

- Update running total delay is the delay before status of the processed products is updated
- Give card delay is the delay before payment card processing starts
- Manual payment is the delay before the products are paid in case of cash payment
- Wait to process is the delay before sale booking is called

In the Simulation Probabilities three probabilities are to be defined (values between 0 and 1):

- Data recognized probability is the probability that data on an article is recognized when barcode scanning
- Payment with Credit Card is the probability that payment is done with Credit Card and not with cash
- Card valid probability is simply the probability that card is valid and that the payment can be done with it

At first, only Configure and Show diagram buttons are enabled in Simulation Management group box. By clicking on a Configure button the simulation parameters are serialised into JSON format and sent over the message bus to the selected workload generator. Upon receiving this message, the generator configures an array of workload agents and waits for commands. A few changes in adhoc demonstrator indicate that generator is configured - the specified number of cash desks is added to the generator item in the tree view, parameters fields are fixed and disabled for modifying, Configure button switches to Clear button and three additional buttons (start, pause, stop) get enabled (Figure 38). They provide a control over starting, pausing and stopping the cash desk threads.

A single workload generator can create arbitrary number of workload agents that all share the same workload characteristics. Generator can be reconfigured at any time. By clicking on a Clear button all generator parameters are cleared, and the button can again be used for workload configuration.

By clicking on a Show diagram button the dialog window with a diagram of ORC process with all functions and delays is opened to provide a better view on a cash desk workflow.

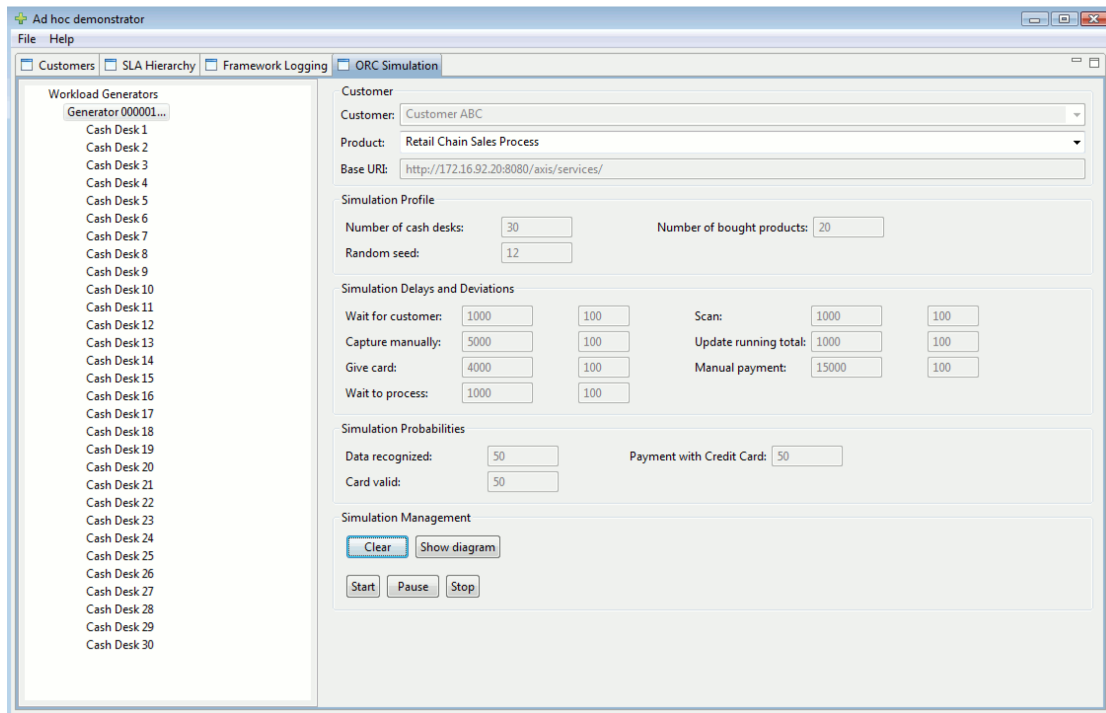

Figure 38: Simulation panel after workload generator configuration.

Whereas buttons in the Simulation Management provide the control over all cash desks, it is possible to control a single cash desk as well. By clicking on a cash desk item in a tree view, the dialog window opens and provides start/pause/stop actions for a specific cash desk and a view over generator messages about executed actions and status of the sales process (Figure 39). An arbitrary number of cash desks windows can be opened at the same time.

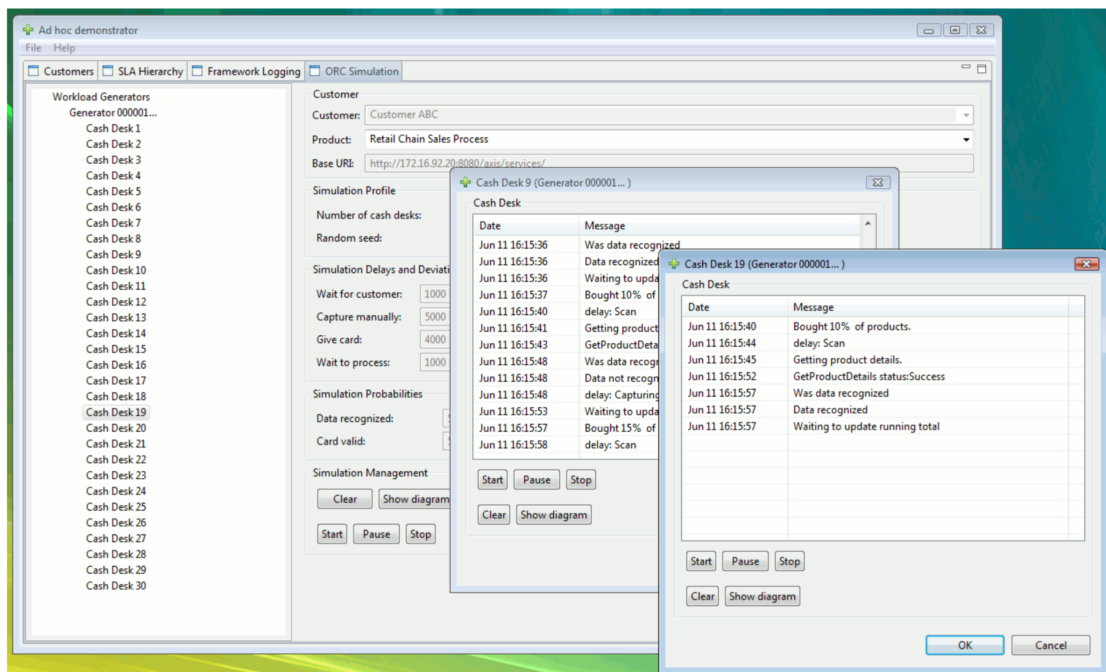

Figure 39: Cash desk control view.

By clicking on a Show diagram button the dialog window with a diagram of cash desk process is opened and the tracing of the currently invoked functions is possible by a moving red square.

3.2.4 Framework logging

Framework Logging panel provides a view on a current status of SLA@SOI framework modules. It is possible to trace in a real time which methods are invoked and their descriptions (Figure 40).

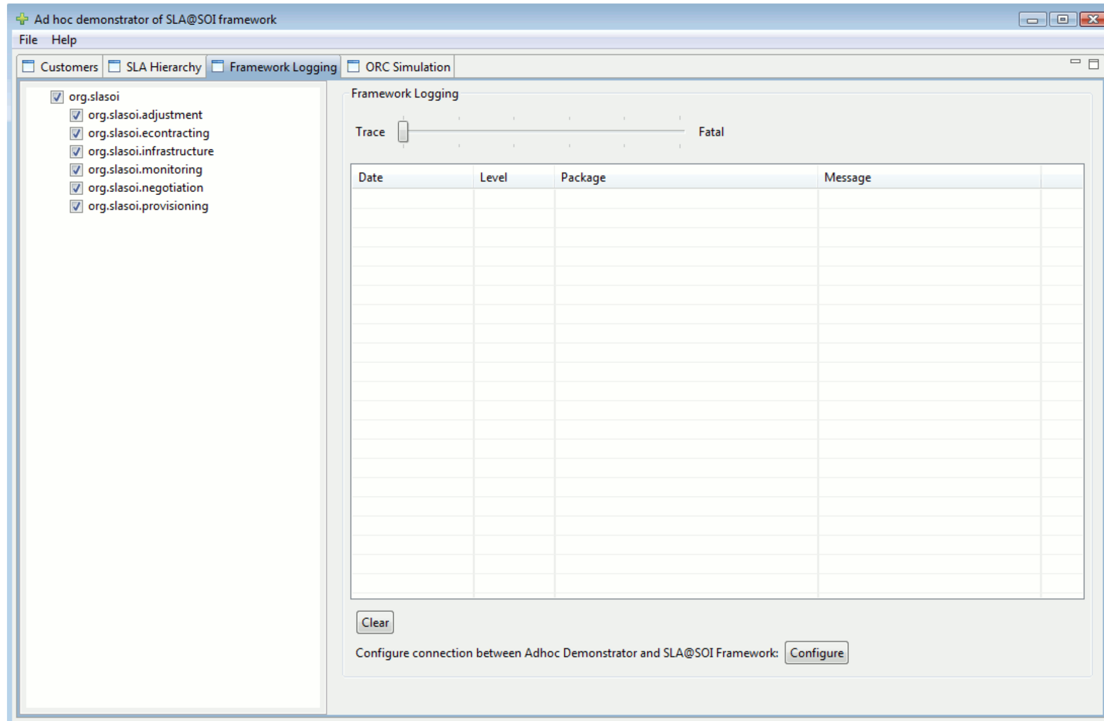

Figure 40: Framework Logging view.

Again, there is a tree view on the left side of a panel, which is used for filtering logging messages. The adhoc demonstrator user can enable or disable messages from an arbitrary framework module or modules by selecting check boxes. The tree items are actually the main framework Java packages that correspond directly to framework modules. Additionally, during the framework runtime Java sub-packages are dynamically added into the tree structure to enable filtering by a specific sub-package as well.

Logging messages are shown in the table on the right side of the panel in the format Date/Level/Package/Message. The slider above the table allows users to filter messages by logging level as well. Possible log levels are:

- Trace
- Debug
- Info
- Warn
- Error
- Fatal

Technically, log4j logging framework [18] is used for logging messages. Each framework module source code contains log4j messages of different level and includes a specialised log4j appender. This appender translates every log message into a message structure and sends it on the central logging message bus. By appender-0.0.4.jar and log4j configuration file all log4j messages are sent. Adhoc demonstrator retrieves these messages from the bus and shows them in the table.

Connection to message bus needs to be configured by clicking on Configure button and specifying the parameters in the Connection Details dialog window (Figure 41). The channel name for this purpose is ADHOC_LOGGING_CHANNEL.

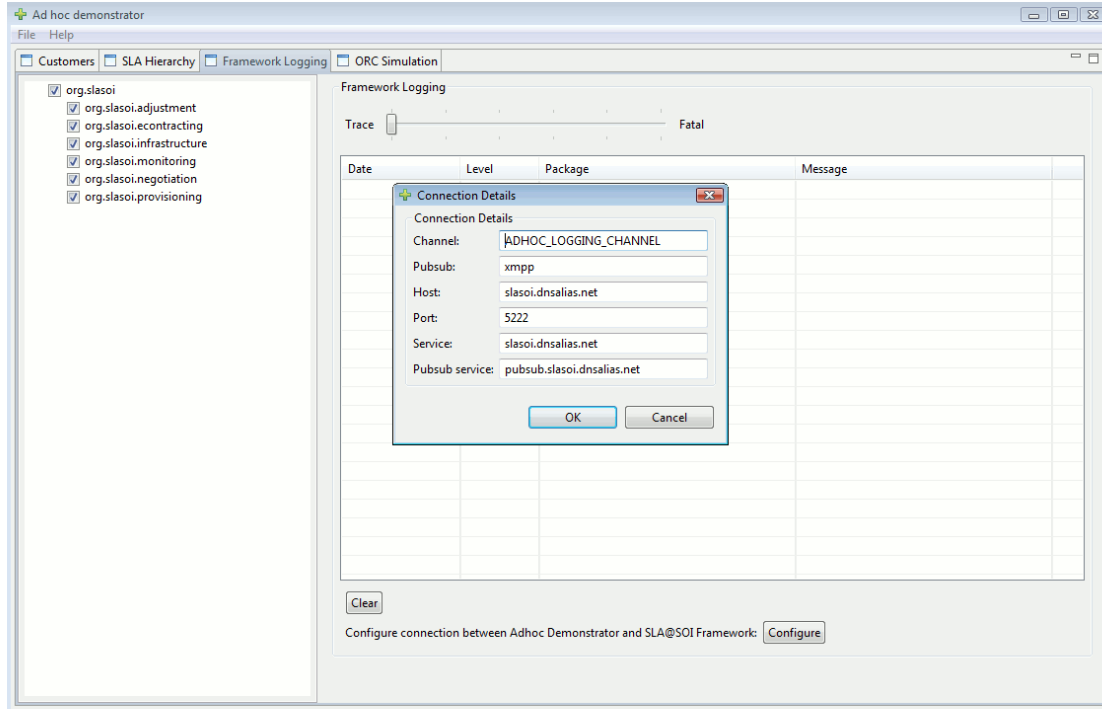

Figure 41: Connection settings for communication over message bus.

4 Architecture of the Adhoc Demonstrator

The first public demonstrator of the SLA@SOI project is based on a simplified service-oriented retail chain application. The latter was the main driving force for the design and implementation of the overarching adhoc architecture. A component diagram comprising of all framework components, adhoc demonstrator and the service-oriented application is depicted in Figure 42. The boundary between the SLA@SOI framework and the demonstrator is mainly represented by interfaces provided by the E-Contracting and the adhoc demonstrator components. The former allows listing of available SLA templates (gold, silver, bronze) for the product offered by the service provider, and proposing the SLA offer. The status the SLA can be queried through the use of the QuerySLAStatus interface.

On the other hand, the adhoc demonstrator component provides a single interface used by the E-Contracting component to notify the demonstrator (i.e., the customer) that the SLA offer was processed. The adhoc demonstrator also provides a message channel as a logging interface for framework components to send their internal status messages (debugging purposes). Service Application (ORC) provides the service-oriented application via RetailSolution interface. Adhoc demonstrator also communicates with the Infrastructure component to simulate various workload scenarios, e.g., launching I/O, CPU and/or network intensive operation, simulating VM failures. The management interface is not part of the public framework.

In Figure 42 some of the interfaces are marked as *offline interfaces*. These interfaces are used in the preparation phase during which the framework and the retail chain application are initialised. They are not intended to be used during the run-time phase.

E-Contracting component provides additional interface for the Adjustment to notify of a detected Business SLA violation (BusinessViolation). It further uses two interfaces from the Negotiation component: first is used to get the list of templates (GetTemplates) while the other (CreateAgreement) forwards the request from the customer to the Negotiation component. SLAs are stored using the StoreSLA interface of the Provisioning componennt. Monitoring component is initialized with a call to the StartMonitoring interface. This interface requires negotiated SLA from which monitoring rules are built. EventBus component is the central component used to transfer messages between loosely coupled components. It's only public interface is PushEvent. The infrastructure component provides interfaces for making reservations and re-provisioning upon detected SLA failures.

The main three phases, namely the negotiation, the provisioning, and the run-time (monitoring) phase, are described using sequence diagrams in Figure 43, Figure 44 and Figure 45. From the diagram in Figure 43 it is evident that the customer only communicates with the E-Contracting component to retrieve the list of supported templates and to start the negotiation based on chosen SLA offer. Negotiation is responsible for the translation of the SLA hierarchy, initialisation of the run-time schedule and appliances and for making the reservation on the infrastructure. Once all components are aware of the newly

created SLA, the E-Contracting component can calculate the price of the offer and return it to the customer.

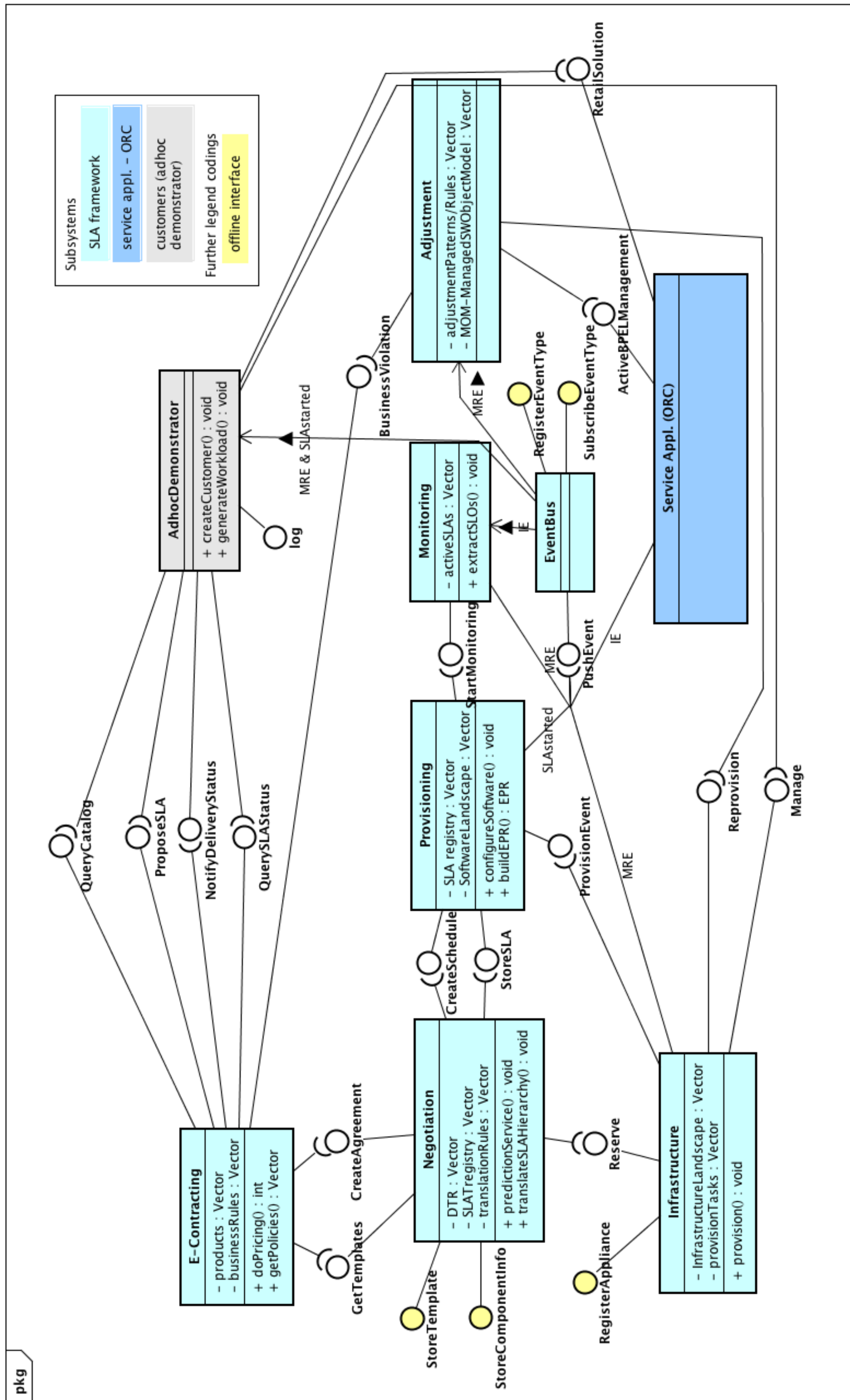

Figure 42: The component diagram showing the adhoc architecture.

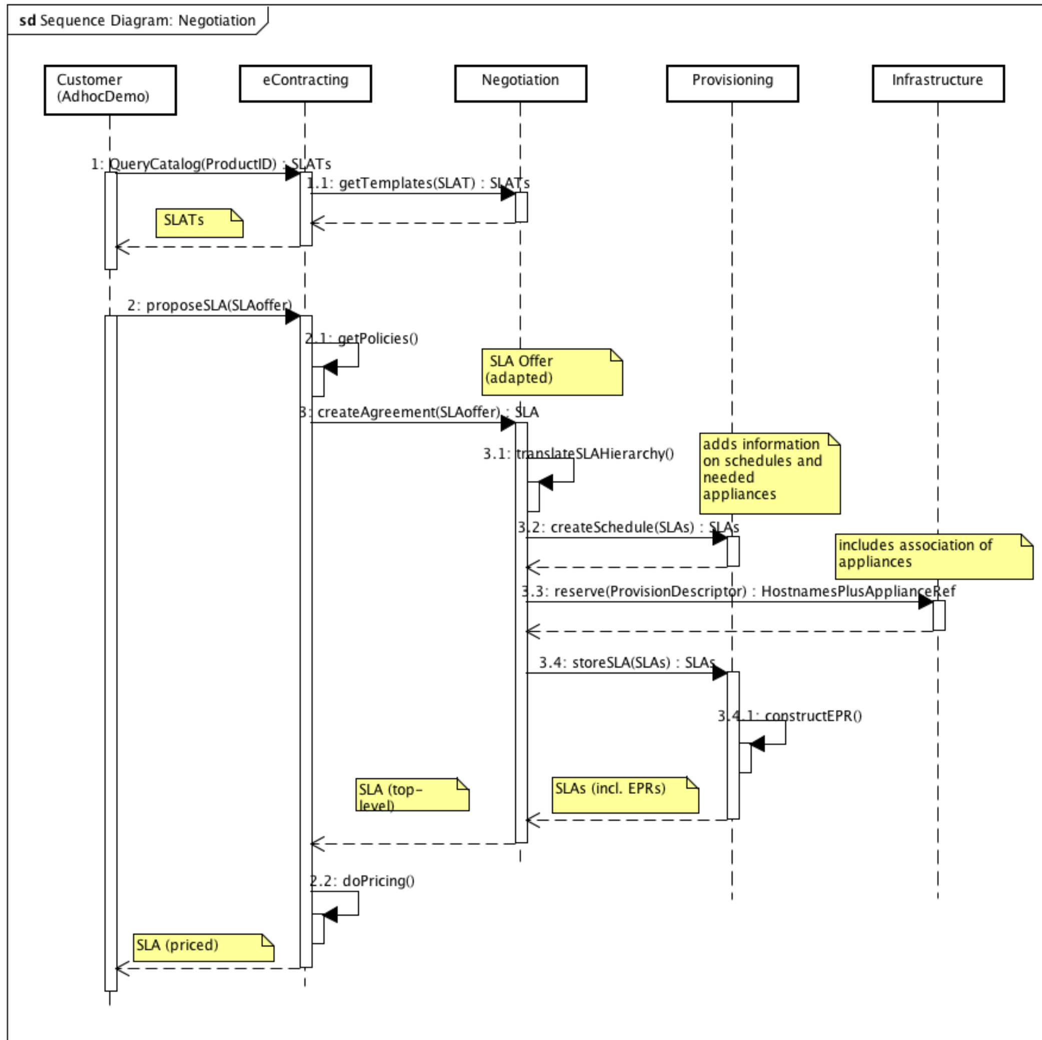

Figure 43: Sequence diagram during the negotiation phase.

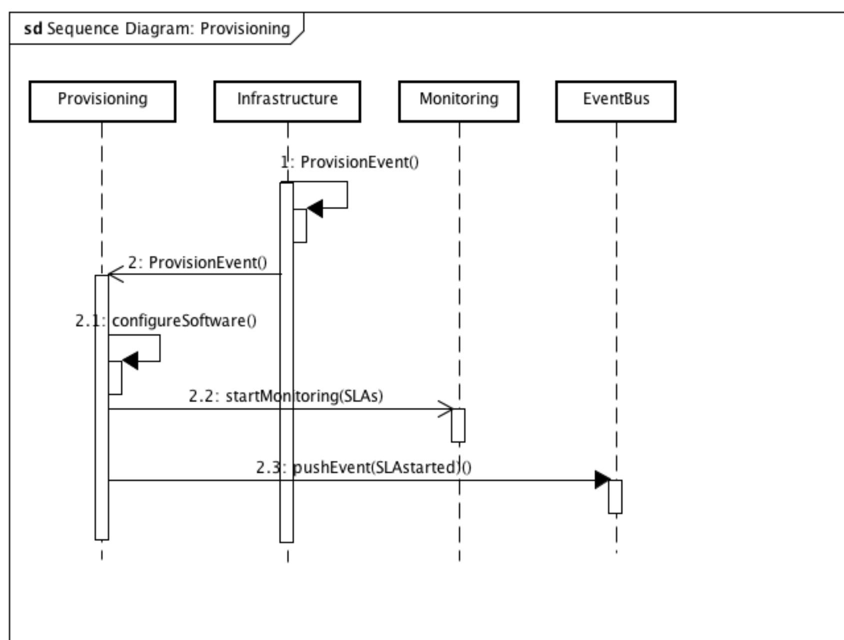

Figure 44: Sequence diagram of the provisioning phase.

The provisioning (Figure 44) is responsible for the configuration of the running software and initialisation of the monitoring component. Events are exchanged between components using the event bus. The Monitoring component detects violations in the software, e.g., the Service application and/or the middleware components (application server, database, ...). Infrastructure uses internal monitor to detect violations and sends them directly to the Adjustment Component that notifies the E-Contracting component. Business violation is then sent to the customer (the adhoc demonstrator) as well. Adjustment also instructs the Infrastructure and the Service Application to re-provision and reconfigure the BPEL engine, respectively.

For a detailed description of these diagrams refer to public SLA@SOI deliverables.

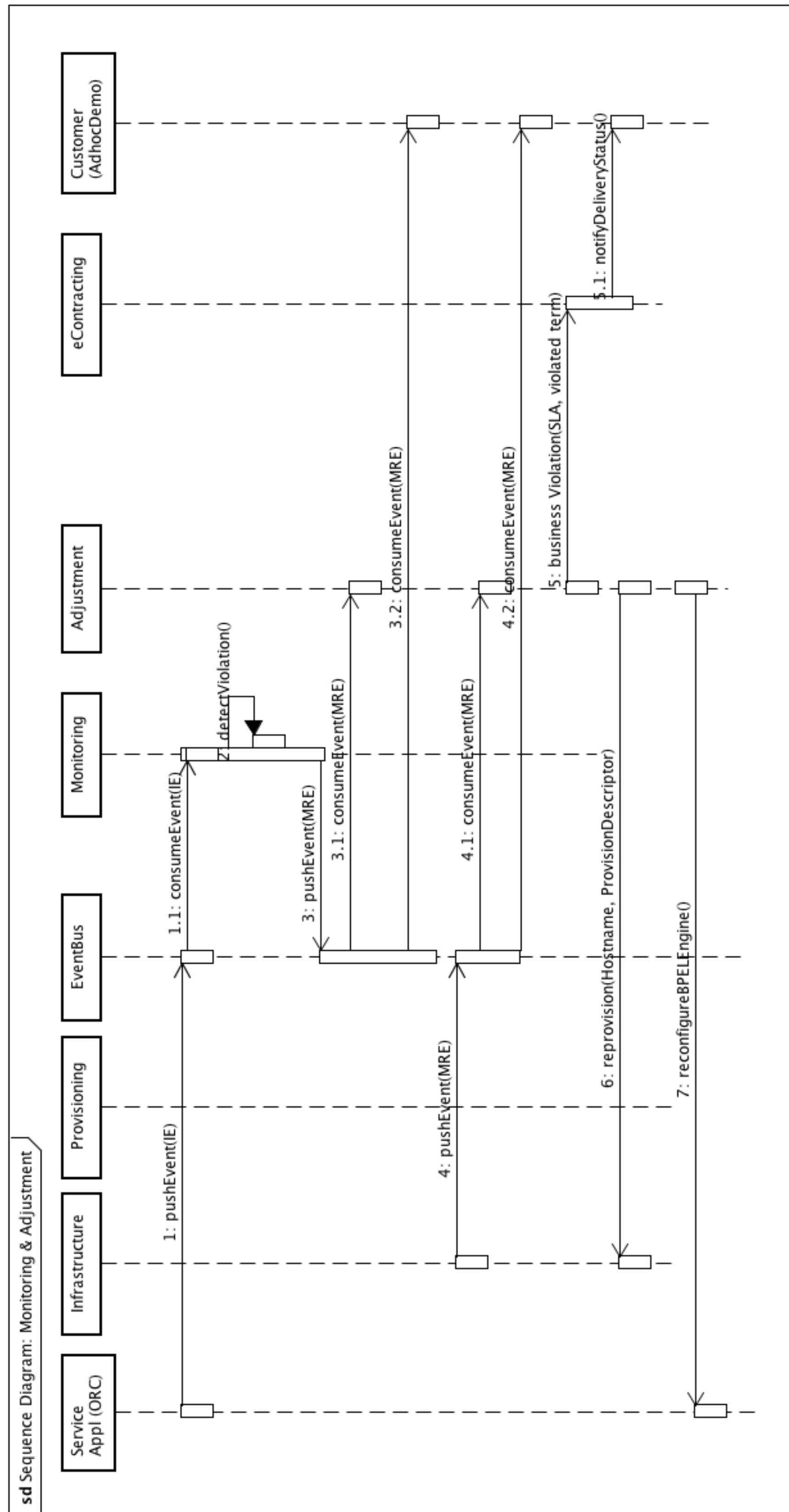

Figure 45: Sequence diagram representing details of the monitoring and the adjustment component.

5 Evaluation plan

The evaluation plan presented in this section forms the basis for the evaluation of the adhoc demonstrator as well as the reference demonstrator. The results of the adhoc demonstrator's evaluation at the end of year one, is the starting point for the development of the reference demonstrator. Using the same evaluation plan for the evaluation of the reference demonstrator at the end of the project, allows the comparison of the evaluations and thereby supports the demonstration of the improvements implemented in reference demonstrator. The plan is based on the Goal/Question/Metric (GQM) approach proposed by Basili, Caldiera, and Rombach [19], which is summarized first. Then we present the concrete instantiation of a GQM plan which will be used to evaluate the adhoc demonstrator.

5.1 The Goal/Question/Metric Appraoch

The Goal/Question/Metric (GQM) approach proposed by Basili, Caldiera, and Rombach is a process model for measurements targeting a particular set of issues (goals) and a set of rules for the interpretation of the measured data. In order to be meaningful, measurements must be goal-oriented and, thus, are defined in a top-down fashion. Basili, Caldiera, and Rombach argue that measurements, which are not performed in a goal-oriented way, are likely to be inefficient. The absence of concrete goals carries the risk of collecting large amounts of unnecessary data. Large amounts of data and missing goals may complicate the interpretation of measurements.

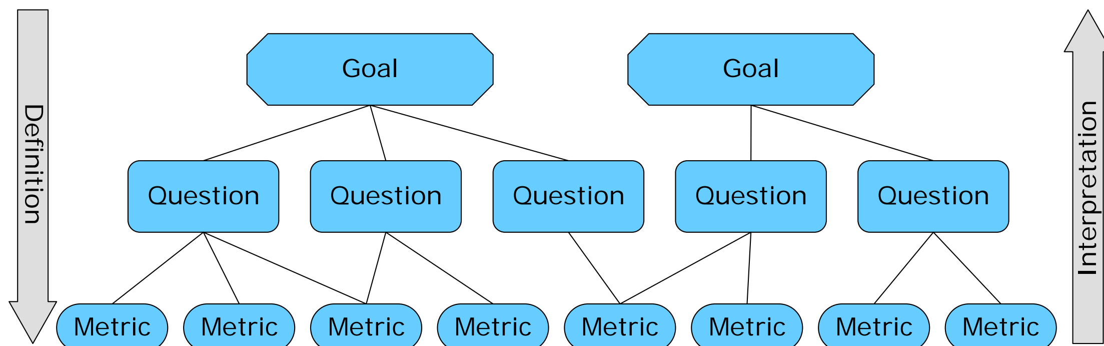

Figure 46: Relations between goals, questions, and metrics

The GQM method starts with the explicit definition of a measurement goal. Several questions serve to refine the goal and to identify its major components that need to be answered by the measurements. Questions are further refined by metrics. Figure 46 depicts the relation between goals, questions, and metrics. When the measurements for each metric have been taken, the resulting data is interpreted bottom up. Each metric is directed towards specific questions. The collected data answers the questions with respect to the goal. This evaluation allows deciding whether the goal has been attained or not. Figure 46 further indicates that the same metric can answer different questions.

In GQM, **goals** are located on a conceptual level. They strongly depend on the context in which measurements take place. The context subsumes the objects, the reasons, the points of view, and the environment of the measurements, as well as the considered models of quality. Possible objects of measurements are products (artefacts, deliverables, or documents), processes (software related activities), or resources (e.g., personnel, hardware, or software). To correctly embed a goal into a given context, the GQM method requires the explicit

definition of the goal's issue, object or process, viewpoint, and purpose. **Questions** determine the assessment of a specific goal. They characterise the object of measurement (product, process, resource) with respect to selected quality attributes from the selected viewpoint. On a quantitative level, **metrics** associate a set of data to each question. The data answer the questions in a quantitative way. In GQM, there exists a distinction between objective and subjective metrics. While objective metrics depend only on the object under measurement (e.g., lines of code), subjective metrics depend on the viewpoint from which the measurements are taken (e.g., readability of a text). When selecting metrics, various factors have to be considered. Basili et al. [BCR94] summarise the most important ones as follows:

- **Amount and quality of existing data**: To minimise the effort during data collection, the use of existing data sources can be maximised.
- **Maturity of the objects of measurement**: Objective metrics are preferable for more mature measurement objects, while subjective evaluations are better suited for informal or unstable objects.
- **Learning process**: GQM plans need iterative refinement and adaptation. The defined metrics have to evaluate not only the object of measurement but also the reliability of the model in use.

5.2 GQM-based Evaluation Plan

The evaluation plan for the adhoc demonstrator and the reference demonstrator, which bases on the aforementioned GQM approach, comprehends the goals described in Table 8. The goals represent requirements from the business perspective as well as requirements from a developer and service provider perspective. These goals are broken down into several questions. For each Question a number of Metrics are defined, that allow answering the respective question.

Table 8: Goals, Questions, and Metrics.

G1	Support of the service lifecycle by the SLA framework in the open reference use case	
	In deliverable D.A1a [2] a number of scenarios are defined that should be supported by the SLA framework. In the open reference use case some of these scenarios are instantiate. This goal focuses on the evaluating which of these base scenarios are supported by the adhoc demonstrator. The questions are therefore aligned with the provisioning lifecycle presented in Section 2.2.	
	Q1.1	Are the software and service provider supported by the SLA framework in offering new or modify service offers?
	M1.1.1	Is there an interface for service providers to deploy/install software on the virtual images or update software on these images?
	M1.1.2	Level of automation of the steps required to install the ORC.
	M1.1.3	Level of automation of the steps required to update the ORC.
	Q1.2	Is the service provider supported by the SLA framework in provisioning the ORC for a new customer?
	M1.2.1	Is there an interface for service providers to create and start new instances of the ORC image?

	M1.2.2	Is it possible to run several independent instances of the ORC?
	M1.2.3	Is there an interface for service providers to see which and how much instances of the ORC are running?
	M1.2.4	To what extend is the creation and start of new instances of ORC images automated?
	Q1.3	Are the customer and the service provider supported by the SLA framework in offering, selecting and negotiating SLAs.
	M1.3.1	Is there an interface for customers to browse and select offered services and the associated SLA offers?
	M1.3.2	Is there an interface for service providers to adapt SLA offers respectively SLA templates?
	M1.3.3	To what extend is the selection and negotiation of SLAs automated from the customer's perspective?
	M1.3.4	To what extend is the negotiation of SLAs automated from the service provider's perspective?
	M1.3.5	To what extend is the adaptation of SLA offers / SLA templates automated from the service provider's perspective?
	Q1.4	Does the SLA framework support the service provider in operating the ORC and detecting SLA violations
	M1.4.1	Are there events for SLA violations on the infrastructure level?
	M1.4.2	Are there events that inform the infrastructure providers about a forthcoming SLA violation?
	M1.4.3	Are warning events on the infrastructure level are used to enhance the SLA monitoring on the software level?
	M1.4.4	Are there events for SLA violations on the software level (services of the ORC)?
	M1.4.5	Are there events for SLA violations of business SLAs?
	M1.4.6	Provide SLA violation events sufficient information to identify the (root) cause of the violation?
	M1.4.7	Are there events for SLA violations from the customer's side?
	M1.4.8	Does the SLA framework provide an overview on the currently active SLAs?
	Q1.5	Does the SLA framework support the infrastructure provider in adapting the resources to fulfil the guaranteed SLA if required?
	M1.5.1	Is there an interface that allows to change the resources associated to the virtual machine running the ORC?
	M1.5.2	To what extend is the adjustment of resources for virtual machines automated?
	M1.5.3	Is there an interface that allows the infrastructure provider to monitor resource utilization of virtual machines running the ORC?
G2	Performance of the Demonstrator	

		The demonstrator more precisely the SLA framework consists of several interacting modules. To estimate the overall performance of the demonstrator and the SLA framework, it is necessary to know the performance of the individual modules as well as the dependencies between modules.
	Q2.1	What is the response time of a module? (not all metrics are adequate for every module)
	M2.1.1	Method invocation response time (e.g. time required to perform a prediction and return the results)
	M2.1.2	Reaction time on events (e.g. how long does it take to recognize an SLA violation, after the last event was received)
	M2.1.3	Processing time of events (e.g., time span from service call to respective monitoring event, time to generate new aggregated events)
	M2.1.4	Processing time of actions (processes) (e.g., time required by the provisioning module to provide required resources)
	Q2.2	Does the response time depend on external influence factors?
	M2.2.1	List of the modules whose response time influences the response time of the module under consideration
	M2.2.2	List of parameters influencing the performance (e.g., size of VM images, size of prediction model)
	M2.2.4	Term to calculate the expected response time (including the external influence parameters) (e.g., RT=internalCalculation + externalService or RT= max(internalCalculation,externalCalculation))
	Q2.3	What is the maximal throughput of a module? *(not all metrics are adequate for every module)*
	M2.3.1	How much events/service calls can be processed per minute by the module?
	M2.3.2	How much events/service calls are produced by the module per indoming event/service call?
	M2.3.4	How much events can be processed by the message bus?
	Q2.4	Is the maximal through limited by external influence factors?
	M2.4.1	List of the modules whose throuput limits the throughput of the module under consideration
	M2.4.2	List of parameters influencing the throughput (e.g., Message Type, size of VM images, size of prediction model)
G3	Extensibility of the Demonstrator	
	The open reference case scenario covers only a small subset of requirements. As the SLA framework should support different scenarios, new requirements on the framework will arise. Therefore, this goal focuses on the evaluation of the extensibility of the demonstrator regarding new requirements that might result from extensions and	

	adaptations of the open reference use case. Potential change requests are: 1. Several independent customers need to be served simultaniously. 2. A new business level SLA (e.g., diamond) should be added. 3. The interpretation of a specific business level SLA needs to be changed. 4. New SLA parameters should be supported (e.g. reliability/ availability). 5. New services are added to the ORC 6. Dependencies between services are changed (e.g., Payment-Service also requires Inventory Service) 7. New deployment options are added (e.g., separated Comosite-Service)	
	Q3.1	What has to be changed within each module to support the change requests (see G3) within the ORC scenario? (Answers for each request seperately)
	M3.1.1	List of affected classes
	M3.1.2	Description of necessary implementation steps
	M3.1.3	Required changes in other modules
	M3.1.4	Sequence (path) of changes
	M3.1.5	Changes of interfaces
	M3.1.6	List of impacted files and documents
	Q3.2	Which effort is required to implement these changes?
	M3.2.1	Affected Lines of Code
	M3.2.2	Estimated effort in person days
	M3.2.4	Number of methods that need be adjusted
G4	Usability of the Framework from a customers' point of view	
	The customer satisfaction is an important aspect for the success of a product or service. As the SLA framework is directly used by customers its usability heavily influences the satisfaction of the service provider's customers.	
	Q4.1	What is the complecity of negotiating a new SLA?
	M4.1.1	Number of required parameters (customer→framework)
	M4.1.2	Number of interaction steps (customer→framework)
	M4.1.3	Success rate of the negotiation process
	Q4.2	What is the complecity of re-negotiating an SLA?
	M4.2.1	Number of required parameters (customer→framework)
	M4.2.2	Number of interaction steps (customer→framework)
	M4.2.3	Success rate of the re-negotiation process
	Q4.3	What is the response time of the framework for customer request
	M4.3.1	Time to respond on an new SLA request (Acknowledge)

	M4.3.2	Time to provision of the runnning software
	M4.3.3	Time to react on a re-negotioation request
	M4.3.4	Time to re-provision the software
	M4.3.5	Time until SLA violations are reported

6 Results of evaluation (FZI/XLAB)

For the Y1 evaluation we developed a questionnaire that covers most of the questions and metrics defined in the evaluation plan. The adhoc demonstrator is a first prototype of the Y1 SLA management framework whose aim is to demonstrate the functioning of the framework in the context of the ORC use case. As the adhoc demonstrator is a proof of concept only, the conducted Y1 evaluation focuses on the fulfilment of requirements on the SLA framework in the ORC scenario and neglects e.g. the usability of the SLA Management framework. Moreover, the evaluation of the adhoc demonstrator prototype is the basis for the planning and development of the reference demonstrator, thus the evaluation focuses on implementation details from a developer's point of view and for example do not evaluates the framework (e.g. the usability) from the customer's point of view. The prototypical state of the SLA framework does not allow a fair evaluation of performance or usability. The questions were answered by the work package and module leads. In the following we present the summary of the provided answers:

6.1 ORC Service lifecycle supported by the SLA framework

In this first part of the evaluation we try to find out to which extend the adhoc demonstrator in particular the SLA framework already supports the service lifecycle of the ORC use case already presented in section 2.2.

6.1.1 Support to offer new or modified software components

Table 9: Support to offer new or modified software

Is there an interface for service providers to deploy/install software on the virtual images or update software on these images?	
Related Metric: M1.1.1	
(answered by WP A4 and WP B2)	
Answer	No
Interface Name:	-
Module Name offering this interface:	-
Described in deliverable:	Deployment of ORC is described in the appendix of D.B2
Short Description:	Only root access via SSH/SCP to images
Level of automation of the steps required to install the ORC.	
Related Metric: M1.1.2	
(answered by WP A4 and WP B2)	
Percentage:	0%
Automated Steps:	-
Manual Steps:	• Create a generic virtual machine • Configure Network setup for this VM

	• Configure the virtual machine to run under the managed A4 environment • Configure Image (previous steps..) • Upload Software • Install/Configure Software
Described in deliverable:	Deployment of ORC is described in the appendix of D.B2
Comments:	Only root access via SSH/SCP to images

6.1.2 Support for provisioning the ORC to a new customer

Table 10: Support for provisioning the ORC to a new customer

Is there an interface for service provider to create and start new instances of the ORC image?	
Related Metric: M1.2.1	
(answered by WP A4)	
Answer	Yes
Interface Name:	`org.slasoi.infrastructure.management.InfrastructureImp.provision` (This is the actual implementation)
Module Name offering this interface:	`org.slasoi.infrastructure.management`
Described in deliverable:	D.A4a
Comments:	ORC provisioning is just like any VM provisioning. Currently A4 does not implement image deployment interface, therefore we can only start one ORC instance at any given time. This is probable the domain of A3 (if we consider Software Management as VM image management)
Is it possible to run several independent instances of the ORC?	
Related Metric: M1.2.2	
(answered by WP A4)	
Answer:	No
Described in deliverable:	A4
Comments:	It is necessary to configure them manually, if the ORC is distributed over more than one VM and network communication is required. ORC needs to be aware that it can be run under different hostnames
Is there an interface for service provider to see which instances of the ORC are running?	
Related Metric: M1.2.3	
(answered by WP A4)	
Answer	Yes
Interface Name:	`org.slasoi.infrastructure.management.InfrastructureImpl.getDetails` (this is the actual implementation)
Module Name offering this interface:	A4 `.infrastructure.management`

Described in deliverable:	D.A4a
Short Description:	getDetails returns a copy of the current registry entry of the provisioning ORC images (like any provisioned VM) from the InfrastuctureLandscape
To what extent is the creation and start of new instances of ORC images automated?	
Related Metric: M1.2.4	
(answered by WP A4)	
Percentage:	Start 100% (A4)
Automated Steps:	
Manual Steps:	• Network configuration, to map names on the correct IP address.
Described in deliverable:	D.A4a
Comments:	From an infrastructure perspective, creation of VM could be the registration/deployment of the images to infrastructure. This is probably an area that A3 needs to look at. The start process is automated. Network management needs to be done manually.

6.1.3 Support in offering, selecting and negotiating SLAs.

Table 11: Support in offering, selecting and negotiating SLAs

Is there an interface for customers to browse and select offered services and the associated SLA offers?	
Related Metric: M1.3.1	
(answered by)	
Answer	Yes
Interface Name:	QueryCatalogue
Module Name offering this interface:	eContracting
Described in deliverable:	DA2.a
Short Description:	
To what extent is the selection and negotiation of SLAs automated from the customer's perspective?	
Related Metric: M1.3.3	
(answered by A5)	
Percentage:	90%
Automated Steps:	Translation of the offer from business to software level and then to infrastructure level, as the hierarchy is constructed.
Manual Steps:	1. Selection of template from within the adhoc demonstrator 2. Filling in the template with desired values and submitting it
Described in deliverable:	D.A5a
Comments:	
To what extent is the negotiation of SLAs automated from the service provider's perspective?	

Related Metric: M1.3.4	
(answered by A5)	
Percentage:	100%
Automated Steps:	Fully automated – no human intervention required.
Manual Steps:	
Described in deliverable:	D.A5a
Comments:	Templates have to be prepared in advance; also, the implementation lacks a number of features to be added in next versions of the framework (e.g. template discovery).
To what extent is the adaptation of SLA offers / SLA templates automated from the service provider's perspective?	
Related Metric: M1.3.5	
(answered by A5)	
Percentage:	0%
Automated Steps:	
Manual Steps:	Templates have to be updated in a fully manual manner. This is not expected to change for the whole project duration.
Described in deliverable:	D.A5a
Comments:	SLAs cannot be changed (re-negotiated) after being established. This will be available in consequent releases of the framework.

6.1.4 Support in operating the ORC offered by the framework

Table 12: Support in operating the ORC

Are warning events on the infrastructure level used to enhance the SLA monitoring on the software level?	
Related Metric: M1.4.3	
(answered by A4)	
Answer	Yes
Module sending this event:	ResourceLease (A4). This will change as a new Agent architecture is integrated
Modules receiving the event:	Adjustment (A3)
Described in deliverable:	Not in A4
Short Description:	Adjustment will wait for confirmation of SLA violations from monitoring, SLA warnings might not contain enough information to determine the adjustment measures
Are there events for SLA violations on the software level (services of the ORC)?	
Related Metric: M1.4.4	
(answered byA5)	

Answer	Yes
Module sending this event:	Monitoring
Modules receiving the event:	Adjustment
Described in deliverable:	D.A5a
Short Description:	As soon as Monitoring detects a violation, an event is emitted through the event bus and the Adjustment module receives it. Following, Adjustment is evaluating the event and conditions and takes a respective course of action.
Does the SLA framework provide an overview on the currently active SLAs?	
Related Metric: M1.4.8	
(answered by A5)	
Answer	Yes
Interface:	getHierarchy
Module providing this overview:	Provisioning
Described in deliverable:	D.A5a
Short Description:	The getHierarchy() method receives a SLA ID and based on that it returns its parent SLA and its children SLAs (i.e. the ones of lower levels on which it relies). Consecutive invocations can traverse the complete hierarchy for a SLA.

6.1.5 Support for the infrastructure provider

Table 13: Support for the infrastructure provider

Is there an interface that allows to change the resources associated to the virtual machine running the ORC?	
Related Metric: M1.5.1	
(answered by A4)	
Answer	Yes
Interface Name:	`org.slasoi.infrastructure.management.InfrastructureImpl.reprovision(this is the actual implementation)`
Module Name offering this interface:	Infrastucture.management
Described in deliverable:	D.A4a
Short Description:	Reprovisioning can - start new VMs under the same infrastructure ID, - stop running VMs, - change the CPU parameters (speed currently)
To what extent is the adjustment of resources for virtual machines automated?	
Related Metric: M1.5.2	
(answered by A4)	
Percentage:	10%

Automated Steps:	Detection of SLA Violations, calling the reprovisioning interface
Manual Steps:	
Described in deliverable:	D.A4a, re-provisioning/ A3? There is also an ongoing experimentation not described in any of the public deliverables.
Comments:	Manual experimentation must come first to be able come out with a model in which different adjustment parameters are tested and see the behaviour of the adjustment. Also, adjustment of resources requires that the SLAs support this adjustment (for example, infrastructure should not give more resources than the ones specified by the SLAs, even if it means that a violation (customer arrival rate) is violated as the infrastructure is overloaded. Automation of this could mean also the automation of SLAs
Is there an interface that allows the infrastructure provider to monitor resource utilization of virtual machines running the ORC?	
Related Metric: M1.5.3	
(answered by A4)	
Answer	Yes
Interface Name:	This will be integrated within Monitoring/new Agent architecture. Currently there is an approach developed for year 1
Module Name offering this interface:	A4 Resource, A4 Monitoring
Described in deliverable:	D.A4a
Short Description:	Infrastructure monitoring employs a hierarchy of distributed monitoring agents. Agents are divided into three main categories: • low-level data collector modules • SLA-validation modules • high-level agents providing services like data storing, decision making etc.

6.2 Extensibility of the Demonstrator

This part of the evaluation focuses on the extensibility of the SLA framework. As already presented in the evaluation plan, we defined several change requests within the ORC scenario. Work package leads were asked to estimate the effort required to fulfil these and other new requirements. The following tables give an overview about these estimations.

Table 14: Extensibility of the ORC

Several independent customers need to be served

simultaneously		
A4	To be able to deploy the ORC for new customers requires software management to be able to manage multiple copies of the same VM (or VMs) and infrastructure to manage network configuration (virtual private networks, public connections). Currently A4 does not perform network management in year 1 and this needs to be addressed with a considerable effort (SOTA/Best Practices, implementation, integration),	
A5	List of affected classes	None
	Description of necessary implementation steps	-
	Required changes in other modules	-
	Sequence (path) of changes	-
	Changes of interfaces	0
	Affected Lines of Code	0
	Estimated effort in person days	0
	Number of methods that need be adjusted	0
A6	List of affected classes	5-10
	Description of necessary implementation steps	Combine the usage profiles of different customers using model transformations
	Required changes in other modules	Other components in Negotiation have to call the Prediction Service differently
	Sequence (path) of changes	
	Changes of interfaces	Interface of the Prediction Service has to be adjusted
	Affected Lines of Code	~2000
	Estimated effort in person days	20 – 30
	Number of methods that need be adjusted	50 – 80
A new business level SLA (e.g., diamond) should be added		
A5	List of affected classes	None
	Description of necessary implementation steps	-
	Required changes in other modules	-
	Sequence (path) of changes	-
	Changes of interfaces	0
	Affected Lines of Code	0
	Estimated effort in person days	0
	Number of methods that need be adjusted	0
The interpretation of a specific business level SLA needs to be changed		
A5	List of affected classes	None
	Description of necessary implementation steps	-
	Required changes in other modules	-
	Sequence (path) of changes	-

	Changes of interfaces	0
	Affected Lines of Code	0
	Estimated effort in person days	0
	Number of methods that need be adjusted	

New SLA parameters should be supported (e.g. reliability/ availability).		
A5	List of affected classes	Negotiator, SLAtranslation
	Description of necessary implementation steps	• Core model update • SLA templates update • Addition of these parameters into Negotiator for reasons of feeding them into Prediction Addition of these parameters into SLAtranslation for creating the new SLA documents based on Prediction's results
	Required changes in other modules	Monitoring
	Sequence (path) of changes	
	Changes of interfaces	None
	Affected Lines of Code	
	Estimated effort in person days	
	Number of methods that need be adjusted	None
A6	List of affected classes	200
	Description of necessary implementation steps	- Identify relevant factors that have to be modelled - Extend prediction models to support reliability - Identify and implement a suitable prediction method
	Required changes in other modules	Negotiation
	Sequence (path) of changes	
	Changes of interfaces	Interface of the Prediction Service
	Affected Lines of Code	30000
	Estimated effort in person days	400 -500
	Number of methods that need be adjusted	1000 - 1200
	Comments	This large refactoring is necessary, because the current prediction service is specialized on performance prediction. Predicting other extra functional attributes e.g. reliability requires big changes within the prediction method, as the influence factors on reliability differ from the influence factors on performance such as CPU speed and usage. Additionally the prediction models have to be extended to allow modelling the influence factors on performance as well as the influence

		factors on reliability.
A new service is added to the ORC		
A5	List of affected classes	None
	Description of necessary implementation steps	New templates need to be created
	Required changes in other modules	
	Sequence (path) of changes	
	Changes of interfaces	
	Affected Lines of Code	
	Estimated effort in person days	
	Number of methods that need be adjusted	
A6	List of affected classes	None, only ORC models are affected
	Description of necessary implementation steps	Modelling the behaviour and dependencies of the new service
	Required changes in other modules	None
	Sequence (path) of changes	
	Changes of interfaces	None
	Affected Lines of Code	0
	Estimated effort in person days	1-5
	Number of methods that need be adjusted	0
Dependencies between services are changed (e.g., Payment-Service also requires Inventory Service)		
A5	List of affected classes	Negotiator
	Description of necessary implementation steps	• Update of templates • Update of Negotiator's respective hard-coded part (comment: in upcoming releases of the framework, service dependencies will not affect Negotiating entity's logic)
	Required changes in other modules	
	Sequence (path) of changes	
	Changes of interfaces	
	Affected Lines of Code	
	Estimated effort in person days	
	Number of methods that need be adjusted	
A6	List of affected classes	None, only ORC models are affected
	Description of necessary implementation steps	Modelling the new dependencies
	Required changes in other modules	None
	Sequence (path) of changes	
	Changes of interfaces	None
	Affected Lines of Code	0
	Estimated effort in person days	1
	Number of methods that need be	0

	adjusted	
New deployment options are added (e.g., separated Comosite Service)		
A5	List of affected classes	None
	Description of necessary implementation steps	Infrastructure SLA templates would need to be updated
	Required changes in other modules	
	Sequence (path) of changes	
	Changes of interfaces	
	Affected Lines of Code	
	Estimated effort in person days	
	Number of methods that need be adjusted	
A6	List of affected classes	20 - 30
	Description of necessary implementation steps	- Extend deployment model - Prediction Service must build allocation and infrastructure model dynamically
	Required changes in other modules	None
	Sequence (path) of changes	
	Changes of interfaces	None
	Affected Lines of Code	5000
	Estimated effort in person days	60 – 80
	Number of methods that need be adjusted	250 - 400

7 Conclusions

7.1 Summary

The Open Reference Case work package has successfully provided the first demonstrator of SLA@SOI framework. The major steps were to adapt the CoCoME application, to develop usage simulation environment and to design and develop adhoc demonstrator graphical user interface, which is a tool for demonstrating the functioning of SLA@SOI framework. Also, a brief evaluation of the first framework version was done.

7.1.1 CoCoME Adaptation

Common Component Modelling Example (CoCoME) was extended to a service-oriented application which can be deployed and managed by SLA@SOI framework. The added Web Service layer comprises a few atomic services and a composed service implemented as BPEL process. The ORC is available in different deployment options that allow the framework to choose from them on base of the prediction service's results. The instrumentation of services was included into application, which enables the SLA@SOI framework to monitor the services, to evaluate the monitoring results and to make the required adjustment steps to avoid violations of SLAs.

7.1.2 Usage Simulation Environment

An environment for invoking ORC services was developed which enables simulating customer's activity. It supports various configuration options to use the ORC with different customer's profiles. This enables testing the SLA@SOI framework and its reaction in different scenarios (e.g. overload). Configuration options enable specifying the number of cash desks, the average number of goods per cash desk consumer, delays between services invocations and different probabilities which determine the workflow behaviour, e.g., card validation probability. Once the simulation process is finished, it is simple to repeat it with exactly the same configuration parameters, if additional monitoring or debugging is needed.

7.1.3 Adhoc Demonstrator Graphical User Interface

The adhoc demonstrator graphical user interface was designed and developed. It was integrated with the usage simulation environment and with the SLA@SOI framework. It serves as a tool for monitoring and controlling different aspects of SLA@SOI framework, open reference case and simulation environment. It includes the starting mechanism for the SLA@SOI framework and the simulation environment. Additionally, it provides an overview about what is happening in the framework, in the simulation environment, and inside the ORC.

7.1.4 Evaluation

Getting the SLA@SOI framework running in the ORC scenario was the first part of the evaluation and demonstrates the feasibility of the SLA@SOI framework.

Additionally, we developed an evaluation plan that is used to evaluate the ORC and the reference demonstrator developed in the next project years. This evaluation plan is based on the Goal/Question/Metric (GQM) approach. The evaluation of the SLA@SOI framework in the ORC scenario is focused on implementation details of framework components rather than on usability and performance, because of the early phase of the project. Work package and module leads were asked to answer on different questions about automation and extensibility of the framework.

7.2 Outlook on Future Work

The adhoc demonstrator will serve as a basis for development of reference demonstrator which has to be presented at the end of the project. Components that were developed inside B2 work package will be upgraded and modified in parallel with the development of framework. In year two, the ORC scenario is extended, which also requires extensions of the ORC's implementation. In the last project year, the final referance demonstrator is implemented on base of the results of the previous project years.

7.2.1 ORC Development

Additional services, especially heavyweight services with different resource consumption patterns will be added to the open reference case. It will be possible to select and compose different services into various products.

Additional deployment options and levels of separation will be created to allow framework to choose between a richer set of possible options.

7.2.2 Usage Simulation Environment Development

The Usage simulation environment will be adapted according to the modifications and improvements of open reference case.

The environment will be modified to offer a precise translator between SLA parameters and simulation configuration options.

7.2.3 Adhoc Demonstrator Graphical User Interface Development

All panels of graphical user interface will be modified to reflect improvements and modifications of SLA@SOI framework and simulation environment. Also, a new panel for service management will be included to allow customer to browse through available products, services and operations, to select them and to compose the desired product.

7.2.4 Evaluation Plan

The final evaluation of SLA@SOI framework will be more customer-oriented. The focus will move from the implementation details to the performance and usability. Framework functionalities, extensibility and level of automation will be also compared and evaluated with the consideration of industrial use case requirements.

References

[1] COCOME, available at http://www.cocome.org
[2] SLA@SOI Deliverable D.A1a Framework Architecture
[3] JDBC, available at http://java.sun.com/javase/technologies/database/
[4] Hibernate, available at https://www.hibernate.org/
[5] Transfer Object, available at http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html
[6] WSBPEL, available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[7] ActiveBPEL, available at http://www.activevos.com/community-open-source.php
[8] SLA@SOI Deliverable D.A4a SLA-aware Infrastructure Management
[9] JAX-WS, available at http://en.wikipedia.org/wiki/JAX-WS
[10] AXIS, available at http://ws.apache.org/axis/
[11] JSON, available at http://www.json.org/
[12] XConsole, available at http://xtreemos.org/publications/plonearticlemultipage.2008-06-26.0232965573/project-deliverables/d3-3-3-4final.pdf
[13] Eclipse Rich Client Platform, available at http://wiki.eclipse.org/index.php/Rich_Client_Platform
[14] SLA@SOI Deliverable D.A2a Business SLA Management
[15] SLA@SOI Deliverable D.A3a SLA-Aware Service Management
[16] SLA@SOI Deliverable D.A5a SLA Foundations and Management
[17] SLA@SOI Deliverable D.A6a Predictable / Manageable Service Engineering Methodology and Prediction Services
[18] Log4j, available at http://logging.apache.org/log4j/1.2/index.html
[19] V.R. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. Encyclopedia of Software Engineering, 1:528-532, 1994

Appendix A : Glossary

For a complete glossary please refer to D.A1a official deliverable [2].

Appendix B: Abbreviations

CoCoME	Common Component Modelling Example
GQM	Goal Question Metric
GUI	Graphical User Interface
JSON	JavaScript Object Notation
ORC	Open Reference Case
POJO	Plain Old Java Object
VA	Virtual Appliance
VM	Virtual Machine
SaaS	Software as a Service
SLA	Service Level Agreement
SOA	Service Oriented Architecture
TO	Transfer Objects
WS	Web Service
WSDL	Web Services Description Language
XML	Extensible Markup Language

Appendix C : ORC Deployment Guide

C.1 Short Description

This documentation serves to describe the deployment details including necessary system configurations of the Open Reference Case (ORC) Demo on CentOS, which is a free community Enterprise-class Linux Distribution.

C.2 Prerequisite

- Java Development Kit (JDK) 1. 5 or higher
- Apache Ant 1.6 or higher
- No white spaces in the installation path of the system
- Deployable ORC Deployment-Package

Note that, so far the ActiveBPEL Engine Community Edition only supports JDK 1.5. Therefore, the Tomcat 5.5 is chosen the servlet container.

C.3 Architecture Overview

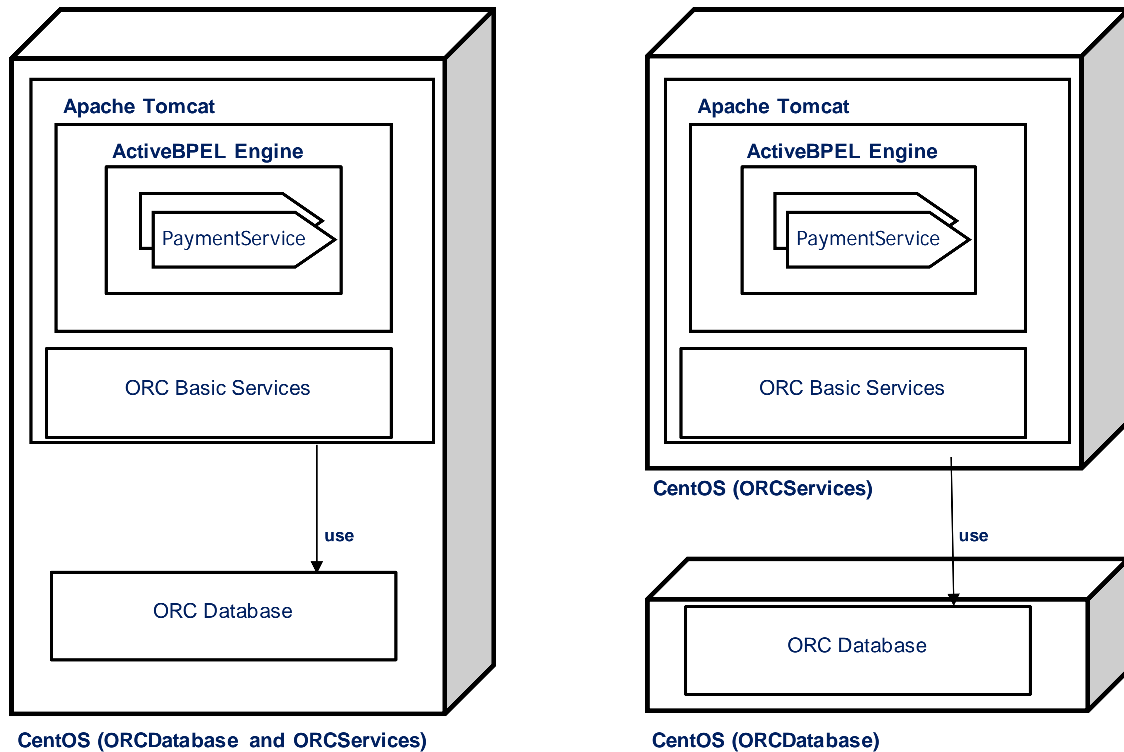

Figure 47: Deployment Options of ORC Demo on CentOS.

As depicted in Fig. 1, the whole ORC demo can be deployed on up to 2 computers running CentOS. In this deployment description they are called *ORCServices* and *ORCDatabase*. The ORC services are deployed on *ORCServices* the underlying database runs on *ORCDatabase*. It is also possible to combine them on one single machine, which means, that one image contains the services as well as the database.

C.4 Deployment Procedure

1. To be done on all images:
   If Java and Ant 1.6 are not installed on the systems, they must be installed first:

   a. Download JDK 1.5 and install it on *ORCServices*, and *ORCDatabase*. For more information of installation of Java on Linux, please refer to http://www.sun.com/java. When the installation is complete, set the environment variable JAVA_HOME in the ./etc/profile file. An example is as follows:

   ```
   export JAVA_HOME=/usr/lib/jvm/java
   ```

2. To be done on the system running the ORC Database

   a. Download the current version of Apache Ant and install it on *ORCDatabase*. For more information of installation of Java on Linux, please refer to http://ant.apache.org/manual/index.html

      With ubuntu this can be done with the following commands:
      ```
      sudo apt-get install sun-java5-jdk
      sudo apt-get install ant
      ```

   b. Download the ORC Deployment-Package and install (extraction needed for .zip format) it. An example of the installation path is as follows:

      ```
      ./usr/ORC_DEMO
      ```

   c. Open the command terminal and change the path to `rsc` of the installation path of ORC_Database package in step 2a. Run the ant target `startderbydatabase` in the `buildDB.xml`. An example is as follows:

      ```
      ant -f buildDB.xml startderbydatabase
      ```

   d. Open the command terminal and change the path to `rsc` of the installation. At the first start of the ORC demo run the ant target fillDB. An example is as follows:

      ```
      ant -f buildWs.xml fillDB (Only at initialization)
      ```

3. To be done on the system running the ORC services

   a. Download the ORC Deployment-Package and install (extraction needed for .zip format). An example of the installation path is as follows:

      ```
      ./usr/ORC_DEMO/
      ```

   b. The atomic web services are configured to run on *ORCBasicServices*, therefore it is necessary to edit the `hosts` file of the system. This file is located in `/etc`. It is necessary to map the name `ORCBasicServices` to the local IP-address (not 127.0.0.1) of this machine. In our example:

```
141.21.4.81   ORCBasicServices
```

To make sure if `ORCBasicServices` has been mapped into 141.21.4.81, one could use the command `ping ORCBasicServices` for testing.

c. To access the database it is also necessary to edit the hosts file of the system. The IP-adress of the database server has to be mapped to the name `ORCDatabase`. In our example:

```
141.21.4.108  ORCDatabase
```

d. When the installation is complete, set the environment variable CATALINA_HOME in the ./etc/profile file. An example is as follows::

```
export CATALINA_HOME=./usr/ORC_DEMO/tomcat
```

e. Open the command terminal and change the path to `bin` of the installation path of ORC Demo – PaymentService. Use the command `chmod` to assign the "execution" right to all the .sh files:

```
chmod a+x *.sh
```

f. Start the tomcat by executing the startup.sh under the path `bin` of the installation path of ORC services.

C.5 Automated Starting of the ORC

This section describes how to start the Database, the BasicServices and the CompositeService automatically when the underlying virtual machine is started. If this automated starting is used, it is necessary to boot the virtual machines in the correct order. This means first the database, second the ORC Services. A wrong order might lead to problems and unavailable services.

1. The startup scripts `ORC_Database`, `ORC_Services`, located in the directories the zip archives were extracted to, must be copied to /etc/init.d of the respective virtual image. It might be necessary to adapt the scripts if different directories are used. Within ORC_CompositeService the JAVA_HOME variable is set to `/usr/lib/jvm/java`, and the AXIS_HOME variable is set to/usr/ORC_DEMO/ORCServices/tomcat/webapps/axis. It might be necessary to adapt this.
2. The copied scripts have to be made executable (using chmod 755)
3. Change to the directory `/etc/init.d` and run
    a. `chkconfig –add ORC_Database`
    b. `chkconfig –add ORC_Services`

   on the respective machine.

   On a Ubuntu system:
   Install sysv-rc-conf via apt-get (**`sudo apt-get install sysv-rc-conf`**) and then run sysv-rc-conf standalone and select the run levels 2,3,4,5 for the ORC_* scripts

```
sudo sysv-rc-conf <<SCRIPTNAME>>
```

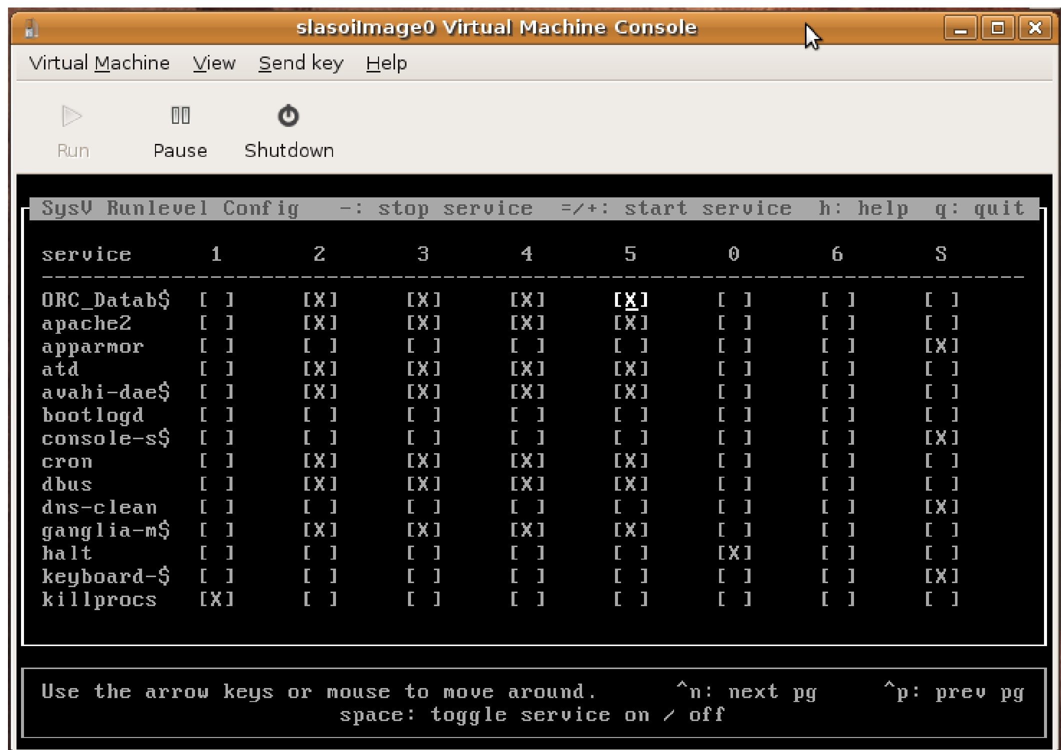

Figure 48: Installed Services

4. The parts of the ORC are now registered as services and are started automatically at boot time. The services can also be started by hand to test them using the command `service <<ServiceName>> start`. With <<ServiceName>> = ORC_Database or ORC_Services

C.6 Testing the ORC

In order to test the correct deployment of the ORC services 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 visit the URL `http://<<IP_of_ORCServices>>:8080/ BpelAdmin/`. Click on `Deployed Services`, the PaymentService should be listed there (see Figure 49: PaymentService deployed in ActiveBPEL.).

Deployed Services

Name	Process Name
AeB4PEscalation	aeb4-task-escalation
AeB4PLifecycle-stateChangeListener	aeb4p-task-lifecycle
AeB4PNotificationLifecycle	aeb4p-notification-lifecycle
AeB4PRemoveOwner	aeb4p-task-remove-owner
AeB4PState-aeInternalTaskOperations	aeb4p-task-state
AeB4PState-aeTaskOperations	aeb4p-task-state
AeB4PState-aeTaskOperations-getAttachmentById	aeb4p-task-state
AeB4PState-aeTaskOperations-getInstance	aeb4p-task-state
AeB4PState-stateChangeRegistry	aeb4p-task-state
AeB4PState-taskOperations	aeb4p-task-state
AeB4PState-taskOperations-getAttachmentInfos	aeb4p-task-state
AeB4PState-taskOperations-getAttachments	aeb4p-task-state
AeB4PState-taskOperations-getFault	aeb4p-task-state
AeB4PState-taskOperations-getInput	aeb4p-task-state
AeB4PState-taskOperations-getOutput	aeb4p-task-state
AeB4PState-taskOperations-getRenderingService	aeb4p-task-state
AeB4PState-taskOperations-getRenderingTypesService	aeb4p-task-state
AeB4PState-taskOperations-getTaskDescription	aeb4p-task-state
AeB4PState-taskOperations-getTaskInfo	aeb4p-task-state
AeB4PSuspendUntilResumer	aeb4p-task-suspenduntilresumer
AeB4PTaskClient-aeTaskOperations	aeb4p-task-client
AeB4PTaskClient-taskOperations	aeb4p-task-client
AeB4PTaskLifecycle	aeb4p-task-lifecycle
AeIdentityService	aeidentitysvc
ApproveLoan	loanApprovalProcess
PaymentService	paymentService
eventHandlerPLTService	event_monitor
eventMonitorDisableService	event_monitor
eventMonitorEnableService	event_monitor
eventNotificationPLTService	event_monitor

Copyright © 2004-2008 Active Endpoints, Inc.

Figure 49: PaymentService deployed in ActiveBPEL.

The functionality of the services can be tested with a SOAP client like SOAPUI[2]. In the following the tests are explained for the PaymentService and InventoryService.

C.6.1 PaymentService

In order to test the composite service PaymentService, you have to create a new SOAPUI project and entering following WSDL location:

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

As shown in Figure 50 you have to enter following parameter values:

```
cardInformation: blablabla
cardnumber: 7777
```

[2] http://www.soapui.org

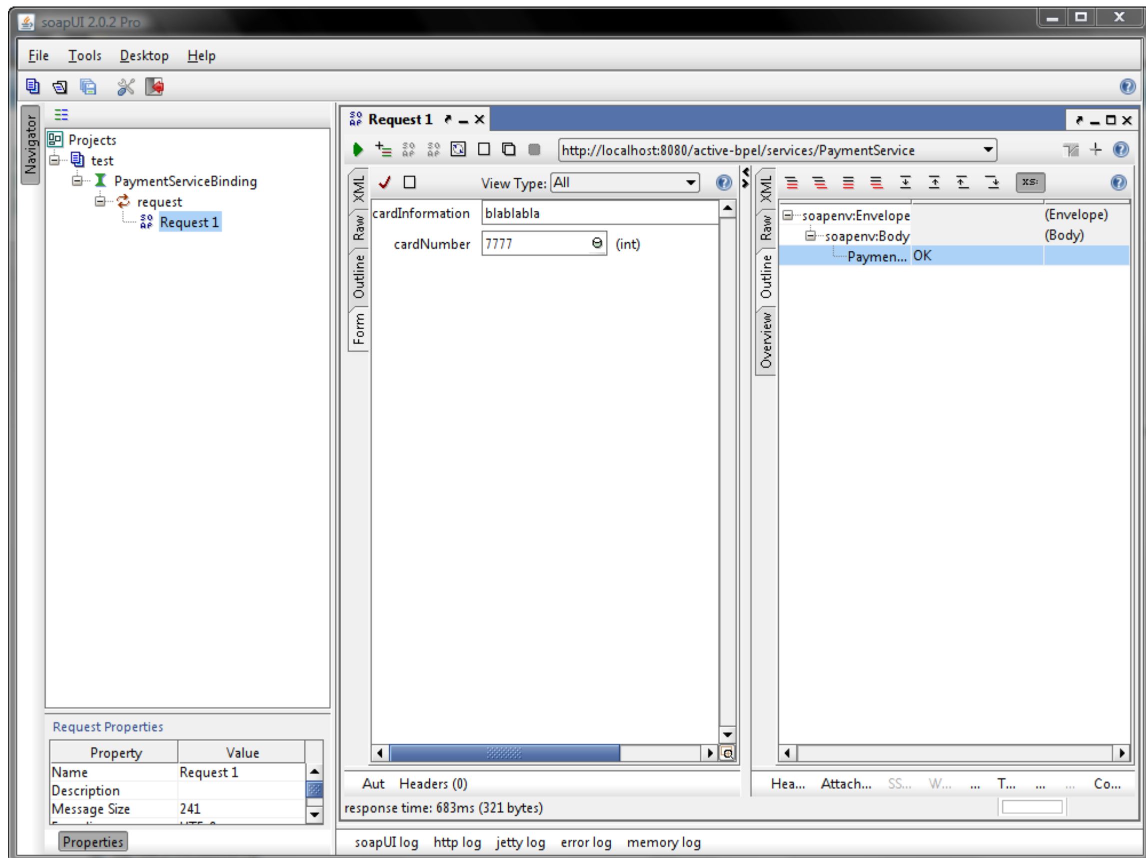

Figure 50: Test PaymentService in SOAPUI.

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

C.6.2 InventoryService

Create a new SOPUI Project with following URL to the wsdl:
Fehler! Hyperlink-Referenz ungültig. `services/InventoryService?wsdl`

Valid parameter values are:

```
In0 = 1
In1 = 1000
```

Appendix D: Simulation Environment Implementation

Implementation of the simulation environment is split into several Java packages.

Table 15: Main classes and packages constituting the adhoc Simulation Environment.

Class	Description
org.slasoi.adhoc.workload. WorkloadGenerator.java	Main program that subscribes to the message bus and delegates the work to the manager.
org.slasoi.adhoc.workload. AdhocJobManager.java	Class that manages all agents and handles commands received from the message bus (more specifically, from the main program that is listening to the message bus).
org.slasoi.adhoc.workload.agent.*	Abstract class for the sales process and concrete implementations based on JAX-WS and Axis.
org.cocome.tradingsystem.webservices.{jaxws,axis}	Classes generated by maven plugins for accessing services
DIXI (eu.xtreemos.*)	The console command parser and code generator is based on the DIXI library [12]

D.1 WorkloadGenerator.java

This is the main program, responsible for:

- Creating unique GUID of the workload generator. All the agents running on this generator use this same GUID to distinct themselves from other workload generators.
- Starting communication via message bus.
- Intercepting message bus messages depending on the message type. Commands are tunnelled into the command line parser, auto-generated by DIXI, configurations are parsed and passed to the generator and its agents.
- Stopping the communication via message bus.

Access to the message bus is configured from property file called *manager.properties*. Default values are shown in Table 16. The messaging engine is usually *XMPP* although it might be different once different message bus implementation are supported. Since we are working with Pub/Sub type of messaging, we need to specify the PubSub engine as well.

Properties xmpp_username and *xmpp_password* need to be changed for every workload program. Alternatively, anonymous users can be used by removing these two properties from the configuration file. It is important to note that the *xmpp_channel* needs to be the same as the one that is used by the adhoc GUI. Its name, however, can be arbitrary.

Properties under "XMPP – server" section depend on the XMPP server and should be adapted as well.

Table 16: Example of manager.properties configuration.

```
#Messaging engine
messaging=xmpp

#PubSub engine
pubsub=xmpp

# XMPP - user
xmpp_username=wg1
xmpp_password=wg1
xmpp_channel=testChannel-adhoc1

# XMPP - server
xmpp_host=192.168.0.23
xmpp_port=5222
xmpp_service=virtuoz
xmpp_resource=workloadgenerator
xmpp_pubsubservice=pubsub.virtuoz
xmpp_chatservice=conference
```

D.2 AdhocJobManager.java

This class is the connector between workload manager main program and agents. It consists of two kinds of methods:

- **Job list management** methods manage agents, create and initialise them, retrieve them, retrieve the number of idle jobs, etc.
- **Methods used to command workload generator, agents and instances** are used by message bus command parser. They are all annotated with Java Annotation

  @XOSDCONSOLE(consoleString = "_console_command_")

  This annotation is used by XtreemOS DIXI service processor, that generates the code needed for emulating a console. It creates a file *eu.xtreemos.xconsole.command.XConAdhocJobManager* with methods needed for command parsing. This way commands that come through message bus, can have console-like style and are therefore more user friendly, eventually a console window could be used for controlling a workload generator.

D.2.1 XOSDCONSOLE Example

Table 17 shows an example of a real function, annotated with the XOSDCONSOLE annotation. DIXI command parser and code generator parses existing source code and generates stub functions for all functions, annotated with the XOSDCONSOLE annotation. The name of the command may be changed with the *_console_command_* parameter, while command parameters are generated from the method signature found in the source file.

At the moment, DIXI works best with textual parameters although support for numeric parameters is also implemented.

Table 17: Example of a function, annotated with the XOSDCONSOLE annotation.

```
@XOSDCONSOLE(consoleString = "startInstance" )
public static void startInstance(String agent, String instance) {
    // implementation
}
```

D.3 Agent.java

This class implements the sales process workflow from **Fehler! Verweisquelle konnte nicht gefunden werden.** of a single agent. It is an abstract class, extended by AgentJaxWs and AgentAxis that provide access to web services using JAX-WS and Axis, respectively.

D.4 Generating Java code from WSDL

Two maven plug-ins were used for generation of Java code needed to access Web Services. This classes are then used by AgentJaxWs and AgentAxis classes in package org.slasoi.adhoc.workload.agent. The use of these plug-ins is briefly described in the following sub-sections.

D.4.1 JAX-WS

For creation of JAX-WS code form certain WSDL the URL of the WSDL must be added to *pom.xml* in the following section. Entries *packageName* and *wsdlUrls* contain the data about the Web Service.

Table 18: JAX-WS pom.xml configuration.

```
<build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <configuration>
          <sourceDestDir>src/main/java</sourceDestDir>

<packageName>org.cocome.tradingsystem.webservices.jaxws</packageName>
          <verbose>true</verbose>
          <wsdlUrls>
            <wsdlUrl>http://virtuoz:8081/InventoryService?wsdl</wsdlUrl>
          </wsdlUrls>
        </configuration>
      <plugin>
    <plugins>
<build>
```

The code is generated using the the following maven command:

```
# mvn jaxws:import
```

Verbose parameter, although not required, can help in finding problems with the plug-in and/or WSDL configuration.

D.4.2 Axis

Axis client classes are generated with a similar Maven plugin.

Table 19: Axis pom.xml configuration.

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.axis</groupId>
        <artifactId>axis-wsdl2code-maven-plugin</artifactId>
        <version>1.4</version>
        <executions>
          <execution>
            <goals>
              <goal>wsdl2code</goal>
            </goals>
            <configuration>
```

```
                    <packageName>org.slasoi.adhoc.webservices.axis</packageName>
                    <wsdlFile>workload/src/main/axis/InventoryService.wsdl</wsdlFile>
                    <databindingName>xmlbeans</databindingName>
                  </configuration>
                </execution>
              </executions>
            </plugin>
       <plugins>
   <build>
```

The code needed is then created by command

```
# mvn wsdl2code:wsdl2code
```

At the time of writing, the plug-in had some difficulties downloading WSDLs from the server. To this end, we needed to download WSDL files manually and use *wsdlFile* property in Maven configuration.

Appendix E : Adhoc Demonstrator Development Requirements

To start using Eclipse Rich Client Platform development environment few components that are not included in the basic Eclipse release are needed:

- Eclipse Plug-in Development Environment
- Eclipse RCP
- Eclipse RCP Plug-in Developer Resources

These components can be found in Eclipse Software Update panel, under Available Software. Alternatively, Eclipse for RCP/Plug-in Developers [1] version that includes all these components can be installed.

Once the development environment is installed and the source code for the adhoc demonstrator is imported into Eclipse framework as an existing project, there are still two steps to enable adhoc demonstrator development:

- "Plug-in dependencies" libraries have to be added to Eclipse project: Project menu/Properties/Java Build Path/Libraries tab/Add Library button.
- Since adhoc demonstrator is an Eclipse plugin, the configuration of plugin.xml file is needed. Plugin.xml, which is provided as part of the source code of adhoc demonstrator, has to be opened with Plug-in Manifest Editor, provided by Eclipse plugin development equipment, and configured to include some additional libraries. They should be included in a Runtime tab of plugin.xml file, in a Classpath section. The required libraries are located in lib folder under the root folder of the project.