Deliverable D2.2: Definition of the Dynamic Development Process for Adaptable QoSaware ULS Choreographies

Marco Autili

ICT IP Project

Deliverable D2.2

Definition of the Dynamic Development Process for Adaptable QoS-aware ULS Choreographies

http://www.choreos.eu

LaTeX template v. 1.16

Project Number	:	FP7-257178
Project Title	:	CHOReOS Large Scale Choreographies for the Future Internet

Deliverable Number	:	D2.2
Title of Deliverable	:	Definition of the Dynamic Development Process for Adaptable QoS-aware ULS Choreographies
Nature of Deliverable	:	Report + Prototype
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	3.0
Contractual Delivery Date	:	30 September 2012
Actual Delivery Date	:	26 October 2012
Contributing WP	:	WP2
Editor(s)	:	Marco Autili (UDA), Davide Di Ruscio (UDA)
Author(s)	:	Marco Autili (UDA), Davide Di Ruscio (UDA), Paola Inverardi (UDA), Massimo Tivoli (UDA), Amleto Di Salle (UDA), Dionysis Athanasopoulos (UOI), Apostolos Zarras (UOI), Panos Vassiliadis (UOI), James Lockerbie (CITY), Neil Maiden (CITY), Antonia Bertolino (CNR-ISTI), Darius Silingas (No Magic Europe), Rokas Bartkevicius (No Magic Europe), Marco A. Gerosa (USP), Gustavo A. Oliva (USP), Leonardo Leite (USP), Paulo Moura (USP), Alfredo Goldman (USP), Yanik Ngoko (USP), Alessio Carenini (CEFRIEL), Gianmarco Panza (CEFRIEL)
Reviewer(s)	:	Valérie Issarny (Inria), Sebastién Keller (THALES), Guglielmo De Angelis (CNR)

Abstract

The CHOReOS development process reported in this document capitalizes on previous work by the CHOReOS team described in the previous deliverable D2.1. The latter described the CHOReOS software development process by defining the main activities that need to be performed and the artifacts manipulated by these activities without referring to specific technologies, tools, standards, models, etc. This deliverable concretely defines the specific process activities (and their data flow in term of manipulated I/O artifacts) we are implementing to develop CHOReOS choreographies by conforming to the CHOReOS development process model presented in D2.1. In particular, we clearly and concisely specify what standards, notations, languages, technologies and tools we are using to implement the process activities that will be then integrated by the CHOReOS Integrated Development and Run-time Environment (IDRE) being developed in WP5. Furthermore, accounting for the review recommendations, we clarify QoS and data aspects, further defining ULS dimensions and their relationships with the development process activities and CHOReOS case-studies.

Keyword List

Dynamic Software Development Process Model, Requirements Specification, Large-scale Service Base Management, Choreography Synthesis, Choreography Enactment, Design-time and Run-time Analysis, Meta-modeling, Choreography-centric Service Oriented Computing and Architecture, Model-driven Engineering, BPMN2, Dependency Oriented Choreography Analysis

Document History

Version	Changes	Author(s)
0.1	Outline Draft	Marco Autili, Davide Di Ruscio, Paola Inverardi, Massimo Tivoli, Amleto Di Salle (UDA)
0.2	Definition of the first-level process diagram.	All
0.3	Preliminary draft of the synthesis method.	Marco Autili, Davide Di Ruscio, Paola Inverardi, Massimo Tivoli (UDA)
0.4	First level “Design Choreography Specification” diagram modified.	Marco Autili, Davide Di Ruscio, Amleto Di Salle (UDA); James Lockerbie, Neil Maiden (CITY)
1.0	Final Outline.	Marco Autili, Davide Di Ruscio, Massimo Tivoli, Amleto Di Salle (UDA)
1.1	First draft of abstract and Introduction.	Marco Autili, Massimo Tivoli, Davide Di Ruscio, Paola Inverardi (UDA)
1.2	Refined draft of Introduction, Development Process Model, and Choreography Synthesis to allow other partners to provide already integrated contributions.	Davide Di Ruscio, Massimo Tivoli (UDA)
1.3	Added contents into Chapter 2 for the requirements specification and for assessing choreography quality.	- James Lockerbie, Neil Maiden (CITY); Alessio Carenini, Gianmarco Panza (CEFRIEL)
1.4	Refined contributions into Chapter 2 for what concerns requirements specification and synthesis.	Marco Autili, Davide Di Ruscio, Massimo Tivoli (UDA); James Lockerbie, Neil Maiden (CITY)
1.5	Added contributions to Chapter 3 detailing the process activities introduced in Chapter 2	Marco Autili, Davide Di Ruscio, Massimo Tivoli, Amleto Di Salle (UDA); Dionysis Athanasopoulos, Apostolos Zarras, Panos Vassiliadis (UOI); Antonia Bertolino, Guglielmo De Angelis (CNR-ISTI); Marco A. Gerosa, Gustavo A. Oliva, Leonardo Leite, Alfredo Goldman, Yanik Ngoko (USP)

1.6	Checking consistency after adding contributions to Chapter 3.	All
2.0	Proof reading and checking integration after final contributions by all have been refined.	Marco Autili, Davide Di Ruscio (UDA)
2.1	The whole deliverable has been revisited according to the comments of the internal reviewers.	All
3.0	Final editing for final version.	Marco Autili, Davide Di Ruscio, Paola Inverardi (UDA)
3.1	Overall revision addressing the comments of the reviewers on ULS, SoaML and Q4BPMN, QoS and data mapping.	All
3.2	Overall revision addressing the comments of the Scientific Leader. Final editing for final version.	All

Document Reviews

Review	Date	Ver.	Reviewers	Comments
Outline	01 July 2012	1.0	All authors	Outline agreed by all
Draft	28 September 2012	2.0	Marco Autili, Paola Inverardi, Davide Di Ruscio, Massimo Tivoli, Amleto Di Salle	Complete version released for review
QA	15 October 2012	3.0	Marco Autili, Paola Inverardi, Guglielmo De Angelis, Valérie Issarny, Sebastién Keller	Internal review comments
PTC	26 October 2012	A	PTC	Internal review comments addressed and final version ready

Glossary, acronyms & abbreviations

Item	Description
BPMN2	Business Process Modeling Notation, Version 2 - OMG
CA	Consortium Agreement
CD	Coordination Delegate
CTT	Concurrent Task Trees
DL	Deliverable Leader
DOW	Description of Work
EE	Enactment Engine
FI	Future Internet
HSG	Hierarchical Services Graph
IAC	Industrial Advisory Committee
M2C	Model-to-Code
M2M	Model-to-Model
MDD	Model Driven Development
MDE	Model Driven Engineering
MPHSG	Multiple-Process HSG
MST	Management Support Team
OSS	Open Source Software
PL	Project Leader
PMT	Project Management Committee
PO	Project Officer
PTC	Project Technical Committee
Q4BPMN	Quality for BPMN
QoS	Quality of Service
SL	Scientific Leader
SNA	Social Network Analysis
SoaML	Service oriented architecture Modeling Language
SPHSG	Single-Process HSG
ULS	Ultra Large Scale
WP	Work Package
WPL	Work Package Leader
WSC	Web Service Composition
WSBQL	Web Service Base Query Language

Table Of Contents

List Of Tables

List Of Figures

1 Introduction

One of the main objectives of CHOReOS is to define a development process for the development of ULS choreographies in FI. Towards the definition of this development process, WP2 identifies the following macro objectives[1]. Due to the different amount of resources required for their development, as pointed out in the following items, some of them are developed in other WPs:

1) A domain-expert requirements specification of adaptable QoS-aware highly-scalable choreographies (developed in WP2 and WP4);
2) Large scale abstraction-oriented service base management (developed in WP2);
3) Automated choreography synthesis (developed in WP2);
4) Choreography deployment and execution (developed in WP3);
5) Design and run-time analysis (developed in WP2 and WP4);
6) Governance V&V, monitoring and V&V configuration (developed in WP4).

The work herein presented capitalizes on previous work by the CHOReOS team described in the deliverable D2.1 [ARS12a]. The latter presents a model of the CHOReOS development process by abstractly describing the "strategy" that CHOReOS uses for specifying, analysing, enacting, governing, and monitoring ULS choreographies during the whole life cycle (from design to runtime to evolution). The dynamic development process model consists of activities that are common to other development processes, but within CHOReOS they are organized in order to fulfill the specific commitments that have been imposed to deal with ULS service choreographies.

In the deliverable D2.1 we gave an abstract description that characterizes the CHOReOS software development process by defining the main activities that need to be performed and the artifacts manipulated by these activities without referring to specific technologies, tools, standards, models, etc. In turn, this deliverable describes the specific set of concrete process activities[2] (their flow and manipulated I/O artifacts) we are implementing to develop CHOReOS choreographies by conforming to the CHOReOS development process model presented in D2.1. The M24 Demo provides companion videos (released on the CHOReOS web site) that show how all the process activities are performed according to the flow dictated by the development process herein defined.

This deliverable clearly and concisely specifies what standards, notations, languages, technologies, and tools we are using to implement the process activities that are going to be integrated by the CHOReOS Integrated Development and Run-time Enviroment (IDRE) being developed in WP5. Furthermore, accounting for the second review recommendations, we clarify QoS and data aspects, further defining ULS dimensions and their relationships with the development process activities and CHOReOS case-studies. To this end, after introducing the high-level view of the CHOReOS development process

[1] A comprehensive discussion of the State-Of-The-Art related to all the above listed objectives has been carried on in Deliverable D1.1 [Tea10] and, hence, the interested reader can refer to it.

[2] As already done in the deliverable D2.1, BPMN2 Process Diagrams are used as graphical notation.

in Section 2 and before detailing its development activities in Section 4, Section 3 reports the ULS dimensions we have defined and their relationships with both the development activities and the WP6-8 case-studies. Section 5 concludes the document and summarizes future work on WP2.

2 The CHOReOS dynamic development process

The CHOReOS solution is based on an innovative model-driven engineering process supporting a complete lifecycle for service choreographies, which is depicted in Figure 2.1. The whole process is split into two major phases – designing choreography specification and running choreographies.

2.1. Design and Run Choreographies Process

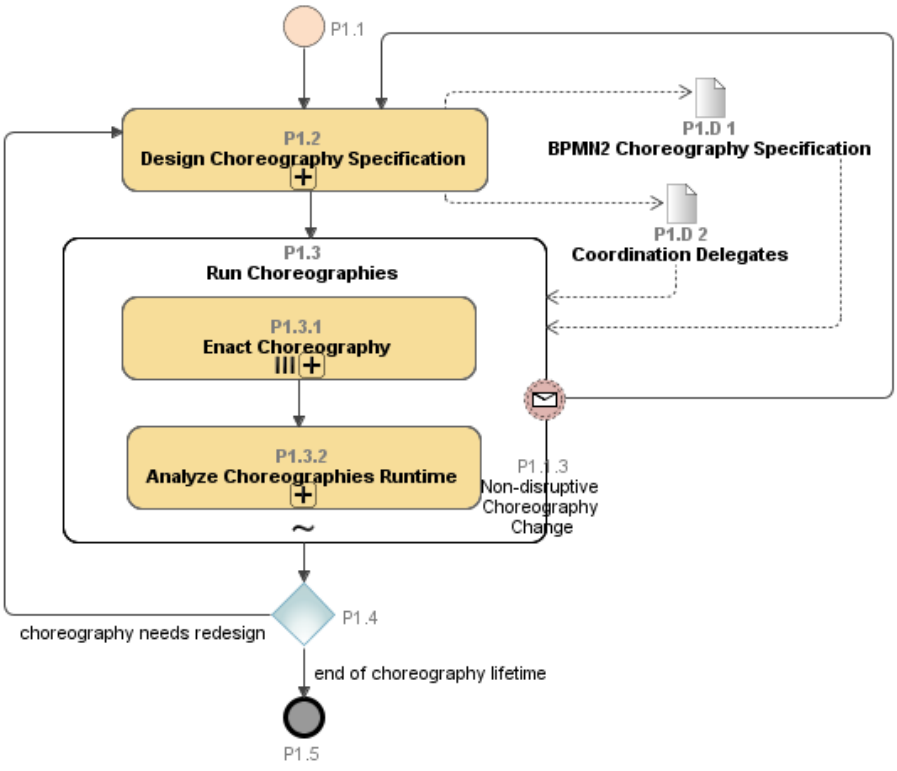

Figure 2.1: Design and Run Choreographies Interactively

The first phase of the CHOReOS development process (see the subprocess *Design Choreography Specification*) takes a choreography designer through a sequence of tasks in order to produce a *choreography specification* and other artifacts that can be used by the CHOReOS service oriented middleware for enacting choreographies. One of the aims of the choreography specification phase is to enable a domain expert-centric approach to specifying QoS aware choreographies, as well as taking an adaptive approach based on the needs of end-users. Once the choreography model is specified taking into account both functional and quality requirements, the business services acting as participants for the modeled choreography have to be discovered based on service descriptions.

Once the choreography specification has been assessed as well formed, the final version of the choreography specification is used for deriving code-based artifacts that are used by the runtime envi-

ronment. More precisely, the final choreography specification is used to synthesize coordination delegates that enable integration of heterogeneous business services in the specified choreography.

As detailed later in the document, in the case of non-disruptive changes (e.g., some services are no longer available, or some QoS requirements are not fulfilled anymore) the choreography needs redesign and thus the *Design Choreography Specification* has to be reiterated. The *Run Choreographies* process accounts for the enactment of choreographies and the subsequent analysis of the choreography runtime. This subprocess only stops when there is a need for choreography redesign or when the choreography reaches the end of its lifetime.

More details about the *Design Choreography Specification* subprocess are given in Section 2.2. Section 2.3 and Section 2.4 detail the two subprocesses of the *Run Choreographies* activity.

2.2. Design Choreography Specification Process

CHOReOS aims at enabling a domain expert-centric approach to specifying QoS aware choreographies. In this respect, an important challenge was to develop a process for situating and specifying the requirements, and required qualities, in context of the choreography to be modeled (see the *Specify Goals and Requirements* task in Figure 2.2). The domain expert is provided with tool support to identify a choreography need and to express functional requirements and end-user qualities on its specification.

In order to define the basic parameters of the search space for services more precisely, we match and combine the requirements on the choreography with user task models. These task models, developed for the domains featured within the CHOReOS use cases, include knowledge of codified workflows (tasks) and natural language terms describing service classes. The matching process returns relevant concurrent task tree (CTT) models and associated requirements which form the basis of the initial, but partial, choreography specification. A set of rules is applied for the transformation into the BPMN2 choreography model (see the *Generate Choreography Specification from CTT Models* task in Figure 2.2). The initial choreography model and requirements are imported into a modeling tool supporting BPMN2 and requirements along with additional properties such as temporal constraints and service class queries.

While the BPMN2 language focuses on functional aspects, it is not designed to handle non-functional requirements or QoS, a term denoting collectively all those properties of the system which are not related to its operational aspects. In order to enable the choreography designer to specify non-functional requirements, CHOReOS proposed Q4BPMN as a way for decorating BPMN2 models with non-functional constraints for choreography participants and tasks (see Section 2.4). A prototype implementation of this approach has been successfully implemented in MagicDraw (see the *Refine Choreography Specification* task in Figure 2.2). In addition, the BPMN2 specification is further refined by producing a specification of service contracts and participants using SoaML.

Once the choreography model is decorated taking into account both functional and quality requirements, the business services acting as participants for the modeled choreography have to be discovered based on service descriptions (see the *Bind Service Descriptions to Choreography Specification* task in Figure 2.2). The searching criteria concern the functional and/or the non-functional properties of the desired services. The discovery process enabled by CHOReOS middleware is based on the provided searching criteria and the abstractions that satisfy such criteria along with their represented services are returned. However, if some services required by the choreography specification are not available from the service base (and hence cannot be discovered), a test-driven development activity is undertaken by considering the SoaML specification of service contracts and participants (see Section 4.4.3).

The CHOReOS process foresees a task to perform a quality assessment by using automated tools (see the *Assess Quality of Choreography Specification* task in Figure 2.2). In particular, if any quality issue is posed on the choreography specification, the choreography designer will need to perform another iteration of choreography design starting from requirements refinement, e.g., setting lower QoS requirements. When the quality is satisfactory the synthesis task is performed to generate the required

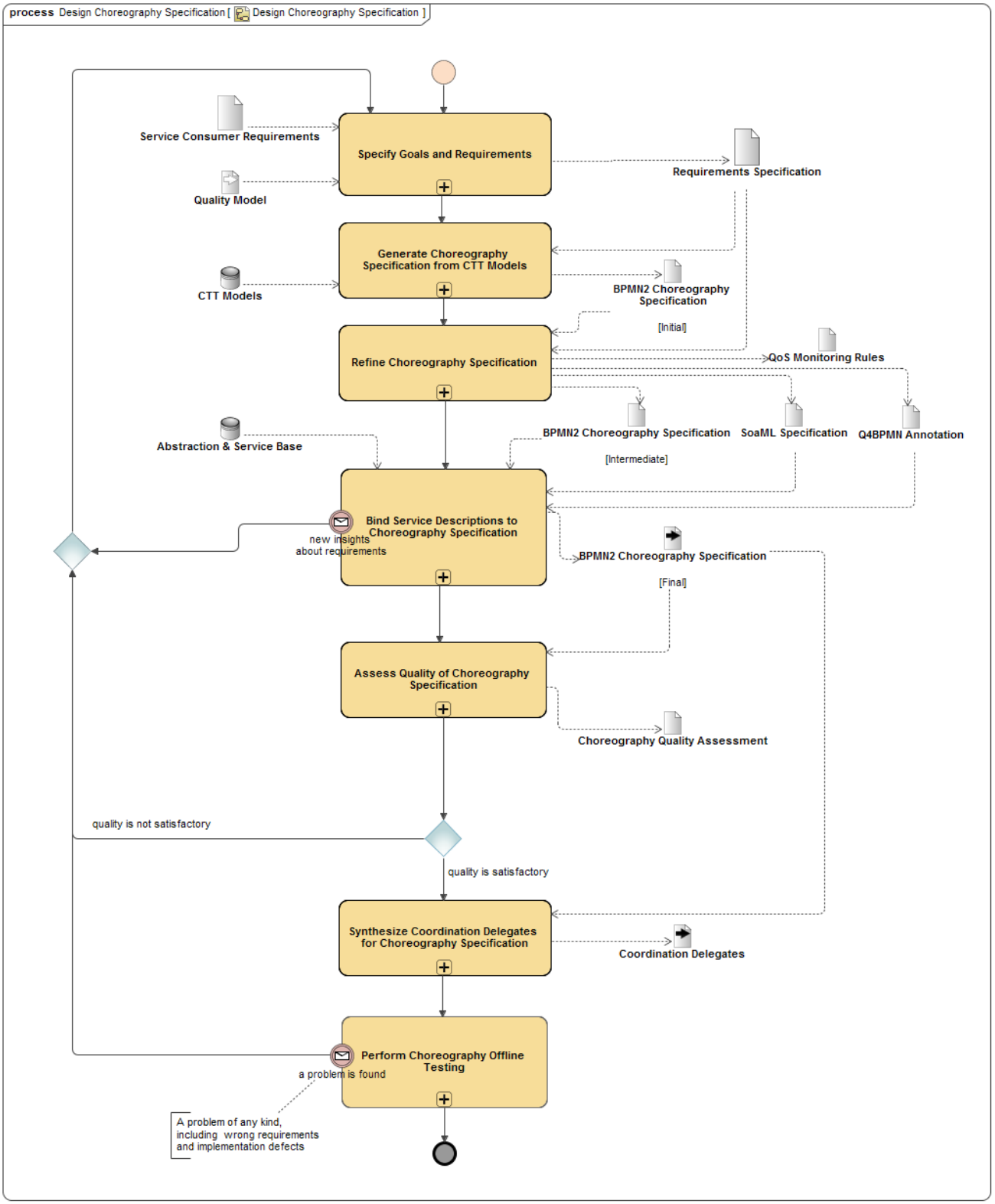

Figure 2.2: Design Choreography Specification

coordination delegates (see the *Synthesize Coordination Delegates for Choreography Specification* task in Figure 2.2), and afterwards offline testing techniques are applied to identify possible wrong requirements and implementation defects (see the *Perform Choreography Offline Testing* task in Figure 2.2).

2.2.1. Specify Goals and Requirements

The approach to designing choreography specifications in CHOReOS begins with the processes for expressing functional and quality requirements on services and choreographies. The development of the requirements specification takes into account service consumer needs and domain expert knowledge. It also applies a quality model developed for CHOReOS in order to extend user expressed non-functional requirements with service qualities, measures, and metrics. The final output is the requirements specification.

More details about the *Specify Goals and Requirements* subprocess are given in Section 4.1.

Input

- Quality Model
- Service Consumer Requirements

Output

- Requirements Specification

2.2.2. Generate Choreography Specification from CTT Models

The process for generating an initial choreography specification begins with the retrieval of relevant user task models. The models are identified using the requirements, from the requirements specification, which are processed through the matching functions of the TEDDiE service. CTT model structures and rules developed for CHOReOS are used to generate a CTT-based choreography specification. Finally, an initial BPMN2 choreography specification is automatically generated in MagicDraw.

More details about the *Generate Choreography Specification from CTT Models* subprocess are given in Section 4.2.

Input

- CTT Models
- Requirements Specification

Output

- BPMN2 Choreography Specification

2.2.3. Refine Choreography Specification

This process consists of two main activities: an initial choreography specification is refined by the choreography designer to produce a coherent process model with well-defined service contracts and quality properties by producing SoaML and Q4BPMN specifications. Then, a validation process is carried out to determine if functional aspects need to be altered and whether quality properties need to be relaxed. Concerning the usage of SoaML in CHOReOS, we recall that services can be discovered or implemented from scratch. To this extent (following a generative approach), SoaML is used as a specification for implementing services and hence it represents a useful input for the TDD activity. More details about SOaML and Q4BPMN are given in Section 4.3.

To further refine the choreography specification, the choreography designer is also asked to precisely specify information on in/out data at choreography task level. In fact, with reference to Figure 2.3, BPMN2 Choreography Diagrams give the possibility to provide a detailed specification of *initiating* messages and/or *return* messages of the choreography task. Specifically, at task level, the messages precisely describe the data (operation names and related parameters) that have to be exchanged to initiate

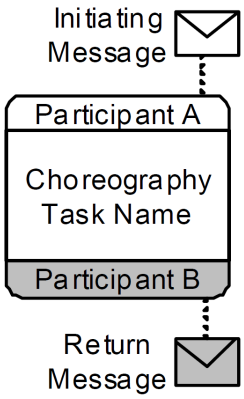

Figure 2.3: Choreography task data specification

the tasks. Further details on how the data specification is used to select concrete services are given in Section 4.4. Finally, the refinement process ends with an analysis of role (participant) dependencies. More specifically, a collection of dependency analysis algorithms extract such dependencies from the BPMN2 diagrams and analyze them to uncover interaction patterns among roles, as well as model vulnerabilities. As a result of this analysis, a report is automatically generated. This report consists of a valuable input for the choreography designer to reason about the way the specified choreography roles interact. In particular, based on involved domain constraints and business rules, the choreography designer may decide to further refine or even redesign parts of the choreography specification.

Input

- BPMN2 Choreography Specification
- Requirements Specification

Output

- BPMN2 Choreography Specification
- SoaML Specification: Specification of participants and service contracts
- Q4BPMN Annotation: This model is given to provide non-functional annotations of the BPMN2 specification.
- QoS Monitoring Rules: Monitoring rules for detecting both real and potential QoS violations at runtime.

2.2.4. Bind Service Descriptions to Choreography Specification

This activity is responsible for discovering services and binding them to choreography participants. If required services are not available in the service base, then they need to be developed (or adapted from existing ones).

More details about the *Bind Service Descriptions to Choreography Specification* subprocess are given in Sections 4.4 and 4.5.

Input

- Abstraction & Service Base
- BPMN2 Choreography Specification
- SoaML Specification

- Q4BPMN Annotation

Output

- BPMN2 Choreography Specification

2.2.5. Assess Quality of Choreography Specification

The quality of the choreography specification is assessed in two steps. The first step consists of a simulation approach that relies on LTS models enriched with QoS and on QN models. The second step consists of a dependency-centric analysis that evaluates change-impact and helps identify services that are more likely to experience side-effects.

More details about the *Assess Quality of Choreography Specification* subprocess are given in Section 4.6.

Input

- BPMN2 Choreography Specification

Output

- Choreography Quality Assessment

2.2.6. Synthesize Coordination Delegates for Choreography Specification

A model-based synthesis process is defined for the automatic realization of service choreographies out of the final choreography specification and a set of discovered services and/or newly developed ones. Since a choreography is a network of collaborating services, the notion of a coordination protocol becomes crucial. In fact, it might be the case that the collaborating services, although potentially suitable in isolation, when interacting together can lead to *undesired interactions*. The latter are those interactions that do not belong to the set of interactions modeled by the choreography specification and can happen when the services collaborate in an uncontrolled way. That is, although a given set of services, if coordinated in the right way, can be used to realize the specified choreography, they prevent choreography realizability if either coordinated in a different way or left uncontrolled. To deal with this problem, additional software entities, called *Coordination Delegates* (CDs), are generated and interposed among the services participating in the specified choreography in order to prevent possible undesired interactions. Thus, the intent of CDs is to coordinate the interaction of the participant services in a way that the resulting collaboration correctly realizes the specified choreography. This is done by exchanging suitable *coordination information* that is automatically generated out of the choreography specification.

More details about the *Synthesize Coordination Delegates for Choreography Specification* subprocess are given in Section 4.7.

Input

- BPMN2 Choreography Specification

Output

- Coordination Delegates

2.2.7. Perform Choreography Offline Testing

In this step, the CHOReOS dynamic development process foresees the choreography offline testing. Firstly, we distinguish between online testing and offline testing. Online testing consists of determining whether a system complies with its intended behavior during its real life operation (i.e., in the production

environment) [GGvD10]. On the other hand, all testing strategies and techniques, analysis (e.g., trace files), and simulations applied before the deployment of the system in its production environment, correspond to offline testing. Tests invoking a system in the development or testing environment are also considered offline testing.

In CHOReOS, we conduct choreography offline testing in two parallel steps: one of them deals with functional requirements testing, while the other deals with non-functional requirements testing. Functional requirements testing involves testing the behavior of the web services that were selected to play the roles defined in the choreography model (Section 4.8.1). Web service offline testing comprises the following tasks: writing and running compliance tests (Section 4.8.1.1), writing and running integration tests (Section 4.8.1.2), and writing and running acceptance tests (Section 4.8.1.3). In turn, non-functional requirements testing involves testing the scalability of the very same set of web services (Section 4.8.2). More specifically, it comprises the task of writing and running scalability tests (Section 4.8.2.1). The whole activity of choreography offline testing is supported by the Rehearsal framework [BMKM12], which is detailed in D4.2.2 [CHO12d].

More details about the *Perform Choreography Offline Testing* subprocess are given in Section 4.8.

2.3. Enact Choreography Process

The Choreography Enactment Process has two main goals (see Figure 2.4). The first one is to automate the deployment of services on a cloud infrastructure by managing virtual machines and installing and configuring required software. The second goal involves wiring up services so that they properly communicate when necessary.

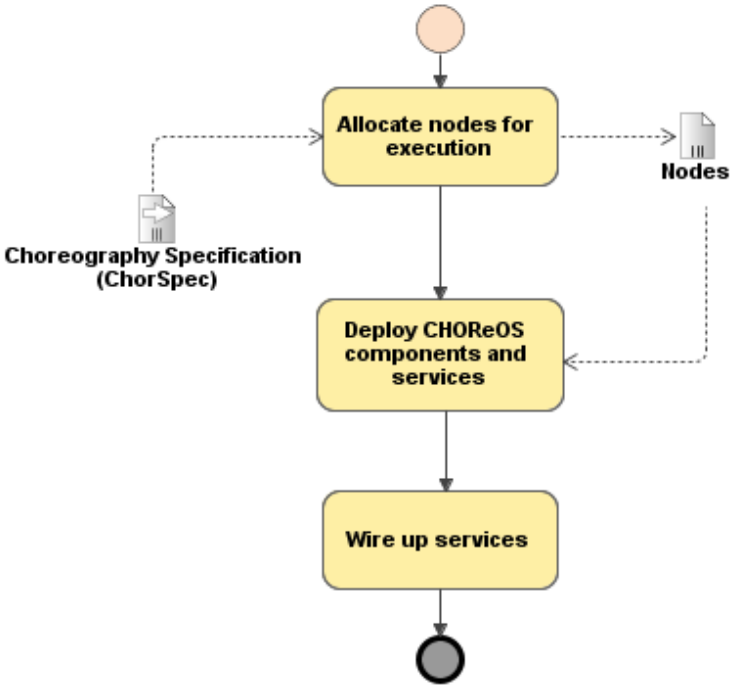

Figure 2.4: Enact Choreography

The Choreography Enactment Process is primarily realized by the Enactment Engine software component, which is able to enact large-scale choreographies in a fully-automated way. The Enactment Engine component exposes a Restful API, which takes as input a deployment-oriented choreography specification in the form of an XML file (ChorSpec). In Figure 2.5, we show a UML representation of the ChorSpec and its relationships. Given the importance of the ChorServiceSpec class, we briefly

describe each of its attributes:

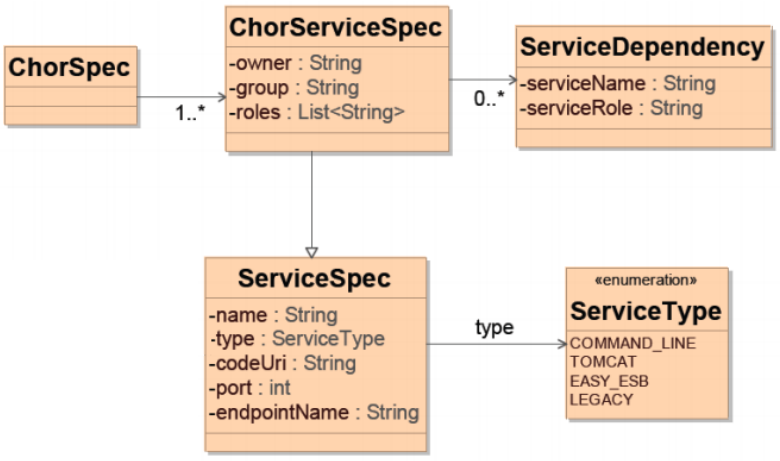

Figure 2.5: Enactment Engine REST API data model

- **name:** a unique name within the choreography specification;
- **owner:** the organization that holds the infrastructure where the service must be deployed;
- **group:** services in the same group will be deployed in the same cloud node;
- **roles:** list of roles implemented by the service;
- **dependencies:** list of ServiceDependency entries; each entry describes the name of the dependency (matching the name attribute), and the role provided by the dependency;
- **type:** the type of the service, according to the ServiceType enumeration
- **codeUri:** the location of the source to be deployed. If type is LEGACY (services that are not being deployed specically for this choreography, but are already available somewhere), this attribute will represent the complete service endpoint;
- **port:** the TCP port used by the service. Mandatory if type is COMMAND LINE;
- **endpointName:** the endpoint suffix after deployment. For example, if the service is deployed at `http://<some_ip>/choreos/service`, then the endpoint becomes `choreos/service`. If type is LEGACY, then the endpoint becomes empty.

The Enactment Engine relies on a series of subcomponents (see Figure 2.6). In the following, we briefly describe the role of each of these subcomponents. More information about the Enactment Engine and its subcomponents can be found on deliverable D3.2.2 [CHO12a].

- Service Deployer: implements a RESTful API that enables the deployment of a given service within the middleware distributed service bus. The Service Deployer asks the Node Pool Manager for a CHOReOS node with the needed configuration, and the client receives a bus entry point to the service. Each cloud environment needs one associated Service Deployer and only the Enactment Engine has to invoke Service Deployers. Finally, each Service Deployer instance is configured with corresponding Chef (configuration management system) and cloud provider accounts.
- Node Pool Manager: implements a RESTful API offering node management services to other Middleware components. The main entity created by this API is the CHOReOS Node, which represents a virtual machine created in a cloud infrastructure

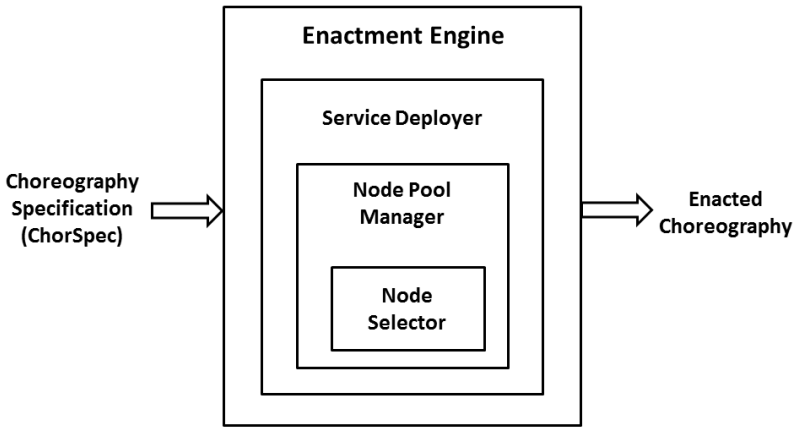

Figure 2.6: Enactment Engine Components

- Node Selector: Defines the policy that governs the allocation of nodes. For now, the available policies are `ALWAYS_CREATE` and `ROUND_ROBIN`. An organization may implement its own policy by implementing the NodeSelector interface.

In the following, we present the main tasks involved in the enactment of a choreography within CHOReOS: (i) allocate nodes for execution, (ii) deploy the DSB and the Coordination Delegates, and (iii) wire up services.

2.3.1. Allocate nodes for execution

Based on the Choreography Specification input (ChorSpec) and in the node allocation policy defined in the properties file of Service Deployer, the required nodes are created. In particular, the Node Pool Manager configures the created nodes by installing CHOReOS components, starting and stopping services, creating and changing configuration files, and so on. Such configurations are applied by Chef in a suitable node chosen by the Node Pool Manager. Clients can retrieve a node with a given configuration by calling Node Pool Manager and providing the desired "Chef recipe" name as input. Recipes specify node configuration and must be uploaded to the Chef server beforehand. The requirements for node creation can be expressed by high-level attributes, such as `<memoryImpact>high</memoryImpact>`, or with low-level attributes, like `<ram>1024</ram>`.

Input

- Choreography Specification (ChorSpec)

Output

- Nodes

2.3.2. Deploy CHOReOS components and services

The Node Pool Manager configures the created nodes by installing CHOReOS components (such as the DSB); deploying, starting and stopping services; and setting up configuration files. Such node configuration is applied by Chef in a suitable node chosen by the Node Pool Manager. The UML sequence diagram in Figure 2.7 depicts the interactions that occur among the Enactment Engine component and its subcomponents when two different services need to be deployed each in one node.

For a thorough explanation of the steps involved in the collaboration, please refer to CHOReOS Deliverable D3.2.2 [CHO12a].

Input

- Nodes

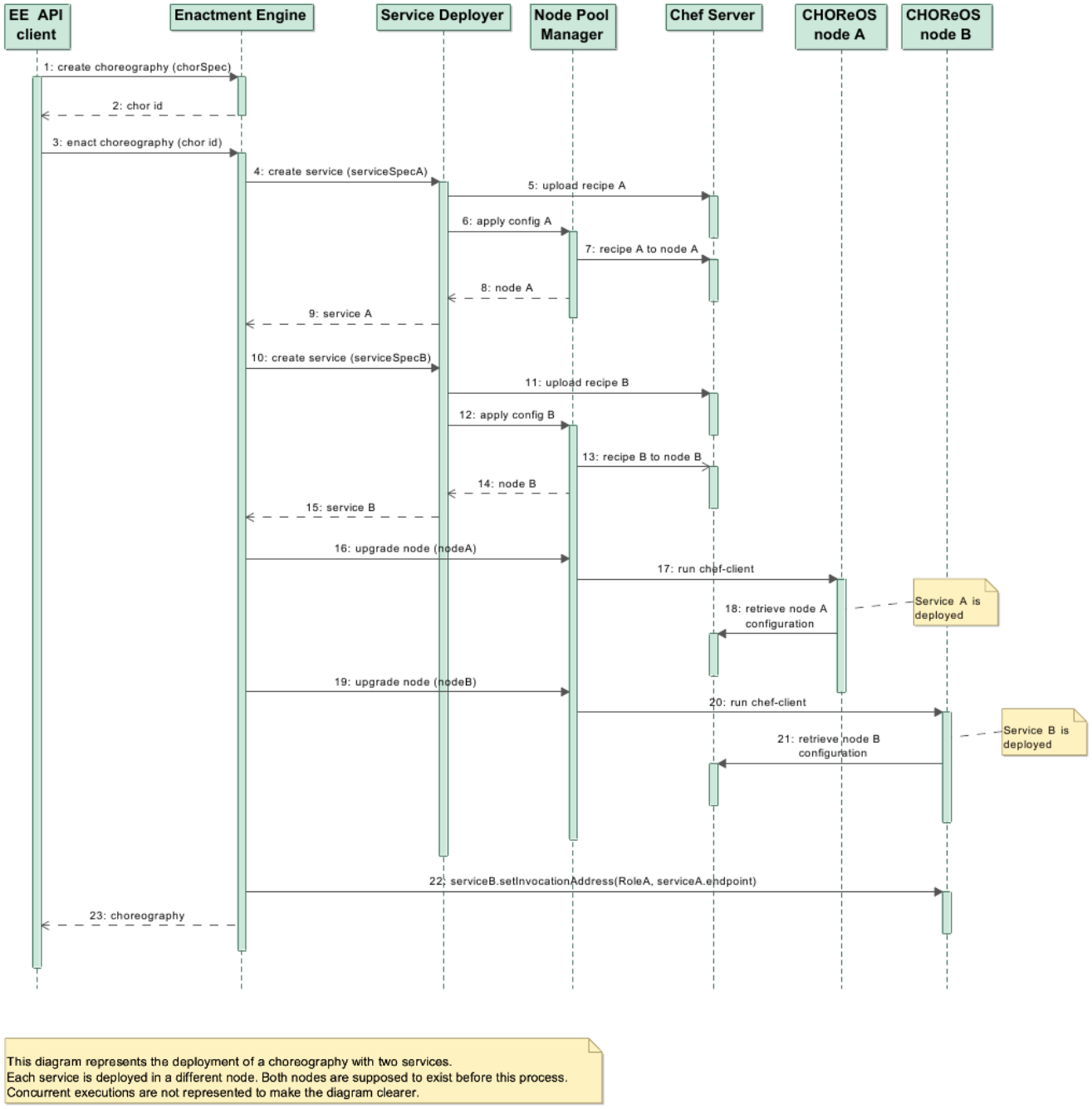

Figure 2.7: Enactment Engine deploying two different services

2.3.3. Wire up services

In a service composition, some services depend on other services. A service that depends on other services is a *dependent service* (client service), and the service that provides functionality to the dependent service is the *dependency* (provider service). In a simple service composition, such dependency relations are hardcoded on dependent services. However, decoupling the dependent service implementation from the actual dependency endpoint is a good practice, which enables dynamic adaptation. Moreover, dependency hardcoding is not possible on cloud environments, since we do not know service addresses before deployment. Therefore, in the CHOReOS environment, each dependent service is declared as depending on *roles* rather than on other service implementations. The dependent service must receive the actual dependency endpoint of a service fulfilling the required role through the

`setInvocationAddress` operation. Such an operation must be implemented by every dependent service. The Enactment Engine will use `ServiceDependency` data to know which calls it must perform to the `setInvocationAddress` operation of participant services. Thus, the Enactment Engine will be able to tell, for example, `Service A` that it must use `Service B` as `Role1`, where `ServiceB` is an endpoint URI. In this way, the CHOReOS middleware provides a *dependency injection*[1] mechanism to wire up service dependencies.

2.4. Analyze Choreographies Runtime Process

This process becomes active after choreography enactment and consists of three main sub-processes, namely (i) *QoS Monitoring*, (ii) *QoS Prediction*, and (iii) *V&V at RunTime* (see Figure 2.8). Since (i) and (iii) have been developed in the scope of WP4, we provide an overview of their essentials in this deliverable and the interested reader can refer to D4.2.2 [CHO12d].

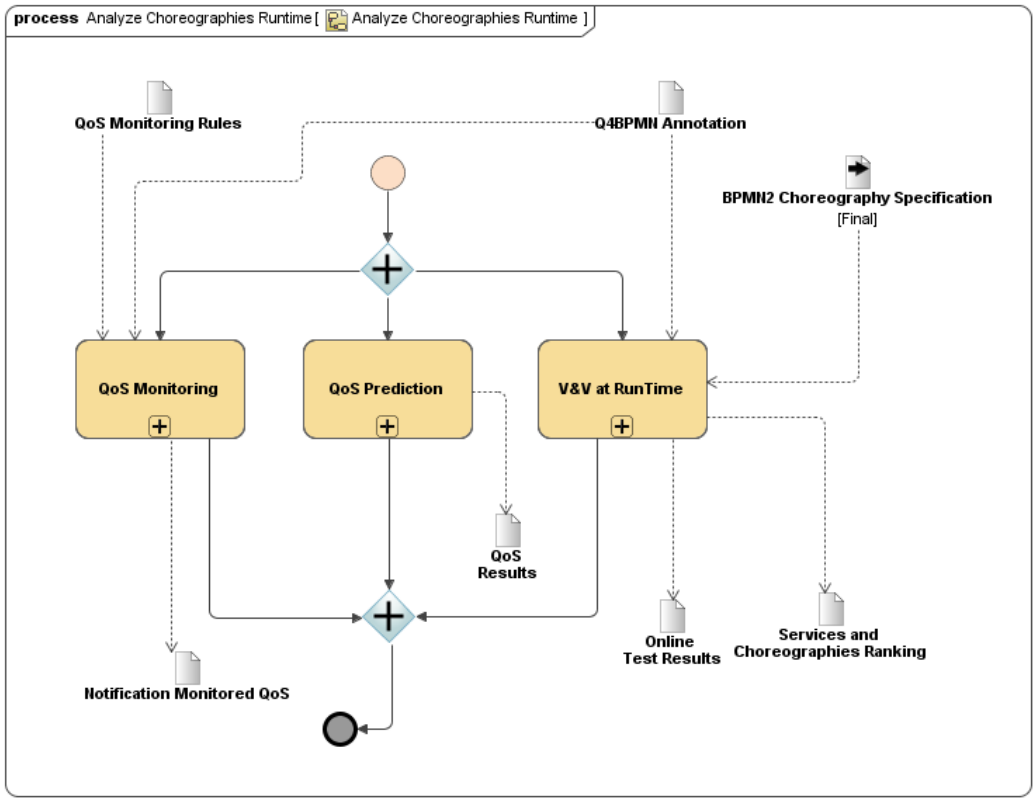

Figure 2.8: Analyze Choreographies Runtime

2.4.1. QoS Monitoring

This step consists in evaluating the data from an enacted choreography at runtime in order to inspect and analyze QoS parameters. As a result, it is possible to either notify or predict runtime anomalies. This process takes into account both the Q4BPMN Annotation and the QoS Monitoring Rules defined during the refinement of the Choreography Specification. The output of this activity is the notifications about the monitored QoS parameters. Such notification can concern both violations of agreed QoS levels or warnings about potential anomalies that may occur in the monitored context. A detailed discussion of the implementation of this task can be found in deliverable D4.2.2 [CHO12d].

[1] Dependency Injection pattern as defined by Martin Fowler: http://martinfowler.com/articles/injection.html

An example of the activities performed within this task are reported in the live demo video released at M24[2] on the CHOReOS Multi-source monitoring framework [HBC+12]. Specifically, the demo shows how the monitoring framework detects and correlates violations of QoS properties defined on top a choreography specification. The considered scenario is based on the Passenger-Friendly Airport Use Case Demo [CHO12b].

Input

- Q4BPMN Annotation
- QoS Monitoring Rules

Output

- Notification Monitored QoS

2.4.2. QoS Prediction

We propose a QoS prediction approach [GN12] that aims at providing the middleware with an estimation for the values of multiple QoS parameters, including service response time, capacity, and failure rates. The prediction assumes the existence of concrete web services that each have local QoS values. At the conceptual level, we organize the prediction in three stages:

1) the definition of an algorithmic representation for Web Service Compositions (WSCs);
2) the definition of a set of aggregation rules for QoS estimation;
3) the elaboration of an automated QoS prediction solution.

Figure 2.9 describes the sequential ordering of these stages within our conceptual approach. In the following, we briefly discuss each process stage. It is important to notice that until now we mainly focused on the prediction of the service response time. However, the solution we propose can be extended to other QoS parameters.

Definition of an algorithmic representation of WSC

We propose to capture Web Services Composition within the concept of Hierarchical Services Graphs (HSG). These graphs define a WSC as an implementation of a business process over Web Services. HSGs are made of three layers capturing from the top to the bottom: the interactions between Web Services operations, the Web Services that are part of the composition, and the machine servers on which Web Services are run.

At the top level, we can have two types of interactions between operations: those that define a unique business process and those defining multiple processes. HSGs derived from the first class will be called *Single Process HSGs* (SPHSG), while the latter will be called *Multiple Processes HSGs* (MPHSG).

Definition of aggregations rules

The usage of aggregations rules is the main thesis in our QoS prediction approach. We assume that given any WSC, we are able to find a set of rules that state how to derive QoS of sub-compositions from the ones measured on web services operations (*decomposability thesis*). These set of rules are those we refer to as aggregation rules.

The decomposability thesis has been assumed in many previous works [CMSA02]. For defining aggregation rules in general, one takes advantage of the business process point of view of WSCs.

[2] see at http://www.choreos.eu/bin/Discover/videos

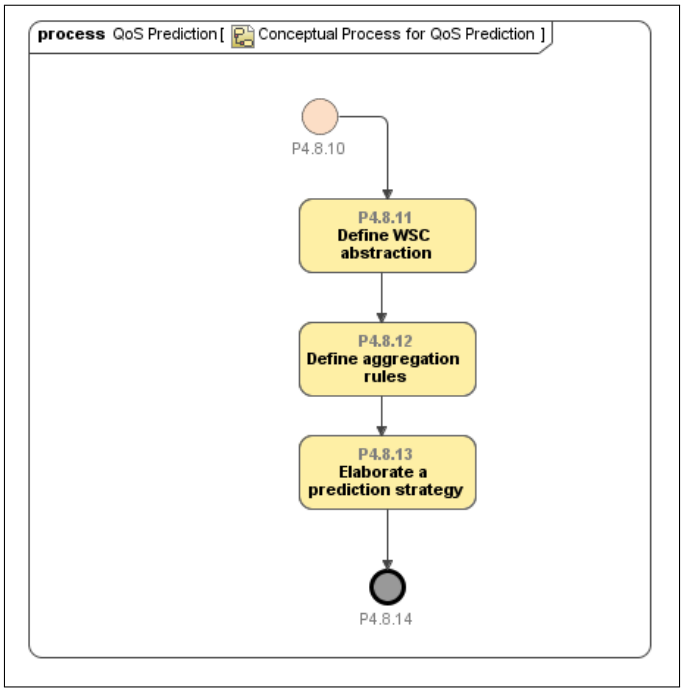

Figure 2.9: Conceptual QoS prediction process

Aggregations rules state how QoS can be aggregated from general business process patterns (sequence, parallel split etc.). Our main contribution at this stage is to propose some rules that handle communication time within multiple business processes.

Elaboration of an automated QoS prediction solution

We assume here that we have a HSG and QoS input values at the level of its operations and communications. For predicting the QoS of the HSG, we consider separately SPHSG and MPHSG.

On SPHSG (single process case), we proceed by graph reduction. The idea is to recursively substitute (at the operation level) the subgraphs that it contains by a single node that aggregates the QoS of subgraphs nodes. More details about this technique can be found in [GN12, CMSA02].

On MPHSG, we propose to use linear programming. This is performed in two stages: a first one where we generate (from the HSG) a linear program whose solution is the QoS estimation that we are looking for; and a second stage where the generated linear program is run using a classical linear solver. More details about this procedure can be found in [GNM12]. The global prediction process is summarized in Figure 2.10.

2.4.3. V&V at RunTime

V&V at RunTime foresees the proactive launching of selected test cases that evaluate the real system within the services' real execution environment. By extending testing from the laboratory to the field during normal operation, on-line testing can help in the timely detection of unrevealed inconsistencies as well as of functional and non-functional failures.

In general, any V&V activity at runtime can occur periodically or after some relevant event, according to the established SOA test governance process [DBP12]. In addition, the application of V&V of services at runtime requires dedicated mechanisms (e.g., dedicated software infrastructures) for proactively launching selected test cases within the services' real execution environment.

Specifically, this process takes into account both the *BPMN2 Choreography Diagrams* as functional notations to specify choreographies, and the Q4BPMN Annotation as reference specification of non-

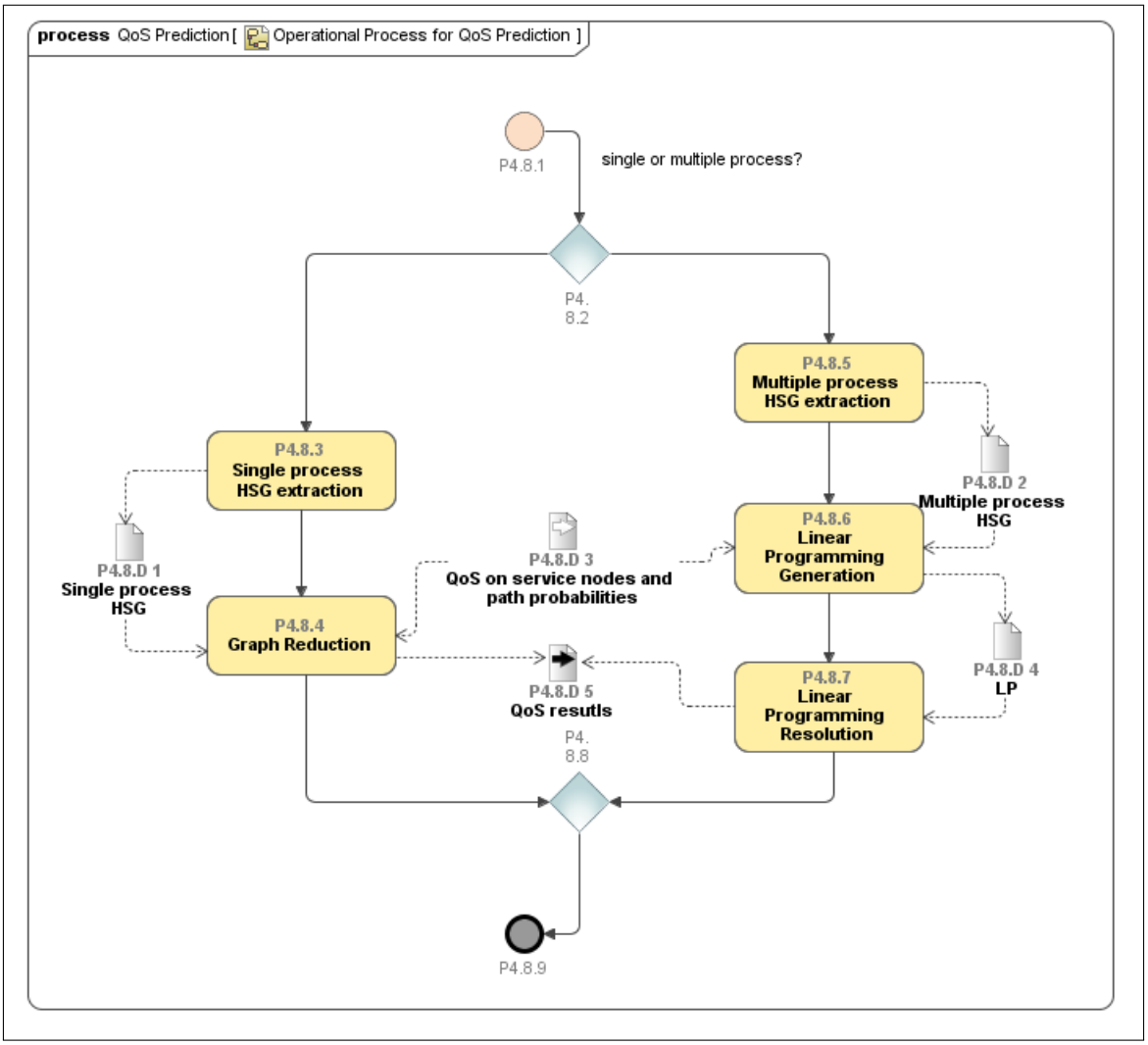

Figure 2.10: Operational QoS process

functional aspects. From these specifications, the V&V activities at runtime define (both functional and non-functional) test cases and their expected outputs. The outputs of the process include the result of the online testing sessions and also some ranking parameters estimating the trustworthiness of both services and choreographies running on-the-field. Further details about these concerns are reported in [CHO12d], and [DBP12].

Input

- BPMN2 Choreography Specification
- Q4BPMN Annotation

Output

- Online Test Results
- Services and Choreographies Ranking

3 ULS dimensions, use-cases and development process activities

This chapter reports the definition of the ULS dimensions we have defined in CHOReOS and their relationships with the CHOReOS development activities (and related IDRE components), as well as the WP6-8 case studies. The nature of these dimensions has been inspired by the outcomes of the deliverable D1.2 [CHO11] where we surveyed the main challenges posed by the Future Internet. Their definition in Table 3.1 (and related scale) is based on the CHOReOS objectives, and hence the SOA paradigm with specific focus on service-oriented systems realized as choreographies of services discovered in the Future Internet, and related supporting middleware.

	Scalability Dimension	**Scale**
1.	Number of abstractions and represented services in the Service Base	- Service Base with a large number of abstractions (10^3 to 10^6), representing an UL number of services (10^5 to 10^9)
2.	Number of service instances in a single choreography (i.e., for a given type of service playing a specific role, we may have an UL number of instances)	- UL number of service instances per choreography (10^3 to 10^6), leading to a UL number of CD instances per choreography
3.	Frequency of messages exchanged by the services in time period	- UL number of invocations per seconds (10^3)
4.	Number of thing-based services (actuators and sensors)	- UL number of actuators and sensors (10^6)
5.	Changing the number of service instances, CD instances, actuators and sensors over time	- Scaling up/down by adding/removing up to 10^3 service instances, CD instances, actuators and sensors per choreography

Table 3.1: Definition of Scalability Dimensions

Table 3.2 considers the ULS dimensions in Table 3.1 and relates them with the CHOReOS development activities by specifying the impact on the associated IDRE components we are developing. Moreover, the table also describes how the IDRE components will be assessed against the targeted dimensions. Note that, since WP2 activities focusses on business services, having a UL number of IoT impacts on the middleware, and hence on components developed in WP3 (specifically, Registration Manager, Query Manager and Sensor Access Middleware). Thus, in Table 3.2 we refer to deliverable D3.2.2 [CHO12a] for details.

Scalability Dimension	Process Activity	Impact	Assessment
1.	AoSBM (part of the *Discover Services* sub-activity of the *Bind Service Descriptions to Choreography Specification* activity)	Clearly, the ULS nature of the service base has to be tested with respect to the number of services stored in the service base. This includes: service types and service instances.	Given the fact that real-world data provide descriptions in the order of no more than few thousand, we must artificially populate the service base with significant numbers of service descriptions, in order to stress test it. We will devise meaningful benchmark queries to perform the query time testing.
1.	AoSBM (part of the *Discover Services* sub-activity of the *Bind Service Descriptions to Choreography Specification* activity)	A second aspect of the ULS capabilities of the service base that has to be tested is the amount of abstractions it can sustain. This involves both functional and non-functional abstractions.	As above, the stress test must be performed with artificial data.
1.	Analyze Choreography Runtime	Deliverable D4.2.2 [BAP12] includes dedicated discussions on which are the issues introduced by this specific dimension, and how each approach included within the V&V and Governance Framework at run-time deals with it.	Please refer to Sec. 3.2.4, Sec. 3.3.1, and Sec. 3.5.1 in D4.2.2.

2. and 3.	*Synthesize Coordination Delegates for Choreography Specification*	The ULS dimensions 2 and 3 do not directly affect the synthesis process when generating the coordination logic (i.e., CD models) from a choreography specification. In fact, the synthesis process generates a CD model per choreography participant (acting as consumer), at most. Thus, since we reasonably assume choreography specifications that do not have a UL number of participants, the number of CD models cannot be UL. However, in a choreography execution we may have multiple instances of a service that play the role of a specified participant. Thus, the overall number of service consumer instances can be UL, and this leads to an UL number of CD instances. We recall that the CD instances exploit the coordination algorithm that, taking as input the CD models, permits to exchange additional information to coordinate the interaction behavior of the services involved in the choreography. In brief, coordination information concerns BLOCKing, UNBLOCKing, and ACKnowledging messages. As detailed in the deliverable D1.3 [ARS12b], the overhead due to the exchange of coordination information among the coordination delegates is negligible with respect to the exchange of business messages related to the services interaction behavior. Moreover, leveraging the theoretical results in [Lam78], in D1.3 we argue that the number of additional messages is the strictly necessary minimum that one can exchange to ensure correct distributed coordination. However, with respect to the seminal algorithm proposed in [Lam78], our version of the algorithm is specifically conceived to solve a different coordination problem at the basis of a choreography specification (e.g., we consider much more complex interactions such as forking interactions, some of which to be joined at later execution points). Thus, although our algorithm has been defined by respecting all the principles dictated in [Lam78] and we can still leverages its theoretical results (to argue on the minimality of the number of required additional messages), an assessment of our implementation of the algorithm against the considered ULS dimension is required as described in the right-hand column.	We will assess the implementation of the coordination algorithm against this dimension by checking that, increasing the number of service instances and hence of CD instances, the overhead due to the exchange of additional messages grows linearly with the number of CD instances.

2.	Write and run scalability tests (from the *Perform Choreography Offline Testing* activity)	During an integration test involving a UL number of services, it might be required a UL number of Service Mocks and/or Message Interceptors.	As mocks and interceptors are locally deployed during tests, the impact of creating a UL number of these elements must be verified.
2. and 3.	QoS Prediction (from the *Analyze Choreographies Runtime* activity)	The number of participants, services and service instances, as well as the frequency of users requests directly impact mean service response time, maximal response time, throughput, availability, and capacity.	We will call the QoS prediction algorithm while making vary the scaling dimensions within the sets defined above. Then we will build a profile stating the behavior of QoS values on scaling dimensions. A typical expected answer can be that the Mean Service Response time increases linearly with the frequency of users requests.
2.	Analyze Choreography Runtime	Deliverable D4.2.2 [BAP12] includes dedicated discussions on which are the issues introduced by this specific dimension, and how each approach included within the V&V and Governance Framework at run-time deals with it.	Please refer to Sec. 3.2.4, Sec. 3.3.1, Sec. 3.4.1, and Sec. 3.5.1 in D4.2.2.
2.	*Enact Choreography*	Number of services and CDs to be deployed should be ULS as described in the "Scalability Dimension" table. We have to test the enactment of a choreography with this quantity of services and CDs to be deployed.	We will measure the scalability as the time to deploy an increasing amount of services within a choreography with a correspondent amount of resources (cloud nodes). We can use a large number of services and nodes to this assessment.

2.	Deploy CHOReOS components and services (from the *Enact Choreography* activity)	The number of participant organizations can be UL in the same scale as the number of service instances. This is due to service types that may have multiple instances, and each instance may belong to a different organization. Since different organizations present heterogeneous cloud environments, it is important to assess the deployment of a choreography across multiple organizations (i.e., the enactment engine must know how to interact with different cloud providers).	We will evaluate the simultaneous deployment of services in different clouds. However, probably we will not have a large number of different clouds to assess.
3.	Write and run scalability tests (from the *Perform Choreography Offline Testing* activity)	It must be possible to generate UL frequency of messages in order to support testing this dimension. The Scalability Explorer features a load generator that should address this issue.	We must test the Scalability Explore's load generator to check whether it is able to produce the desired UL frequency of messages. Furthermore, we must also test whether Rehearsals web service dynamic client is able to handle such workload.
3.	Analyze Choreography Runtime	Deliverable D4.2.2 [BAP12] includes dedicated discussions on which are the issues introduced by this specific dimension, and how each approach included within the V&V and Governance Framework at run-time deals with it.	Please refer to Sec. 4.2.1, Sec. 4.3.7, and Sec. 4.4.2 in D4.2.2.
4.	IoT management (WP3 activity)	This dimension impacts on components developed in WP3, i.e., Registration Manager, Query Manager and Sensor Access Middleware.	Please refer to Section 1.3.2 in deliverable D3.2.2 [CHO12a] for details.

5.	*Enact Choreography*	Changing the number of service instances and CD instances over time is an important aspect to be considered by the Enactment Engine. We have to test the deployment of new services when the choreography is already running and processing requests.	This we will be assessed by the deployment of a large amount of services within a choreography, followed by several updates. After each update, we verify if the choreography is working as expected.
5.	Analyze Choreography Runtime	Deliverable D4.2.2 [BAP12] includes dedicated discussions on which are the issues introduced by this specific dimension, and how each approach included within the V&V and Governance Framework at run-time deals with it.	Please refer to Sec. 3.4.1, and Sec. 3.5.1 in D4.2.2.

Table 3.2: Impact and assessment of ULS dimensions

Table 3.3 discusses to additional dimensions that, although not directly related to the ULS dimension in Table 3.1, are of importance to the requirements specification phase, to the enactment of choreographies and to the analysis of change-impact.

Dimension	**Process Activity**	**Impact**	**Assessment**
Number of service consumer instances specifying requirements	Requirements collection (part of the *Requirements Specification* activity)	The number of source service consumers who impose requirements on a single choreography and the number of service consumers who impose requirements on all choreographies in a single application.	This will necessitate in-situ evaluation of collection of requirements from stakeholders by domain experts in the 3 use-case domains.
Number of choreography instances	*Enact Choreography*	It is not UL, but may be high. For example: if the federal government (in a big country, like Brazil) has to instantiate the choreography to each city in the country, there will be thousands of instances (in Brazil, about 5 thousands). Since each instance multiplies the choreography scale, it is important to consider this aspect.	We will measure the scalability as the time to deploy an increasing amount of choreographies with a correspondent amount of resources (cloud nodes). We can use a large number of services and nodes to this assessment.

Complexity of choreography structure and degree of interconnection	Analyze Role Dependencies in Choreographies (from the *Refine Choreography Specification* activity) and Analyze Service Dependencies in Choreography (from the *Assess Quality of Choreography Specification* activity)	When doing inter-choreography change impact analysis, we compute *choreography call-graphs* (to know which choreographies call a certain one) and we also traverse choreography BPMN specifications (to know the different possible flows of execution). At the same time, choreographies of the Future Internet tend to be highly interconnected and quite complex, in the sense that there will be a high number of possible flows of execution and message exchanges among participants in each choreography. Given this context, it might be the case that several choreographies will need to be taken into account when performing our analysis, thus directly impacting the performance of such analysis. We intend to address the issue by developing fast algorithms to calculate choreography call-graphs and traversing choreography structures.	We intend to assess the performance of this analysis by generating a series of synthetic choreography call-graphs (which tend to be trees) and synthetic directed graphs that would represent choreography structures. We will then run our analysis on such graphs and quantitatively evaluate it.

Table 3.3: Impact and assessment of dimensions related to choreography enactment and choreography change-impact analysis

In CHOReOS, each use case (WP6, WP7 and WP8) is developing a demonstrator to assess the use case against (among other aspects) the ULS dimensions we are considering. The following three Tables 3.4, 3.5 and 3.6 specify, for the three use cases, what is the expected scale in real-life scenarios, and what is the scale that will be reached by the demonstrators for the assessment.

WP6 Use case - The Table 3.4 illustrates the ULS aspect of the WP6 use case. The reported numbers represent real-life context. The WP6 demonstrator is focused on choreography-based business services linked to the air traffic flow management. It also addresses IoT (as UL dimension) through the use of sensors and actuators, both installed inside the airport and carried by passengers' smartphones. It thus provides an efficient solution for highlighting significant variations of Things to be considered by the system along the runtime.

Scalability Dimension	**"Passenger-friendly Airport" use case assessment**
1. N° abstracted services	In the service base, more than 1000 Web services are available. In order to refine the choreography and for demonstration purpose, a subset (approximately 50) has been already implemented.
2. N° service instances	The number of service instances is proportional to the number of passengers in the Airport (10^4 a day) plus the specialization of shops and companies (different luggage handling, different stand and gate management, different shops, different hotels, different travel agencies, and so on).

3. Frequency of messages	The frequency of exchanged messages is proportional to the number of passengers in the Airport (10^4 a day) plus the specialization of shops and companies (different luggage handling, different stand and gate management, different shops, different hotels, different travel agencies, and so on).
4. N° thing-based services	10^3 sensors and actuators inside the Airport plus 10^5 sensors and actuators inside the passengers MIDs. For demonstration purpose, 500 sensors and actuators are instantiated then deployed.
5. Changing N° service instances	Before the arrival of the airplane, only the sensors and actuators of the Airport are requested (10^3). Then, in the second part of the choreography, the passengers disembark from the airplane, and about 10^3 new sensors and actuators are added to the system at the same time. For demonstration purpose, the number of services instances grows from 100 to 500.

Table 3.4: WP6 use case *VS* ULS dimensions

WP7 Use case - The ULS numbers for the real applications are as in Table 3.5, depending of course on the size of the issued market chain. The WP7 demonstrator will not address IoT (at least as UL dimension), therefore even if the scale can be the same as for business services it will not be developed as such, since it was agreed within the project consortium that ULS IoT will be addressed mainly by the WP6 use case, and with a lower scale by the WP7 use case. Though not limited by the choreography-based solution of the use-case demonstrator, the actual number of business service instances will be of some thousands, since it is a reasonable value to demonstrate the ULS property of the CHOREOS solution (i.e., the IDRE). This avoids aside issues not relevant for the proof-of-concept (e.g., not enough available resources at a reasonable cost on the used cloud infrastructure). However, the numbers of involved people, service instances and exchanged messages, as well as their variations along the run-time, can be (pre-)configured as desired and needed.

Scalability Dimension	**"Adaptive Customer Relationship Booster" use case assessment**
1. N° abstracted services	Not UL number of service abstractions ($\sim$30) giving place to a UL number of service instances.
2. N° service instances	The number of instances is determined by the number of clients, suppliers, shopping carts, etc. Of these, at least the number of clients will be UL. Depending on the choreography and the service type, the number of service instances may vary widely (10 to 10^5).
3. Frequency of messages	The frequency of requests is approximately proportional to the number of clients (which is UL). A few hundred per day for clients. In the order of 10^4 for marketing managers and directors.

4. N° thing-based services	The number of atomic services can be higher than 10^4, but will not be addressed as an ULS dimension in this use case.
5. Changing N° service instances	The number of clients (and their related Basic Communication Devices and Shopping Assistant Apps) in a store change over time. The arrival rate can be changed and we can simulate the simultaneous entrance of all the clients at the same time (the clients then leave the store at random time) as by configuration and needed.

Table 3.5: WP7 use case *VS* ULS dimensions

WP8 Use case - Table 3.6 provides numbers for the five scalability dimensions as expected to be covered in real-life deployment. In the WP8 context we are working on providing the DynaRoute demonstrator both on lab and real-life demonstration. While in the lab environment assessment in being conducted with ULS numbers, the trial (real-life demo) supports lower numbers (as a proof of concept). Still, the actual deployment and enactment of choreographies is the main aspect of the real-life trial, which is expected to scale as the supporting infrastructure (IDRE, cloud) scales. Main ULS aspects within the DynaRoute scenario are the number of citizens (users) with their smartphones (10^6-10^9), the number of sensors both on smartphones and GPS devices (carried on taxis) (10^6-10^9) and location-based provisioning of services (10^4-10^6).

Scalability Dimension	**"DynaRoute" use case assessment**
1. N° abstracted services	DynaRoute scenario considers services that are location and/or language dependant, like the taxi booking service. In a large scale deployment, these kind of services are available in a large number of variations. Thus, a number of services in the magnitude of 10^3-10^5 is expected. As a consequence of these variants, in every-day life we expect the corresponding service base abstractions to be large as well. For the sake of a real-life prototype most of these services are implemented within the project, so its number in the Service Base would be lower.
2. N° service instances	Services like "friend proximity notification" that are part of DynaRoute scenario, are mapping concrete users with groups of users. The expected service instances are proportional to applications users, which may reach 10^4 for a city and scaled up to 10^6 -10^9 for countries and the globe. Here we note that in the prototype the number of users will be low.
3. Frequency of messages	The communication rate between services acting in the DynaRoute choreography is expected to be in the magnitude of tens of messages per minute during usage peak.

4. N° thing-based services	In DynaRoute the number of sensors is proportional to the number of Mobile Internet Devices (MIDs) using the DynaRoute choreography. DynaRoute choreography utilizes location-based services (for instance, at city level). Number of sensor-based services at the region of a city can be 10^4-10^6 which is considered as Ultra Large. These ULS numbers are expected to be simulated, while the real-life demos will include small number of things ($<$100) for a working prototype to be exposed.
5. Changing N° service instances	The population of participating things (sensors and actuators) is going to vary over time, because many of them rely on MIDs. Activation and deactivation of MIDs is common in the large communication environment that covers as a city, which is the case for DynaRoute case study. Another effect of using DynaRoute application through a MID is that the service instances are going to dynamically change over time, since services like "personal calendar" and "contact directory" are utilized in this scenario. In the real-life demonstrator we are considering a small ($<$20) and controlled population, but in the lab simulation we can assess varying numbers of things and service instances.

Table 3.6: WP8 use case *VS* ULS dimensions

4 Detailing development process activities

This chapter provides a more detailed description of the process activities outlined in the previous chapter.

4.1. Specify Goals and Requirements

The process for specifying goals and requirements in CHOReOS allows for expressing functional and quality requirements on services and choreographies (see Figure 4.1). In order to reflect the needs and desired qualities of users, service requirements are used as inputs into the start of the process. CHOReOS then provides tool support for calculating requirements similarity which assists the domain expert in identifying the choreography need and the requirements to realise it. To do this we use techniques designed to overcome incompleteness and ambiguity in natural-language requirements, building upon work from the SeCSE project[1]. The domain expert can also document facts, or assumptions, about the domain that the choreography will be realised in. Such domain knowledge is used to reason about the achievement of quality requirements later on in the process. Finally, a quality model developed for CHOReOS is used to associate end-user quality requirements with QoS of the services to be choreographed. The model includes underlying quality definitions, measures and metrics.

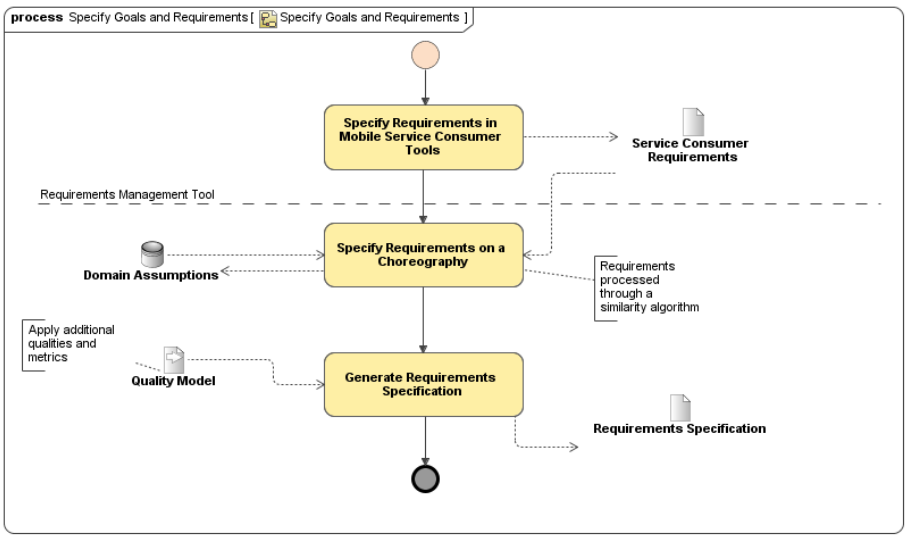

Figure 4.1: Specify Goals and Requirements

4.1.1. Specify Requirements in Mobile Service Consumer Tools

The choreographies developed in CHOReOS must take into account the users requirements and the context they are expressed in. Therefore, it is important that we consider service user requirements

[1] http://www.secse-project.eu/

during choreography design and also at run-time. Whilst this activity is outside the domain expert-centric development process, we have proposed a possible approach using an iPhone application, as reported in D2.1 [ARS12a]. The output from this activity, the service consumer requirements, is used by the domain expert in the next activity to reflect the service needs of the end-users in the choreography specification.

Input

- Service Consumer Requirements

Output

- Service Consumer Requirements

4.1.2. Specify Requirements on a Choreography

The domain expert identifies service consumer requirements related to a user goal, i.e., the need for the choreography, and also adds their own requirements in the requirements management tool. These requirements need to be grouped in a meaningful way in order to define boundaries for the choreography and search spaces for the services. The tool enables the domain expert to consolidate the requirements by assessing them for similarity. This operation is performed by the calculate similarity web service as described in D2.1 [ARS12a].

In addition, new capabilities enable the domain expert express facts, or assumptions, about the domain the choreography will operate in. These domain assumptions are stored in a database to enable reuse within the domain expert specification, and also later on in the process to reason about the achievement of quality requirements during choreography design and refinement. Also, it is possible to identify new or changed requirements in the database that may lead to a revised grouping of requirements on the choreography. Changes to the requirements on the choreography, as expressed by domain experts, provide a direct input into the runtime support for the adapting the choreography (developed in WP3).

Input

- Domain Assumptions
- Service Consumer Requirements

Output

- Domain Assumptions

4.1.3. Generate Requirements Specification

The domain expert commits the requirements group by generating the requirements specification on the choreography. Part of this process includes the application of the CHOReOS Quality Model, reported in D2.1 [ARS12a], which extends the user expressed requirements with related service qualities and associated measures and metrics. The requirements specification provides inputs into user task model retrieval and choreography design later on in the process.

Input

- Quality Model

Output

- Requirements Specification

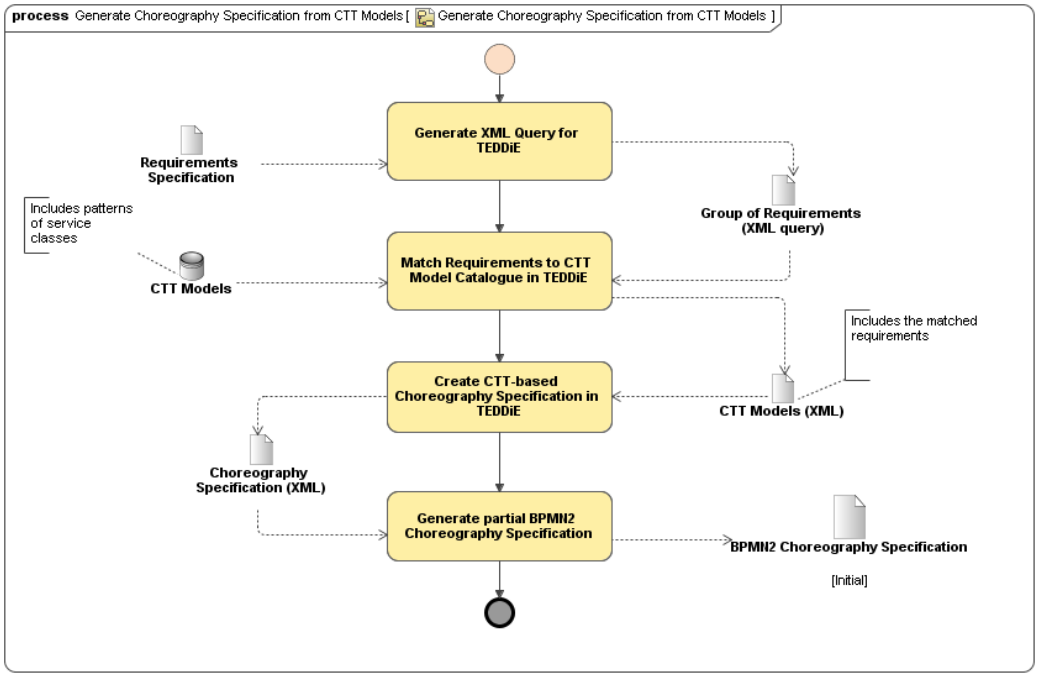

Figure 4.2: Generate Choreography Specification from CTT Models

4.2. Generate Choreography Specification from CTT Models

In CHOReOS, user tasks models are used to generate an initial, but partial, choreography specification (see Figure 4.2). A user task model defines a reusable task structure that encapsulates well-defined functionality for a recurrent design problem [KA92].They describe structured activities that are often executed during the interaction with a system, influenced by its contextual environment, and performed to attain goals. Our approach uses the CTT (ConcurTaskTrees) task modeling formalism [PS02], chosen because it adapts an engineering approach to user task models. The more complete and precise semantics of CTT has the potential to support automated choreography specification and service query formulation more effectively than other user task modeling formalisms.

We apply CTT models to fulfill two important roles. The first role is to specify important constraints on a service choreography. For example, each task imposes important constraints on when sub-tasks can occur, thus informing the required and permitted orderings of service invocation within a given choreography. As such, the user tasks and temporal associations between sub-tasks provide choreography design guidance in the form of reusable patterns of service classes.

The second role is to link reusable knowledge about the domain in which the task occurs to the requirements on software services to invoke in the task. We aim to exploit the knowledge to discover services that meet the users' goals and needs, choreograph services to support cognitive and interactive tasks more effectively, and invoke services that provide resources that users need. For example, a task model might specify the role for a class of software service that service consumers and choreography designers do not know about or have overlooked.

4.2.1. Generate XML Query for TEDDiE

The requirements and desired qualities on a choreography, as expressed in the requirements specification, are organised to form a set of terms describing the user problem. These terms are specified as an XML query according to the schema designed for input into the TEDDiE service [ARS12a].

Input

- Requirements Specification

Output

- Group of Requirements (XML query)

4.2.2. Match Requirements to CTT Model Catalogue in TEDDiE

The group of requirements in the generated XML query are processed using the TEDDiE service which applies sophisticated information retrieval techniques to identify relevant CTT models. The approach includes term disambiguation to determine the meaning of terms in the requirements and query expansion to add terms with similar meanings. These techniques seek to reduce ambiguity and improve completeness to identify the most suitable CTT models.

The expanded set of requirements is then matched to a searchable catalogue of generic user task models, also represented in XML. To do this, we build on a library of domain-independent models developed in S-Cube, the EU-funded Network of Excellence for Software Services [SC04]. In simple terms, each model in the library describes the tasks undertaken by users and machines in commonly-occurring class-level tasks such as reserving, purchasing and travelling. Each embodies important contextual knowledge needed to specify a service choreography. Within the CTT models, each user task is specified with natural language descriptions of the task in context and the user goal achieved by completing the task. These descriptions facilitate the matching between the CTT models and the requirements.

In order to demonstrate that the requirements have been accounted for, the TEDDiE service records which requirements have been matched with which CTT models. The relationships between the requirements and the retrieved CTT models are then specified in the generated XML results file.

Input

- CTT Models
- Group of Requirements (XML query)

Output

- CTT Models (XML)

4.2.3. Create CTT-based Choreography Specification in TEDDiE

The knowledge from the retrieved CTT models is processed in TEDDiE to formulate an initial choreography specification in XML. As the CTT models express reusable class- level tasks rather than complete business processes, the requirements on a complete choreography are likely to return more than one CTT model from the matching process. Indeed, following a review of the CTT models in the catalogue we found that their coverage commonly relates to two high-level choreography tasks. Simple structures and rules are used to map between the semantics of the CTT models (introduced and explained in D2.1 [ARS12a]) and the BPMN2 choreography diagrams.

In terms of structure, the CTT models follow a 3-tier hierarchy: the top level user task/goal, decomposed into the interaction level and finally into the application level. For example, Figure 4.3 shows the main user task *find hotel*, decomposed into interactions between the user and the applications such as *provide hotel search details*. Application tasks at the interaction level are decomposed into application level sub-tasks, such as *detect current location* and *retrieve date*.

The XML-based CTT models also include a further facet called a service class. The application subtasks are mapped to these service classes (33 of which were already specified in the catalogue). For example, *retrieve date* is associated with *online calendar* and *date checker* service classes. These

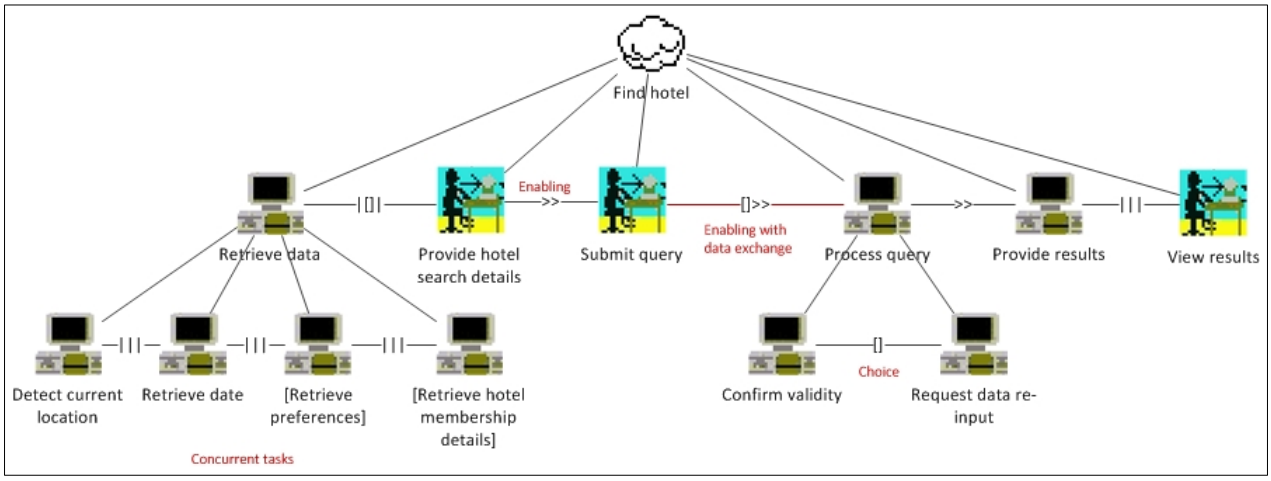

Figure 4.3: Example CTT model: Find Hotel

service classes contain queries, expressed in the CHOReOS query language (WSBQL), which are pre-loaded as inputs to the discovery module of the CHOReOS service base.

A set of rules for mapping a CTT model to a BPMN2 choreography model have been defined, as described in Figure 4.4. For example, the *enabling with information passing* CTT operator denotes the boundary between choreography tasks, showing where information from one logical activity (i.e. message exchange between participants) is passed onto the next activity.

Temporal operator		
Name	**Symbol**	**CTT mapping to BPMN2**
Enabling with information passing	[]>>	At the interaction level – denotes the boundary between choreography tasks. It shows where information from one logical activity (i.e. message exchange between participants) is passed onto the next activity.
Enabling	>>	At the interaction level – denotes the sequence of application and interaction tasks within a choreography task. At the application task level – denotes the sequence and important ordering between service classes within a choreography task.
Concurrent tasks	|||	At the application task level – shows that service classes within a choreography task can be invoked concurrently
Choice	[]	At the application task level – denotes choice and the possibility to flow back to a prior choreography task

Figure 4.4: Set of mappings between CTT and BPMN2 models

Input

- CTT Models (XML)

Output

- Choreography Specification (XML)

4.2.4. Generate partial BPMN2 Choreography Specification

In the final part of the process, the CTT-based choreography specification from TEDDiE is imported into MagicDraw via the XML file. The XML schema contains separate sections of the choreography model, related to each of the CTT models that were retrieved from the requirements expressed on the whole choreography. The BPMN2 choreography tasks and relationships are automatically generated

in MagicDraw. Each choreography task includes attributes captured from the CTT models, i.e. the requirements and the application tasks with their associated temporal constraints and service class queries.

Input

- Choreography Specification (XML)

Output

- BPMN2 Choreography Specification

4.3. Refine Choreography Specification

The automated process for retrieving CTT models and generating an initial choreography specification provides the choreography designer with a partial choreography model with associated design advice. Therefore, a degree of refinement is required to produce a coherent intermediate process model with well defined functional and quality properties (see Figure 4.5). This includes (i) the specification of satisfaction arguments and the expression of quality properties using the Q4BPMN notation, and (ii) the specification of *service contracts* and *participants* using SoaML. In particular, SoaML diagrams are referenced from BPMNv2 choreography tasks.

As already anticipated, SoaML can be used when services are not found in the service base and need to be implemented (from scratch or by adapting existing ones) by adopting the test-driven development approach described in Section 4.4.3. To this end, service contracts define the specification of the agreement between the provider and consumer in terms of required/provided operations and sequence diagrams with input/output data. Moreover, for each participant composite application components diagrams are used to denote the types of service and/or request ports. Types are specific service interfaces that each participant must implement to adhere to service contracts that it participates either as a provider or as a consumer. Participants follow a matching service interface pattern, meaning that for every provided request port on a participant there must exist at least one compatible (of the same type) service port on some other participant. For details on the usage of SoaML models we refer to deliverable D6.2 [CHO12b].

The refinement process is completed with a validation process to determine if functional aspects need to be altered and whether quality properties need to be relaxed.

4.3.1. Refine Functional Properties of the Choreography Specification

The initial BPMN2 choreography specification is partial, in that it comprises sections of the choreography model that need to be integrated. Therefore, the choreography designer reviews the sections of the generated model and integrates them, possibly adding loops and gateways where appropriate. Also, the choreography designer adds participant names, as it is not possible to derive this information from the generic user task models. Similarly, it is not possible to determine the choreography task names, although the names of the associated interaction level tasks from the CTT models are provided as prompts. For example, *retrieve data*, *provide hotel search details* and *submit query* from the CTT model inform the choreography task name *request hotels* chosen by the choreography designer.

Along with the initial choreography specification, the requirements specification is an input into the refinement process. Functional requirements from the domain expert also provide the choreography designer with functional design advice. The end result of refining the functional constructs and properties is the intermediate BPMN2 choreography specification together with the SoaML specification.

Input

- BPMN2 Choreography Specification

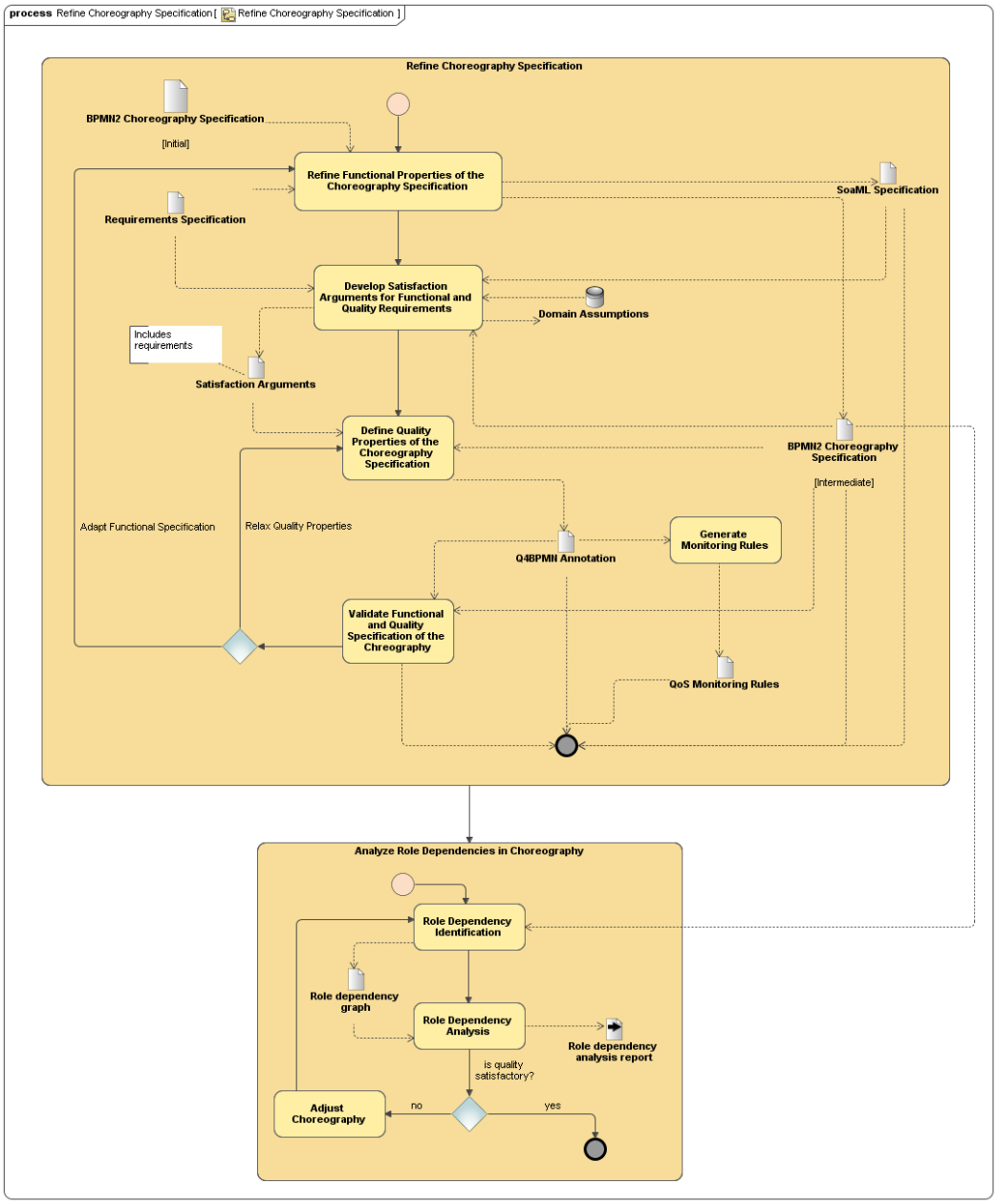

Figure 4.5: Refine Choreography Specification

- Requirements Specification

Output

- BPMN2 Choreography Specification
- SoaML Specification

4.3.2. Develop Satisfaction Arguments for Functional and Quality Requirements

The choreography designer develops satisfaction arguments as a means to reason about the relationships between user expressed quality requirements and lower-level system requirements. Satisfaction arguments were first reported in [Jac95] and applied in the REVEAL requirements method [HRH01] to recognize the role of assumptions in specifications. The combination of domain knowledge and system

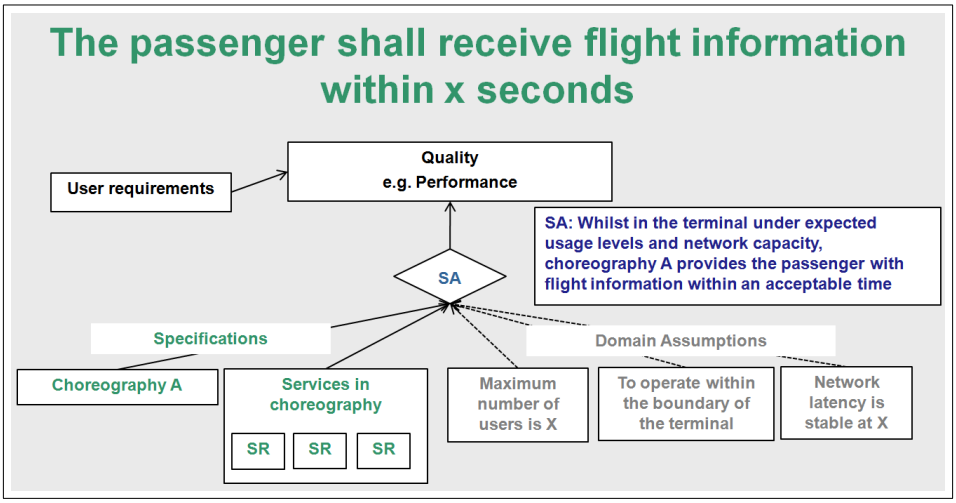

Figure 4.6: Example satisfaction argument

knowledge (specification) is the basis of the satisfaction argument, which serves as a framework to show that a requirement will hold. When applied in CHOReOS, we can say that given the properties of the domain and the choreography specification it is possible to show that a given quality will hold.

There are three inputs into this task, the choreography specification, the user requirements and the domain assumptions specified earlier in the process by the domain expert. The choreography designer uses these inputs to begin developing the satisfaction arguments, as described using the following example shown in Figure 4.6. Taking the user requirement *The passenger shall receive flight information within x seconds*, the QoS most likely to be prioritised by the service consumer is performance. A number of relevant facts or domain assumptions specified by the domain expert and stored in a repository can be accessed, or indeed be added to by the choreography designer, e.g. the service-based system is to operate within the boundary of the airport terminal, there is a given maximum number of users, and there is a given network latency. The choreography itself is part of the specification, along with the system requirements on the services to be monitored. The choreography designer specifies the service requirements, guided by the functional requirements and lower level functional elements brought into the specification from the CTT models.

The final output of this task is the completed satisfaction arguments, which include the requirements, the relationships between them and associated domain assumptions and specifications. The satisfaction arguments can be revisited and assessed after the choreography specification has been evaluated and the services discovered later on in the process.

Input

- Domain Assumptions
- Requirements Specification
- BPMN2 Choreography Specification

Output

- Domain Assumptions
- Satisfaction Arguments

4.3.3. Define Quality Properties of the Choreography Specification

Starting from the intermediate specification of the choreography, from the satisfaction arguments, and from the requirements it includes, the choreography designer defines the non-functional annotations of

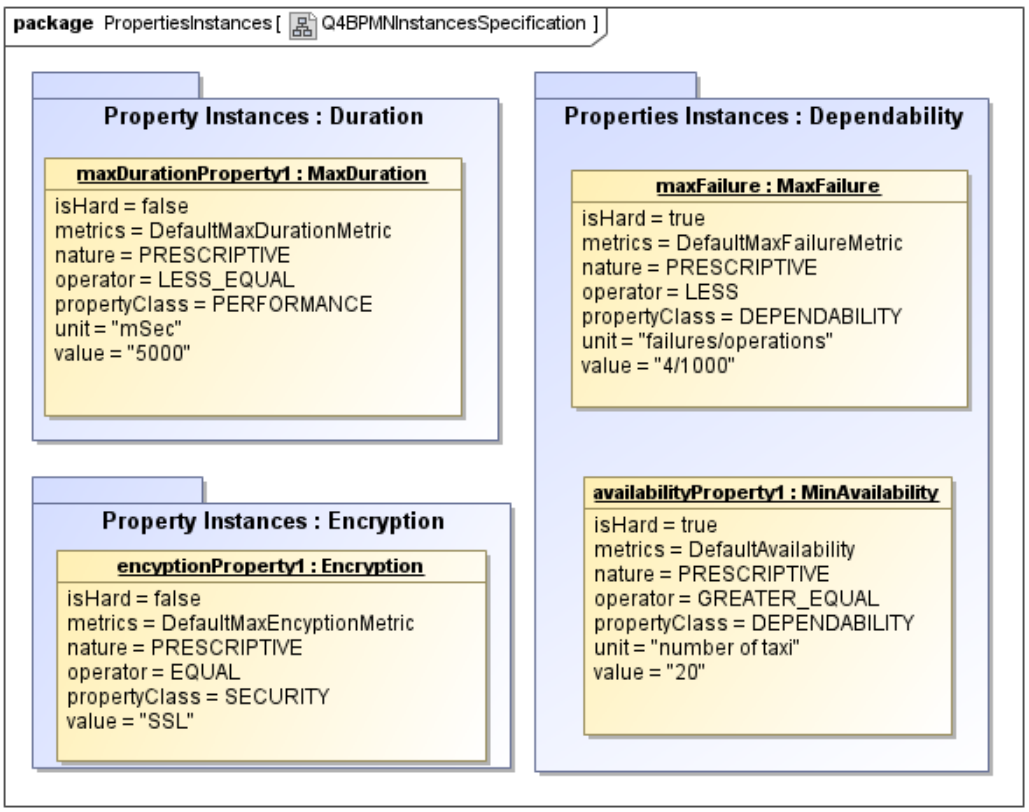

Figure 4.7: Q4BPMN**: Example of Property Instances**

the choreography specification expressing them in Q4BPMN [BAP12, BBC+12].

Q4BPMN provides an extension for BPMN supporting both designers and analysts by annotating the choreography diagram with quality requirements. Specifically, Q4BPMN implements the concepts defined by the Property Meta-Model (PMM) [DPB+11] as a UML profile.

In this sense, Q4BPMN does not bind any specific definition of the properties it can model. Rather, it leaves to the domain experts the possibility to define at design-time, their most suitable formalization of a non-functional facet, and linking it with an appropriate metric (i.e. domain specific way for computing such a property). Nevertheless, as deeply discussed in [BAP11, BAP12], and [BBC+12], Q4BPMN includes some default definitions of non-functional properties such as : performance properties which are related to the amount of resources of the system; security properties that determine the trustability of the system, and allow the exchange of private data with reasonable privacy; dependability properties concerning the capability of the system to behave reliably, even under stress or error conditions.

Furthermore, according to the PMM definitions, Q4BPMN describes properties that can be ABSTRACT, DESCRIPTIVE, or PRESCRIPTIVE. An ABSTRACT property indicates a generic property that does not specify a required or guaranteed value for an observable or measurable feature of a system. A DESCRIPTIVE property represents a guaranteed/owned property of the system. A PRESCRIPTIVE one indicates a system requirement. In the two latter cases, the property is defined taking into account a relational operator with a specified value.

Figure 4.7 depicts some example of non-functional properties specified using the Q4BPMN notation. For the sake of completeness, we remark that the definition of any concrete instance of a non-functional parameter is a difficult task that strictly depends on the specific scenario/demonstrator considered. Thus, the definition of specific non-functional parameters for the demonstrator scenarios are not part of this report.

As introduced in Section 4.3.2, satisfaction arguments can be used in order to identify the quality requirements. Such design requirements could be further refined so that they can be translated into

quality specification expressed as Q4BPMN that can be exploited by both choreography and service designers. Currently we are working for refining the transformation steps required in order to define such a mapping.

Input

- BPMN2 Choreography Specification
- Satisfaction Arguments

Output

- Q4BPMN Annotation

4.3.4. Validate Functional and Quality Specification of the Choreography

The Q4BPMN annotations are transformed into an intermediate representation called KLAPER models [GMRS08]. This representation is useful in order to analytically evaluate the non-functional defined non-functional constraints. Specifically, we refer to a KLAPERS SUITE [CDF+11] for evaluating the tolerance of the choreography for the constraints on the tasks. This activity has been developed in WP4, thus refer to deliverable D4.2.2 [BAP12] for details.

Input

- BPMN2 Choreography Specification
- Q4BPMN Annotation

4.3.5. Generate Monitoring Rules

Although the non-functional annotations are already useful by themselves helping the service providers meet the choreography requirements elicited by the choreography designer (see Section 4.3.2), their full potential is expressed when they are toolchained in a set of operations.

Specifically, at design time the Q4BPMN annotated constraints can be automatically translated into monitoring rules aimed at detecting violations of the annotated properties at runtime (for a more detailed description refer to [BBC+12]). In addition, some explicit constraints could have an impact also on tasks that have no annotations, for example this is the case of constraining global properties of a choreography. We have enriched the approach with the capability to derive and monitor also implicit constraints, as we describe in [BBC+13]. Those monitoring rules for implicit constraints will be used at runtime by the monitor for the possible generation of warnings [BAP12].

Input

- Q4BPMN Annotation

Output

- QoS Monitoring Rules

4.3.6. Analyze Role Dependencies in Choreography

This subprocess involves making sense out of the existing dependencies among the roles (participants) specified in a choreography. As pointed out in deliverable D2.1 [ARS12a], choreographies can be seen as social networks in which participants share information and collaborate through message exchanges. In particular, ULS choreographies of the FI will originate large social networks. Centrality measures presented in deliverable D2.1 [ARS12a] offer a straightforward and yet powerful way to

assess the prominence of participants in a choreography. Moreover, we intend to support the choreography designer in analyzing vulnerability [BIK+07] and recognizing patterns of relationship among participants. The diagram depicted in Figure 4.5 provides an overview of the steps and data elements (inputs and outputs) involved in the process of role dependency analysis. We plan to use the framework Jung[2] and a generic dependency analysis library we developed to accomplish the analysis of role dependencies. More details about the methods and tools for role dependency analysis will be provided in the next deliverable D2.3 ("CHOReOS dynamic development process: methods and tools").

Input

- BPMN2 Choreography Specification

Output

- Role dependency analysis report

4.4. Bind Service Descriptions to Choreography Specification

As previously mentioned, this activity is responsible for discovering services and binding them to choreography participants. The service discovery employs the functionalities of the Abstraction-oriented Service Base Management (AoSBM), the main concepts of which have been discussed in D2.1 [ARS12a]. The AoSBM allows to organize available service descriptions with respect to functional/non-functional abstractions. Then, it allows to lookup for services based on queries specified in terms of a query language that was developed during the 2nd year of the project. The AoSBM facilities can be used either at design time via a corresponding GUI (see D5.3.1 [SOA09]), or at runtime through the AoSBM API (see D3.2.2 [CHO12a]). In general, different instances of the AoSBM may be deployed in a distributed setting. In the case where certain services required for a choreography are not available, we assume that the missing services can be developed by means of a test-driven approach and then registered to the AoSBM (see Figure 4.8).

4.4.1. Discover Services

The `Discover Services` activity employs the query engine facility of the Abstraction-oriented Service Base Management (AoSBM) towards the discovery of services that can play the roles specified in a BPMN2 Choreography Specification. The AoSBM query engine takes as input queries written in terms of the Web Service Base Query Language (WSBQL). WSBQL is a language proposed specifically for the CHOReOS AoSBM. The WSBQL storage model is specified in Section 6.1, while its syntax and semantics are given in Section 6.2 of the Appendix.

The WSBQL query expressions may be generated by AoSBM clients, based on a list of operations that concern a particular choreography participant. For each operation, information on task-level in/out data is extracted from the BPMN2 choreography specification by exploiting a dedicated functionality offered by the synthesis processor (see Section 4.7). This information contributes to the generation of the functional part of WSBQL expressions by also accounting for message parameters (see Section 6.2 of the Appendix), and allow for addressing operation data mapping between the BPMN2 specification and concrete services to be discovered and bound.

The output of WSBQL queries can be exploited towards dealing with the issue of data mappings as follows. By construction, the functional abstractions that are returned by a WSBQL query contain mappings between the operations of the services that are represented by a functional abstraction. Moreover, the functional abstraction contains mappings between the messages of the mapped operations. As discussed in detail in D2.1 [ARS12a], the message mappings map the leaves of the messages; they are constructed by the functional abstractions hierarchical clustering algorithm that was proposed in D2.1

[2] http://jung.sourceforge.net/

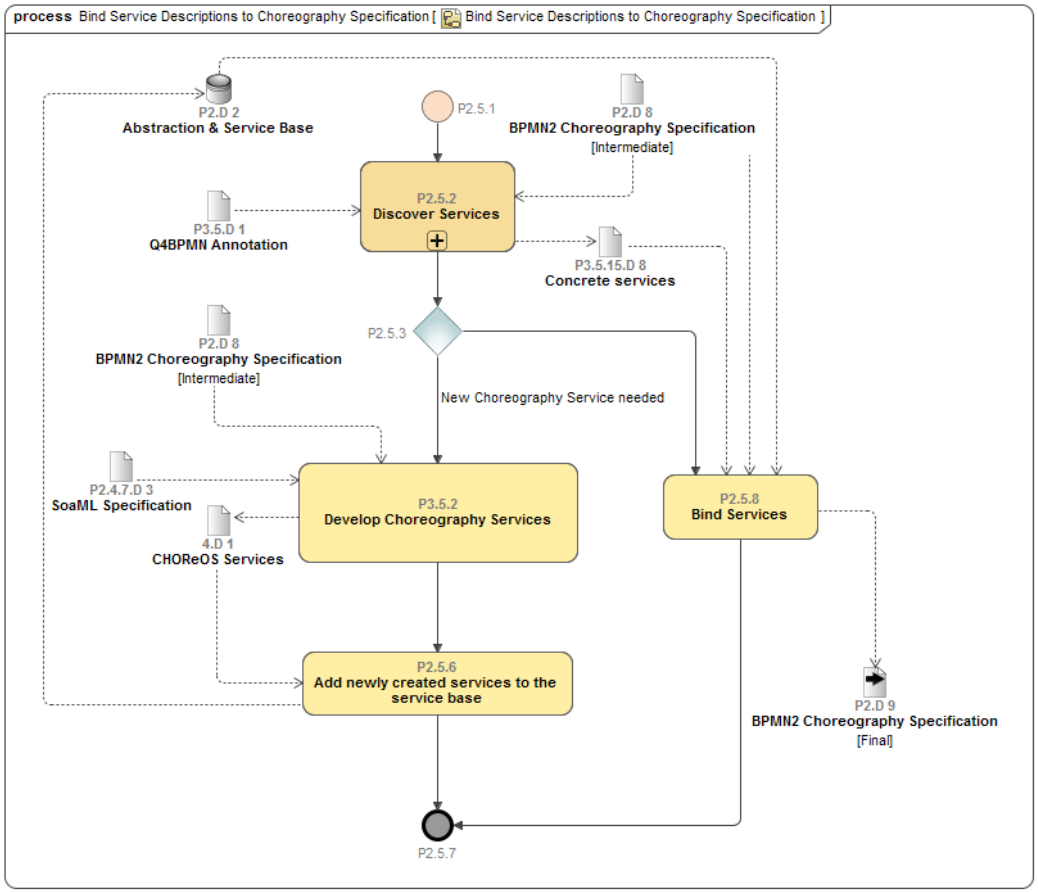

Figure 4.8: Bind Service Descriptions to Choreography Specification

by solving the maximum weighted matching problem in a bipartite graph. The exploitation of the message mappings can be done in two ways: (1) the choreography developer can use them to resolve data mapping issues at the choreography level, (2) the choreography adaptation middleware specified in D3.2.2 [CHO12a] shall rely on them to enable the data translation needed for the adaptation of services at the middleware level.

A Q4BPMN model may also be considered for generating the non-functional part of WSBQL expressions.

The WSBQL query expression is parsed by the AoSBM query parser and, following, executed by the AoSBM query engine. The result returned by the query engine contains the descriptions of the functional and/or the non-functional abstractions that satisfy the constraints specified in the WSBQL query. The result may further contain the descriptions of the concrete services (represented interfaces, instances) that are represented by the retrieved abstractions. As discussed in Section 6.2 the contents of the query result are specified in the query expression (i.e., return only abstractions or return abstractions along with their represented service descriptions).

The query results can be browsed at design time, through the AoSBM GUI, or processed at runtime via the AoSBM API. Browsing the query results consists of exploiting the information that is provided for the retrieved abstractions and the concrete services that are represented by the retrieved abstractions. The end-user traverses the abstractions by expanding and collapsing the abstraction nodes. The expansion of a functional (resp., non-functional) abstraction reveals information regarding lower-level functional (resp., non-functional) abstractions, or information concerning the represented concrete services. More details about the `Discover Services` activity are given in Section 4.5.

Input

- BPMN2 Choreography Specification
- Q4BPMN Annotation

Output

- Concrete services

4.4.2. Bind Services

The ultimate goal of the discovery is to bind the choreography participants to concrete services. To bind a concrete service to the choreography specification, the `Discover Services` functionality is triggered for each choreography participant. Such a participant is characterized by functional and non-functional requirements/constraints. The purpose of the `Discover Services` functionality is to identify concrete services that meet these constraints. To improve the efficiency of the discovery process, hence dealing with the UL scale of the service base, the searching can be performed over the abstractions search space instead of the services search space.

Note that the discovery process can return for each choreography participant a set of candidate concrete services. To select the proper service instance, according to functional requirements/constraints, an extra step is required. This step is performed by the synthesis processor and checks if, within the set of service candidates, there is a service whose behavior is suitable to play the role of the participant. Having selected abstractions and service instances for each role, the choreography specification is updated accordingly to account for these selections in the subsequent synthesis steps.

As detailed in Section 2.3, and in particular in Section 2.3.3, we recall that dependent service implementation is decoupled from the actual dependency endpoint by declaring dependencies on roles (rather than on implementation) and by setting the actual invocation address at runtime. Moreover, a participant may be bound directly to a concrete service, or to a functional abstraction that represents a set of concrete services. Binding a participant to a functional abstraction shall further enable the adaptation of the actual concrete service that is represented by the functional abstraction as detailed in D3.2.2 [CHO12a].

Input

- Abstraction & Service Base
- BPMN2 Choreography Specification
- Concrete services

Output

- BPMN2 Choreography Specification

4.4.3. Develop necessary choreography services

In the CHOReOS dynamic development process, it can be the case that required services are not found in the service base. As a consequence, such services need to be implemented (from scratch or by adapting existing ones). In CHOReOS, we support the use of Test-Driven Development (TDD) and provide adequate tools for it. Taking as input the BPMN2 choreography specification together with the SoaML specification, TDD consists of a design technique that guides software development through testing [Bec03]. TDD can be summed up by the following iterative steps (Figure 4.9):

- Write an automated test for the next functionality to be added into the system;
- Run all tests and see the new one fail;

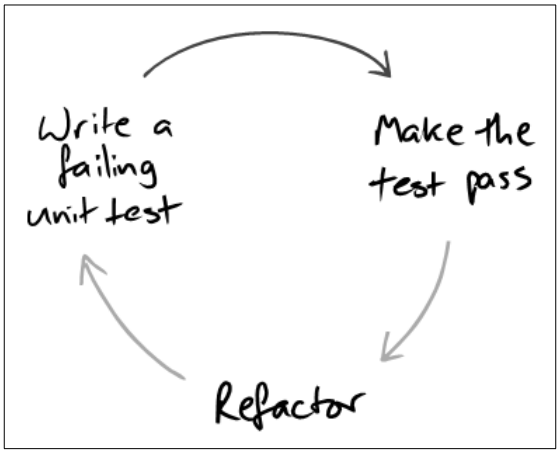

Figure 4.9: The fundamental TDD cycle [FP09]

- Write the simplest code possible to make the test pass;
- Run all tests and see them all succeed;
- Refactor the code to improve its quality.

According to Astels [Ast03], in addition to these steps, to apply TDD developers should follow principles such as: (i) maintaining an exhaustive suite of programmer tests; (ii) only deploying code into the production environment if it has tests associated. Differently from unit tests, which are written to assess a method or class, programmers write tests to define what must be developed. Programmer tests are similar to an executable specification since these tests help developers understand why a particular function is needed, to demonstrate how a function is called, or what the expected results are [Jef11].

In the CHOReOS dynamic development process, we offer guidance for using TDD in the complex scenario of service choreography development. Indeed, as opposed to the detailed information provided in both D4.2.1 [CHO12c] and D4.2.2 [CHO12d], here we focus on how TDD fits in the process. In the following, we describe the two main steps involved in choreography development using TDD.

Step 1: Creation or adaptation of atomic web services. During the implementation of choreography roles, new web services need to be created or existing ones must be adapted to implement the role requirements. Normally, a contract-driven approach is used for creating or adapting these services. In this case, the service operations have already been defined in a contract, and based on it, the service is coded.

To invoke the services, developers can use tools such as Apache Axis[3] and JAX-WS[4]. With these tools, it is possible to create stub objects (also called clients) from valid WSDL specifications, and then write tests that invoke the service through the stubs. One drawback behind this approach is the lack of flexibility: if the WSDL specification of the requested service has to change, then the client stub has to be regenerated. This can turn into a problem, since a developer might notice that a certain choreography role needs to be changed while implementing the associated web service(s). More flexibility can be achieved by using Rehearsal [BMKM12], which is a framework we developed that provides a feature for the dynamic generation of web service clients. By using Rehearsal, developers can interact with a service without creating stub objects.

As an illustrative example, consider the code depicted in Figure 4.10, which shows how to dynamically request operations defined in a web service interface (WSDL) using Rehearsal.

[3] Apache Axis: http://axis.apache.org/axis
[4] JAX-WS: http://jax-ws.java.net

```
@Test
public void testname() throws Exception {
  String wsdlUri = "http://localhost:1234/storeWS?wsdl";

  WSClient client = new WSClient(wsdlUri);
  Item response = client.request("getPrice", "milk");

  assertEquals((Double)2.50, response.getChild("price").getContentAsDouble());
}
```

Figure 4.10: Example of dynamic request to a Soap service using the WSClient

In Figure 4.10, the service under test does not exist yet. However, by using Rehearsal, one can apply a test-driven approach for implementing it. Thus, in the test, developers can specify the service endpoint (WSDL URI), the operation name (`getPrice`), and signature (receive a `String` and return a `Double` object). After writing each test, the developer must then code the service in order to make the tests pass.

Step 2: Integration of services to fulfill the choreography roles. After the services are created (or adapted) and tested properly, they must be integrated to fulfill the choreography roles. When a set of services is needed for the composition, third-party services may not be available at development time. To solve this integration problem, Rehearsal provides a service mocking feature where real services (e.g., third-party ones) can be simulated. In the example in Figure 4.11, we are mocking a real registry service, which is published in the URI specified in line 12. For playing the `Supermarket` role, the service must register its endpoint into a `Registry` service. If the Registry service is only available in the production environment, it can be mocked as presented in Figure 4.11. Thus, we can simulate the service registration for validating the role implementation in the testing environment. Thus, in lines 14-15, we instantiate a `WSMock` object; in lines 17-18, we define the response this object must provide for any parameter received. Finally, in line 19, we start the mock, and its WSDL interface is published on the URI: `http://localhost:1234/registryMock?wsdl`. After all real dependencies have been mocked, the services can be integrated to compose a choreography role. To assess the messages exchanged among the services within the executable process, Rehearsal provides a message interceptor feature. Using this feature, tests to validate this message exchange can be written before the developer performs the real integration.

```
10    @Before
11    public void publishSMRegistryMock() throws Exception{
12        String realUri =" http://a-remote-host:8084/petals/services/smregistry?wsdl";
13
14        WSMock registryMock = new WSMock("registryMock", realUri);
15        registryMock.setPort("1234");
16
17        MockResponse response = new MockResponse().whenReceive("*").replyWith("registered");
18        registryMock.returnFor("addSupermarket", response);
19        registryMock.start();
20    }
21
22 }
```

Figure 4.11: Example of web service mocked with WSMock

Figure 4.12 illustrates a test case that validates the integration of the service assessed on Figure 4.10, with the `Registry` mock service (see Figure 4.11) by analyzing the messages exchanged during their interaction. First (lines 17-18), a `Service` object is created to represent the previously created mock. Then, on lines 21-22, we define a message interceptor to intercept all messages sent to the mock service. To trigger the messages exchanged within the executable process, or in other words, the

role implementation, we invoke the process, which is exposed as a service in lines 25-26. Finally, we retrieve and validate the messages intercepted in lines 29-31.

```
15  @Test
16  public void shouldSendTheCorrectEndpointToRegistryWS() throws Exception {
17    Service registry = new Service();
18    registry.setWSDL("http://localhost:1234/registryMock?wsdl");
19
20    // Create an interceptor for the registration process
21    Interceptor registration = new Interceptor();
22    registration.intercept().to(registry);
23
24    // Invoke the Supermarket execution process
25    WSClient client = new WSClient("http://localhost:1234/supermarket1?wsdl");
26    client.request("registertSupermarket", "http://localhost:1234/storeWS?wsdl");
27
28    // Validate the messages intercepted
29    List<Item> messages = registration.getInterceptedMessages();
30    Item msgIntercepted = messages.get(0).getChild("endpoint");
31    assertEquals("http://localhost:1234/storeWS?wsdl", msgIntercepted);
32  }
```

Figure 4.12: Example of message intercepted using the Message Interceptor

Since all role contracts are already defined in the choreography model, the developer can also use the contract as an oracle and to validate his/her implementation. The idea is that the role implementation should have the same interface and behavior of the oracle. Rehearsal provides a feature for applying compliance tests which aim at verifying if a service is playing the role properly based on the interface of this oracle contract. As depicted in Figure 4.13, an oracle specification (line 35) is defined and the service implemented is assessed based on this specification. This assessment is performed by the `assertRole` assertion, which verifies if the service interface matches with the oracle interface. Then, the test cases defined on `SMRoleTest` class are applied on the service: if all tests succeed, the service is in compliance with the role. This class consists of JUnit test cases that interact with an endpoint which is retrieved from the `assertRole` parameters at runtime. In the example, the endpoint is retrieved from the `Supermarket` object. In our future work, we aim at generating this test suite automatically from the oracle object.

```
33  @Test
34  public void serviceMustBeCompliantWithTheSupermarketRole() throws Exception {
35    Role oracle = new Role("supermarket", "./roles/supermarket?wsdl");
36    Service supermarket = new Service();
37    supermarket.setWSDL("http://localhost:1234/supermarket1?wsdl");
38
39    assertRole (oracle, supermarket, SMRoleTest.class);
40  }
```

Figure 4.13: Example of compliance tests

Compliance tests ensure that the implemented service plays the choreography role properly. Thus, these tests give confidence and courage to the developer to (i) integrate the new role into the production choreography, (ii) refactor the implemented code, and (iii) modify the software when requirements change. More information about compliance testing, integration testing, and acceptance testing can be found in Sections 4.8.1.1, 4.8.1.2, and 4.8.1.3, respectively. Finally, more information about Rehearsal can be found in deliverables D4.2.1 [CHO12c] and D4.2.2 [CHO12d].

Input

- BPMN2 Choreography Specification

- SoaML Specification

Output

- CHOReOS Services

4.4.4. Add newly created services to the service base

Once new services have been developed, they are registered to the service base so that they can be discovered later.

Input

- CHOReOS Services

Output

- Abstraction & Service Base

4.5. Discover Services

This activity aims at sustaining the service discovery concern in the context of the FI of Services. It relies on a first activity which produces a set of services that are possible candidates (with respect to the offered and required operations signature and related non-functional properties) for playing the participants as required in the choreography specification. Then, for each participant, the behavior of the candidate services in the set is analyzed to check if it is correct with respect to the projection of the choreography into the considered participant.

4.5.1. Configure Service Discovery

During the configuration process, the end-user can select the target service collections that interest him from the ones that have already been stored in the `Service Base`. To this end, the end-user can either provide the identifiers of the desired collections (in case he knows them) or ask for the projection of the identifiers of all the existing collections from which he can select one or more of them. In the second case, note that the configuration process retrieves from the database only the identifiers of all the existing collections (`Java Representation of Target Collection Identifiers` object), instead of retrieving their full descriptions and content (e.g., the services that belong to these collections). Having completed the selection of the collections, the end-user can select the mode (browsing or querying) in which he shall interact with the content of the selected collections in the next steps of the discovery process (`Java Representation of Discovery Mode` object). Moreover, the end-user has the option to bypass the configuration process. In this case, the default configuration is set by the service discovery process. This configuration involves *all the existing collections* as the target ones and the *browsing mode* as the interaction mode of the end-user with them.

The information that is accepted as input and returned as output by this activity is detailed as follows:

Input

- Abstraction & Service Base (MySQL)

Output

- Discovery Mode (Java)
- Target Collection Identifiers (Java)

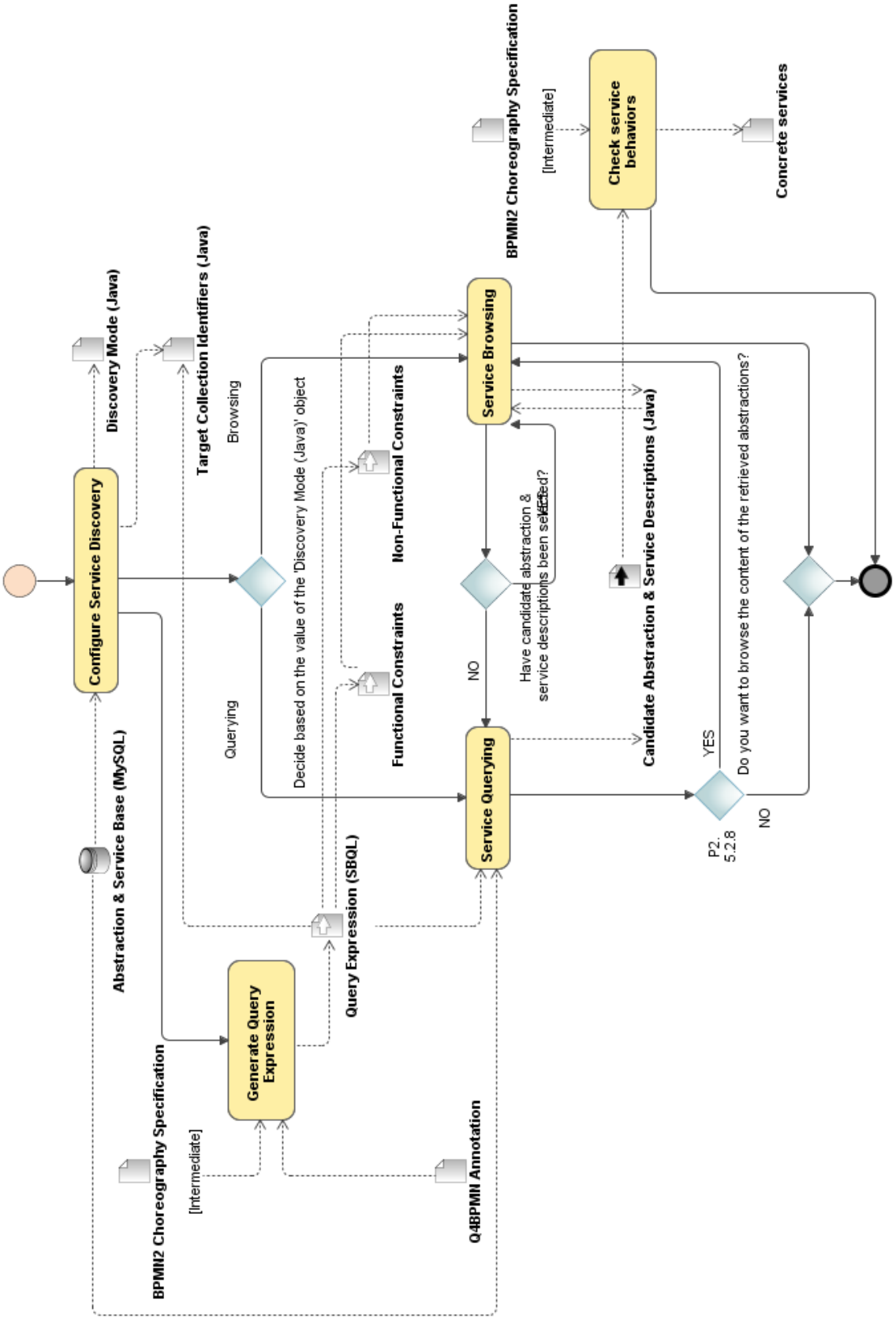

Figure 4.14: Discover Services

4.5.2. Generate Query Expression

This activity generates WSBQL expressions, which are later used for querying and browsing activities. In particular, taking as input the choreography specification, the functional part of WSBQL query expressions is generated considering the list of operations the considered role provides/consumes. A Q4BPMN model is considered for generating the non-functional part of WSBQL expressions.

Input

- BPMN2 Choreography Specification
- Q4BPMN Annotation

Output

- Query Expression (WSBQL)

4.5.3. Service Browsing

This activity visualizes the organization of existing service descriptions into abstractions hierarchies and facilitates the selection of abstractions and /or service descriptions by the end-user. Prior to visualizing the hierarchies, the activity asks by end-user to select the mode in which the browsing shall be executed. The value of the mode corresponds to the following three scenarios: (a) browsing functional abstractions and, following, browsing their nested non-functional abstractions; (b) browsing non-functional abstractions and, following, browsing their nested functional abstractions; (c) browsing functional and non-functional abstractions independently.

The browsing activity can be used either directly after the configuration of the discovery process or over the results returned from the query engine. In the first case, the browsing engine visualizes the contents of the abstractions that organize the service descriptions that belong to the collections that have been selected during the configuration of the discovery process. In the second case, the abstractions that shall be visualized have been retrieved by the `Service Querying` activity and are given as input to the browsing engine.

In both cases, the browsing consists of firstly visualizing the abstractions. The end-user traverses the abstractions by expanding and collapsing the abstraction nodes. The expansion of a functional (resp., non-functional) abstraction reveals further lower-level functional (resp., non-functional) abstractions. While traversing the hierarchy, the end-user has in mind the desired functional (resp., non-functional) constraints and compares them against the properties of the traversed abstractions. In case the end-user finds an abstraction that matches his constraints, he selects it as a candidate one and adds it to the result set. The end-user may also browse the represented service descriptions of the selected functional and/or non-functional abstractions. Finally, the activity outputs the set of the selected abstractions and/or service descriptions stored in the `Candidate Abstraction & Service Descriptions (Java)` object (Figure 4.14).

Overall, the service browsing accepts as input and returns as output the following information:

Input

- Candidate Abstraction & Service Descriptions (Java)
- Functional Constraints
- Non-Functional Constraints

Output

- Candidate Abstraction & Service Descriptions (Java)

4.5.4. Service Querying

This activity accepts as input the desired functional and/or non-functional constraints for a choreography role, written in a single query expression. The syntax of this query expression follows the specific format of a newly proposed query language, called *WSBQL*.

The input query expression is parsed and, following, executed by the querying engine that is used by this activity. The result returned by the query engine contains the descriptions of the functional and/or the non-functional abstractions that satisfy the corresponding input constraints. Note that the result may further contain the descriptions of the represented service descriptions. It depends on the type of the result specified in the query expression (i.e., return only abstractions or return abstractions along with their represented service descriptions). In any case, the result of the querying engine is stored in the `Candidate Abstraction & Service Descriptions (Java)` object (Figure 4.14).

Furthermore, the end-user has the option to browse and further filter out the results retrieved by the querying engine. To this end, the query results are given as input to the browsing engine whose functionality is described in Section 4.5.3. More details about the Web Service Base Query Language (WSBQL) are given in Appendix 6.2.

Overall, the `Service Querying` activity accepts as input and returns as output the following information:

Input

- Abstraction & Service Base (MySQL)
- Query Expression (WSBQL)

Output

- Candidate Abstraction & Service Descriptions (Java)

4.5.5. Check service behaviors

This activity checks if the selected services are suitable to play the roles in the considered choreographies.

Input

- BPMN2 Choreography Specification
- Candidate Abstraction & Service Descriptions (Java)

Output

- Concrete services

4.6. Assess Quality of Choreography Specification

4.6.1. Assess Choreography Quality via Simulation

After the specification of a choreography has been completed with the binding of the service descriptions, there's no hint about its expected performance. Various factors can contribute to worsen the response time, from slow services to unoptimised data flow. Simulation can provide a possible solution to obtain an estimate of the scalability of a choreography, as it allows implementing different scenarios to stress and find the key weaknesses of a complex and distributed software architecture. Simulation softwares, such as OPNET and OMNeT++, are based upon the queue networks theory, and require simulation models to be expressed using queues, service centers and customers entities.

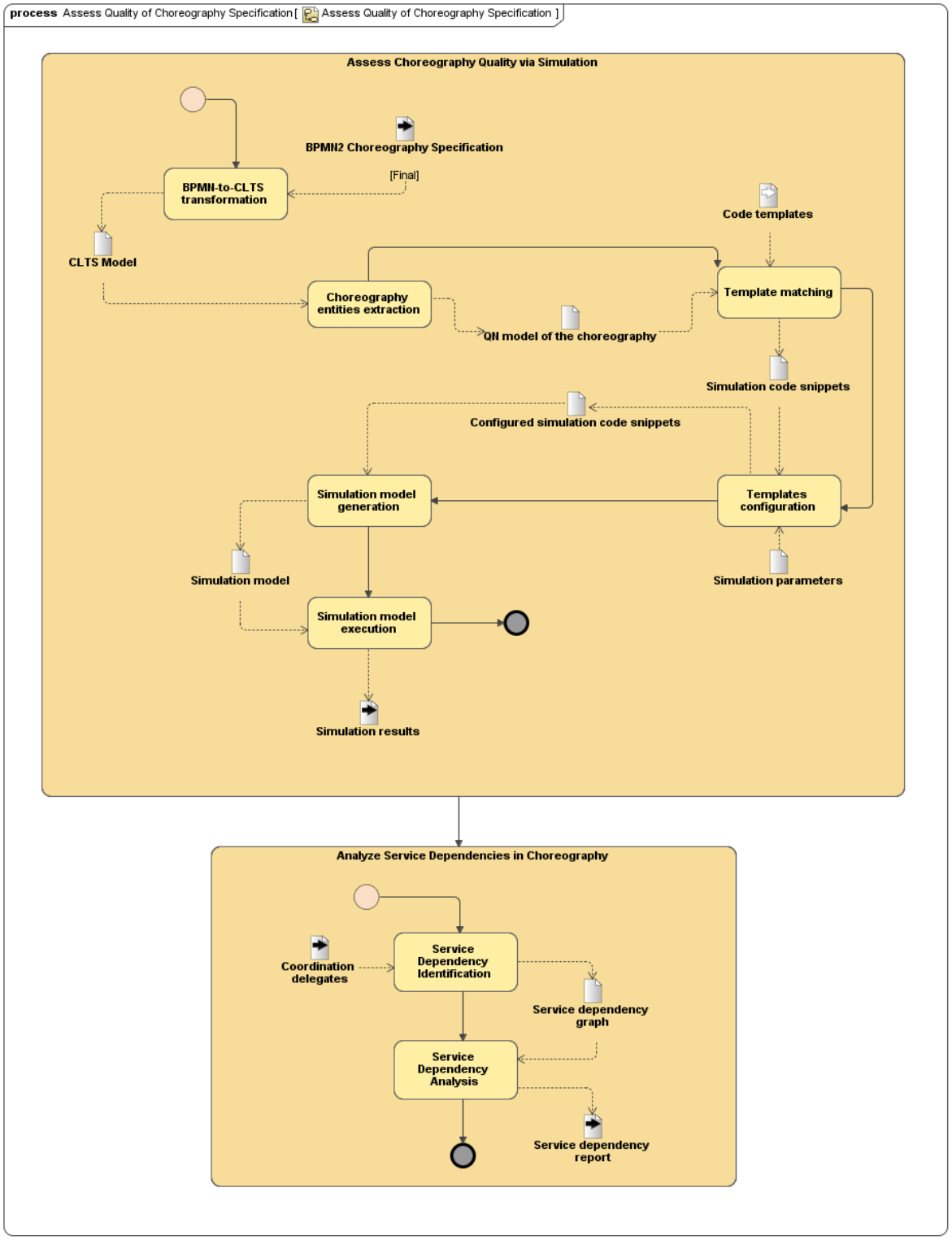

Figure 4.15: Assess Quality of Choreography Specification

A QN model is defined by the service centers the customers and network topology. Service center characteristics include the service time, the buffer space with its queuing scheduling and the number of servers. The buffer space of each service center can be finite to represent finite capacity system resources or population constraints. Customers are described by their number for closed models and by the arrival process to each service center for open models, the service demand to each service center and the types of customer.

Network topologies model how the service centers are interconnected and how the customers move between them. Different types of customers in the QN can model different behaviors of the customers, i.e., various types of external arrival process, different service demands and different types of network routing.

In order to simulate the behavior of a choreography, we have therefore to convert its specification into a QN model. By analyzing the techniques used to automatically generate simulation models and testbeds (see Appendix 6.3), we designed a process for turning choreography models, enriched with QoS information, into simulation models.

The process of generating such a choreography simulation model can be divided into multiple tasks. The choreography specification, in terms of a QoS-enriched LTS graph, is parsed as an input file (Extraction), then the choreography components are matched to a library of code templates modeling generic components and their usual behavior (Template matching). The identified code templates are then configured using QoS information and externally provided simulation settings (Templates configuration), and assembled into a coherent code base which will be used as basis for simulating the choreography using a software simulator.

4.6.2. Choreography entities extraction

This task will perform the extraction of relevant data from the input LTS and will convert the choreography model into its QN representation. The LTS model is generated out of the source BPMN2 choreography specification by means of the *BPMN2-to-CLTS* transformation described in Section 4.7.1. We will follow the approach described in [BBG08] to derive a queue network (QN) model from a LTS:

PHASE 1: Analysis of the LTS.

- Perform a LTS visit by considering all the paths in graph G to derive interaction sets formed by interaction pairs.
    - Examine all the adjacent states of a visited state and the connecting transitions. If the transition corresponds to a communication the algorithm generates a pair, called Interaction Pair (IP), denoted by (p1, p2) that signals a flow of data elements from p1 to p2, where p1 is the system component that acts as sender in the communication and p2 is the component that acts as receiver.
- For synchronous communication, where p1 starts after the completion of p2, both elements of the pair are system components, while asynchronous communication is modeled by introducing connecting elements with buffer, called Passive Connecting Element (PaC), that are elements of the corresponding interaction pair.
- Mark the interaction pairs corresponding to a system non-deterministic behavior , so to correctly distinguish, and model, concurrency and non-determinism, in the second phase.
- Put all the interaction pairs derived at the i-th visit step in Interaction Set Ii, that represents all the communications among system elements that can happen at a given time.

PHASE 2. In order to derive the QN model of the SA described by the LTS, we examine the obtained Interaction Sets to associate elements of the QN.

- Examine Interaction sets Ii, defined in Phase 1 to generate the service centers and the topology of the QN.
    - Besides the element ENV$\in$P representing the environment, distinguish the system components that are represented as internal or external elements of the QN. Hence identify a subset

A of set P, that contains those SA elements to be represented as external environment in the QN.

- Elements in A can be seen as sources which model the production of system customers or elements from which the system communicates the results to the environment. If the element ENV has been introduced, then it belongs to set A. From A we can determine whether the QN is open or closed.
- The analysis of the interactions among system components, i.e. interaction sets Ii, allows identifying their real level of concurrence. We identify components that are strongly synchronized and have a sequential behavior , even if they are modeled in the SA description as independent entities, and independent components that can be concurrently active. Each component is first considered as an autonomous server, thus modeling the maximum level of concurrency, then it can become part of a more structured QN element.

- If an interaction pair (p1, p2) corresponds to a synchronous communication, then define a corresponding QN element that is a complex server with a unique service composed of p1 followed by p2 to represent the sequence of operations.
  - If an interaction pair (p1, p 2) corresponds to an asynchronous communication, then the component that receives data is modeled as a service center with a infinite buffer that implicitly models the communication channel. Moreover from this interaction pair we can build the customer transition in the QN.
  - The interaction pair (p1, p 2) with an external element, i.e. where p1 or p2 belong to A, corresponds either to a service center with exogenous arrivals (if p1$\in$ A) or from which there are departures (if p2$\in$ A).

- To model synchronous communication among concurrent system components assign distinct service centers to the communicating components to model their independence, associate to the receiver component a service center with a zero capacity buffer and assume BAS blocking mechanism for the sender component.
  - To model a non-deterministic computation introduce multi-customer service centers that at the end are transformed in simple-customer service centers whose service times depend on the service times of the original classes.
  - To model one to many and many to one communications assign to the involved components distinct service centers, with a zero capacity buffer if the communication is synchronous.

- When each interaction set Ii has been examined perform merging operations to reach the final configuration of the service centers. Several merging operations reduce the number of service centers of the QN to model only the necessary concurrency of the system

The algorithm to derive the QN model from the choreography is based on the analysis of the LTS that represents the dynamic behavior of the software architecture under study, and follows the approach defined in [ABI01]. The algorithm will be further modified in order to consider the fact that according to D1.3 definitions the notions of role and operation abstract the notions of participant and task, respectively, of a BPMN2 choreography specification.

Input

- Choreography LTS : Choreography specification in CLTS as obtained during the Coordination Delegates synthesis
- CLTS Model

Output

- QN model of the choreography

4.6.3. Template matching

This task will analyze the QN model obtained from the Extraction task and will match it with a library of code templates representing the basic communication patterns of the choreography components and coordination delegates.

Input

- Code templates
- QN model of the choreography

Output

- Simulation code snippets

4.6.4. Templates configuration

The code obtained from the Template matching task will be adapted to the simulation scenario, defined in an external configuration file. During this task, the code snippets will also be configured using QoS data obtained by analyzing the LTS itself, plus data coming from either Q4BPNM or SoaML.

Input

- Simulation code snippets
- Simulation parameters

Output

- Configured simulation code snippets

4.6.5. Simulation model generation

The configured code snippets obtained from the Configuration task will be assembled into a coherent code base ready to be compiled and run in the simulator.

Input

- Configured simulation code snippets

Output

- Simulation model

4.6.6. Simulation model execution

The obtained model will be run in the simulator, and the results will be used to evaluate the expected QoS performance of the choreography.

Input

- Simulation model

Output

- Simulation results

4.6.7. Analyze Service Dependencies in Choreography

This subprocess involves making sense out of the existing dependencies among the services implementing the choreography. In the dynamic environment in which FI choreographies take place, services may malfunction or become unavailable. In this context, we apply the different graph centrality measures and other algorithms to identify services that are more likely to experience side-effects. Furthermore, we also analyze the choreography as a whole by detecting tangles among services and calculating stability. As pointed out in deliverable D2.1 [ARS12a], coordination delegates provide the illusion to the end-user that business services are calling each other and realizing the choreography. We thus capture service dependencies by actually considering their wrapping coordination delegates and the dependencies that exist among such delegates. The diagram depicted in Figure 4.15 provides an overview of the steps and data elements (inputs and outputs) involved in the process of service dependency analysis. As in the case of *role dependency analysis*, we plan to use both the framework Jung[5] and a generic dependency analysis library we developed in order to accomplish the analysis. More details about the methods and tools for service dependency analysis will be provided in the next deliverable D2.3 ("CHOReOS dynamic development process: methods and tools").

4.7. Synthesize Coordination Delegates for Choreography Specification

When the choreography specification is assessed as well formed, the final version of BPMN2 choreography specification is used for synthesizing the code of the Coordination Delegates that are used to enact the choreography. The synthesis process consists of different model transformations as detailed in the following subsections.

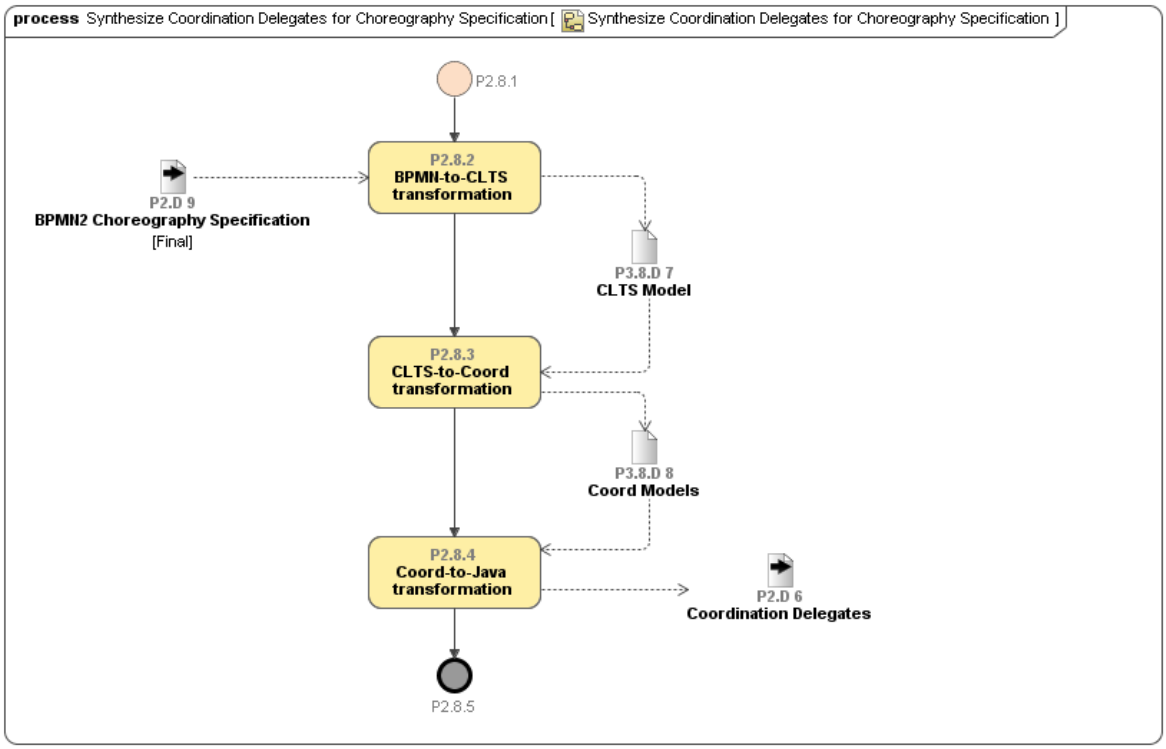

Figure 4.16: Synthesize Coordination Delegates for Choreography Specification

[5] http://jung.sourceforge.net/

4.7.1. BPMN2-to-CLTS transformation

By means of the *ATLAS Transformation Language* [JABK08] (ATL), the BPMN2 choreography specification is transformed into a Choreography Labeled Transition System (CLTS) specification by adopting the transformation rules that we have aptly defined to translate BPMN2 constructs to CLTS constructs. The CLTS model has been formally defined in deliverable D2.1 [ARS12a]. Informally, a CLTS is a Labeled Transition System (LTS) [Kel76] that, for coordination purposes, is suitably extended to model choreography behavior, e.g., by considering conditional branching and multiplicities on participant instances. In particular, the transformation consists of transformation rules each devoted to the management of specific source constructs that contribute to the generation of corresponding constructs in the target CLTS model. The transformation takes into account the main gateways found in BPMN2 Choreography Diagrams: exclusive gateways (decision, alternative paths), inclusive gateways (inclusive decision, alternative but also parallel paths), parallel gateways (creation and merging of parallel flows), and event-based gateways (choice based on events, i.e., message reception or timeout). Moreover, as anticipated in Section 4.4.1, the transformation also allows for extracting, from BPMN2 choreography specifications, information on in/out data at the choreography task level. This information is used to precisely specify in the CLTS the in/out parameters of the exchanged messages.

Input

- BPMN2 Choreography Specification

Output

- CLTS Model

4.7.2. CLTS-to-Coord transformation

An ATL model-to-model transformation is defined to automatically distribute the CLTS into a set of models, whose metamodel is denoted as `Coord`. A `Coord` model M_{CD_i}, for a delegate CD_i, specifies the information that CD_i needs to know in order to properly cooperate with the other delegates in the system. The aim of this cooperation is to prevent undesired interactions in the global collaboration of the participant services, hence enforcing choreography realizability.

Input

- CLTS Model

Output

- Coord Models

4.7.3. Coord-to-Java transformation

The `Coord` model specifies the logic that a CD has to perform independently from any target technology. To validate our approach in practical contexts, we chose Java as a possible target language of our Acceleo[6]-based model-to-code transformation.

Input

- Coord Models

Output

- Coordination Delegates : The Java code of a delegate CD_i exploits the information contained in its `Coord` model M_{CD_i}.

[6] http://www.eclipse.org/acceleo/

4.8. Perform Choreography Offline Testing

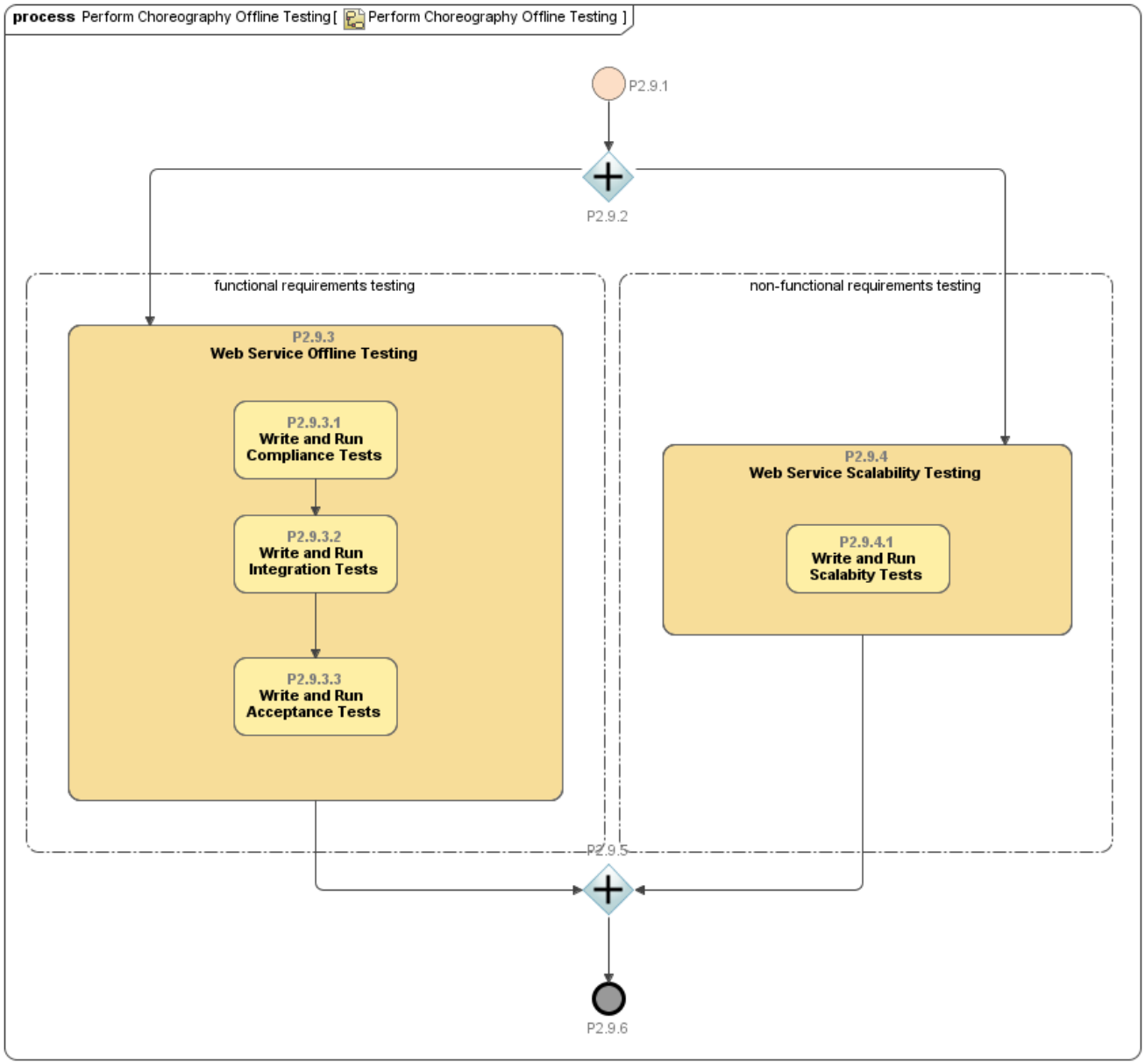

Figure 4.17: Perform Choreography Offline Testing

During a software development project, different testing strategies can be applied depending on the stage of development (Figure 4.18). In the following subsection, we present three different strategies to test the set of concrete web services that were either developed from scratch or obtained from the service base. After that, we describe how to test choreography scalability. As opposed to the detailed information provided in both D4.2.1 [CHO12c] and D4.2.2 [CHO12d], here we focus on describing how web service testing fits in the CHOReOS dynamic development process. Furthermore, we also highlight the steps that must be followed to accomplish each testing task.

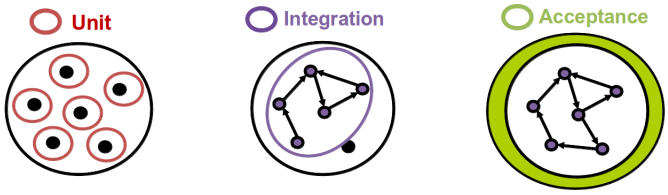

Figure 4.18: Levels of testing

4.8.1. Web Service Offline Testing

Write and Run Compliance Tests

Compliance tests rely on the concept of unit testing. Unit tests verify the behavior of a single class or method, and are not directly related to the requirements of the project, except when a key chunk of business logic is encapsulated within a specific class or method [Mes07]. In the CHOReOS context, we consider the small unit of software as a web service, thus our unit tests validate the service behavior by verifying each provided functionality. To achieve that, each functionality is verified using black-box testing. To play a role, the service can also be a composition, for instance an orchestration of atomic services [Pel03]. By definition, an orchestration is accessible as an atomic service from a user perspective and unit tests can be applied to validate the role behavior. In this case, units tests in fact represent compliance tests that validate whether the role is being played correctly or not.

Write and Run Integration Tests

Integration tests aim to solve the problems produced when unit tested components are integrated. Their goal is to verify the unit interfaces and interactions. A set of integration tests must be built to exercise these interactions and not the unit functionalities [Del97]. In this context, integration tests are also called component tests, which aim to verify components consisting of groups that collectively provide some service [Mes07]. There are some strategies to perform this integration [Pre01]:

- Top-down: the integration is performed from the main module. Initially, all components that this module depends on are mocked or stubbed, then, these dependencies are replaced incrementally by its real implementation until the system is totally integrated;
- Bottom-up: the integration starts in the atomic modules (i.e., components at the lowest levels in the program structure). These components are grouped in clusters. Once all components of a cluster are integrated, the entire cluster is integrated to other clusters.
- Big bang: all components are combined at once and tested as a whole.

In the CHOReOS context, we propose an approach for applying integration testing based on message exchange validation. Thus, we propose two levels of integration:

- Role: during the composition of web service(s) into a role, the messages exchanged inside the local orchestration (that defines the role) are validated;
- Choreography: during the composition of roles into a choreography, the messages exchanged by the roles are validated.

We propose an approach for supporting both levels of integration. After a web service is integrated, we verify whether the service newly integrated service acts as expected. This step is achieved by checking the messages sent by that component. For each message sent, its name, destination, and content are compared with the expected values.

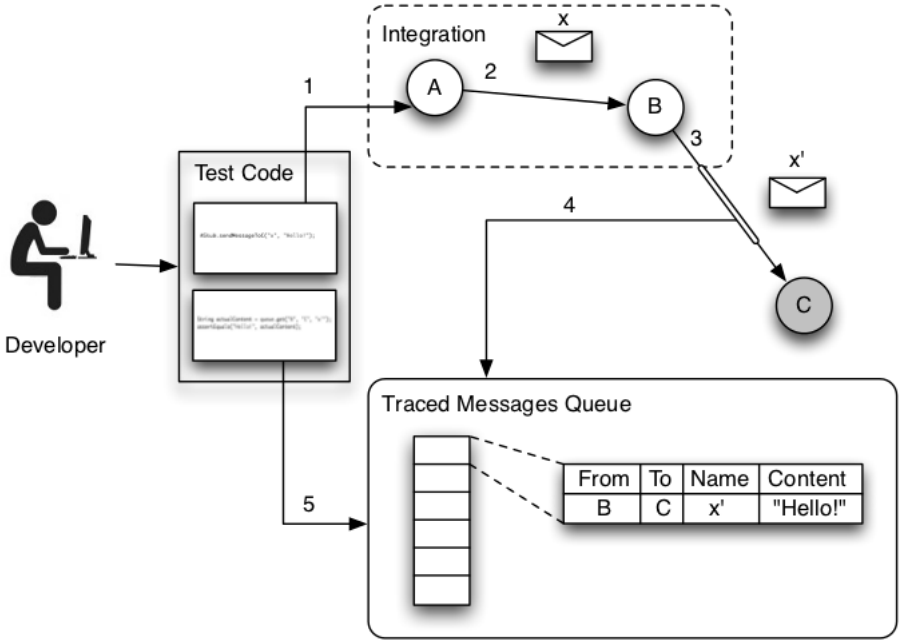

Figure 4.19: Integration test flow example

Figure 4.19 shows an example of this approach. As depicted in this figure, the developer is integrating the A and B services. At development-time, the developer specifies in the test code what services must be invoked as well as what messages must be intercepted and validated. In Figure 4.20 we present a draft version of a test code for validating the integration of the services A and B. In this example, we want to validate the message sent from service B to service C.

```
serviceA.invoke("x", "Hello!");

String actualContent = queue.get("B", "C", "x'");
assertEquals("Hello!", actualContent);
```

Figure 4.20: Test code

During the execution of this test code, in the first step of Figure 4.20, after the services deployment, the framework invokes the service A. Then, service A sends a message to service B (step 2). Our framework collects the output message from B and stores it in a queue, this is performed in the third and fourth steps. When the execution is over, the collected data is validated against the expected results (step 5).

Write and Run Acceptance Tests

Acceptance tests verify the behavior of the entire system or a complete functionality. They typically correspond to the execution of scenarios present in the use cases, features, or user stories specified by the customer. They do not depend on the implementation. Normally, they are slower than the other test strategies because they exercise all layers of the system, accessing the real components (mock objects are not used) [Mes07].

Differently from other testing strategies, acceptance tests verify the behavior of the entire system or a complete functionality. From the perspective of an end-user, the choreography is available as an atomic service. Thus, the acceptance test validates the choreography as a single service, testing a complete functionality. In this context, this type of test is similar to unit tests using the black-box model and there is no need to know how the system is implemented internally.

4.8.2. Web Service Scalability Testing

An application is scalable if it achieves the same performance by increasing the architecture capability with the same proportion of the problem size increase. In the following, we describe how the Rehearsal framework helps in the creation of scalability tests for choreographies.

Write and Run Scalability Tests

Rehearsal provides the Scalability Explorer component that assists the developer to verify the choreography scalability for future improvements. The developer must apply the following steps to assess a system scalability:

1) Choose the variables that define the problem complexity (e.g., number of requests per second), the performance (e.g., average response time), and the software architecture (e.g., number of nodes of a role);
2) Define the functions of the complexity size of the problem, performance metric, and the architecture capability;
3) Choose initial values for these variables;
4) Execute the application with these initial values for obtaining the initial value of the performance metric;
5) Execute multiple times with the same process and collect the performance metric for each execution;
6) Analyze the performance metric.

In steps 1 to 3, the developer writes the test supported by the Scalability Explorer. In order to automatize steps 4 and 5, a *ScalabilityTestItem* must be implemented, so that a *ScalabilityTest* can execute its *test(Number...)* method multiple times, increasing the parameter values according to a *ScalabilityFunction*. The framework also includes facilities such as *LoadGenerator* and *AggregationFunction* to support the test writing. For the 6th step, a chart is generated, so that the developer can analyze the test results.

The Scalability Explorer already contains a *ScalabilityTester* abstract class which implements *ScalabilityTestItem* and encapsulates a skeleton to a simple choreography test. It assumes the problem complexity is the number of requests per minute, the relevant architecture dimension is the quantity of instances of some resource and the performance is measured as the response time, in milliseconds. Figure 4.21 presents a piece of code from a scalability test that extends *ScalabilityTester* and Figure 4.22 shows how the test can be executed.

As can be seen in Figure 4.22, the test will start with 300 requests per minute and one resource. It also shows that the test consists of five batteries of ten executions each and the test will be interrupted if a response time greater than 300 milliseconds is measured. In default implementation, the tester linearly increases the quantity of resources and requests per minute. Besides, the executions are made in a uniform distribution over time, as well as the value obtained in each battery is the mean of the execution results. But other types of *ScalabilityFunction*, *LoadGenerator* and *AggregationFunction* can be set. The overridden methods in Figure 4.21 are executed in specific points during the test execution. *setUp* is called at the beginning of the test and *resourceScaling* before each battery. The *test* method is the one which response time will be measured and *beforeTest* is always called before it. Thus, in this example, all the portals are removed from the registry and, before each execution battery, one portal is registered. For each execution a portal is retrieved from the registry and a purchase request is sent to it, but only the purchase operation response time is measured. A *tearDown* method is also available and is executed at the end of the test. The *run()* method can be called many times (e.g. with different

```
private void includePortal(int index) throws WSDLException, XmlException,
        IOException, FrameworkException, InvalidOperationNameException {
    WSClient client = new WSClient(portals[index]);
    client.request("register");
}


@Override
public void resourceScaling(int resourceQuantity) throws WSDLException,
        XmlException, IOException, FrameworkException,
        InvalidOperationNameException {
    includePortal(resourceQuantity - 1);
}

@Override
public void setUp() throws Exception {

    removeAllPortals();
    registry = new WSClient(REGISTRY_WSDL);
}

@Override
public void beforeTest() throws Exception {
    Item result = registry.request("getServiceRoundRobin", "portal");
    String wsdl = result.getContent("return");
    client = new WSClient(wsdl + ENDPOINT);
}

@Override
public void test() throws Exception {
    client.request("purchase", getShopList());
}
```

Figure 4.21: Example of scalability tests

```
public static void main(String[] args) {
    SupermarketScalabilityTester tester = new SupermarketScalabilityTester();
    tester.setInitialRequestsPerMinute(300);
    tester.setInititalResoucesQuantity(1);
    tester.setMeasurementLimit(300.0);
    tester.setNumberOfExecutionsPerTest(10);
    tester.setNumberOfTestsToRun(5);
    tester.run("purchase operation");
    tester.showChart("supermarket choreography");
}
```

Figure 4.22: Example of scalability tests

configurations) and its argument is used to identify the results of each specific execution in a chart presented when *showChart* is called.

5 Conclusions and Future Work

Capitalizing on the abstract model of the CHOReOS development process presented in deliverable D2.1 [ARS12a], this deliverable describes the specific set of concrete process activities, and their flow, being implemented in CHOReOS. To this end, each activity in the process model has been refined by precisely identifying techniques, methods and tools (to be integrated by the IDRE) for choreography design, analysis, synthesis, deployment and enactment. Where needed, the process has also been extended by leveraging the results of the technical work from other WPs.

This deliverable takes into account the Commission recommendations for WP2 covering the project months up to M18 and has been deeply refined according to the request for revision at M24 by clarifying the aspects concerning ULS, data mapping and SoaML, as well as QoS and Q4BPMN.

The next step for WP2, at M36, will be to finalize the implementation of the defined approaches and prototype tools, and fully integrate them into a coherent framework supported by the IDRE.

6 Appendix

6.1. Abstraction & Service Representation and Storage Model

We present abstraction and service representation in two layers. First we start with the service-oriented component model that realizes the basic concepts of the CHOReOS architectural style. The service-oriented component model is the representation of our data in main memory. All data that are about to be stored, as well as the data that are retrieved as an answer from a query, are represented in the service-oriented component model. Then, we discuss the database schema, used for the persistent storage of the data in the `Service Base`.

The Service-Oriented Component Model. The service-oriented component model in Figure 6.1 is largely divided in two parts. The first part includes the representation of architectural style concepts that concern services, including service types, service interfaces, service instances, messages and their structure.

The center of the service model (Figure 6.1) is the notion of `ServiceInterface`, which comprises a collection of `ServiceOperation` objects. The `ServiceOperation` concept comprises information regarding input and output messages. The `Message` concept carries a set of types parameters/fields (coded as message types and components in the Web service model). The `ServiceInterface` concept is further associated with `ServiceInstance` objects that represent information concerning the actual service endpoints (URI). The `ServiceInstance` concept is characterized by a `QualityProperties` object (e.g., availability, reputation, price, ...), each with an abstract domain (low, high, ...). The concrete characterization of the `ServiceInstance` concept for its quality properties is performed via the a set of `NonFunctionalDescription` objects, which contain scores for specific quality properties, given by a third-party assessing source. The `ServiceType` concept is practically a high-level representative of the `ServiceInterface` concept – in theory a service type can implement several interfaces, however practically there is a 1:1 relationship). The service model includes also information on the provenance of the collected information (i.e., where did the `Service Base` receive the incoming information), via the `ServiceSource` concept. The `ServiceCollection` concept groups service information that comes from a specific provenance, identified via a set of `ServiceSource` objects.

The second part of the service-oriented component model includes the representation of abstractions. Abstractions form hierarchies that include `FunctionalAbstraction` objects, `NonFunctionalAbstraction` objects, or both.

The core ideas of the abstraction model (Figure 6.2) have been elaborated in deep, in previous deliverables. We quickly summarize that the `FunctionalAbstraction` concept contains information concerning a set of structurally similar interfaces. More specifically, the `FunctionalAbstraction` concept is characterized by (a) the set of the interfaces that are represented by the functional abstraction, and, (b) an abstract interface, which stands out as a representative for the represented interfaces. The represented interfaces have been described in the service model of the previous paragraph; the representative interface is similarly characterized by ServiceOperation, Message and Parameter objects. The interesting part is that we record the mappings among represented and representative interface (also keeping trace of their similarity in the form of a distance attribute): this happens for

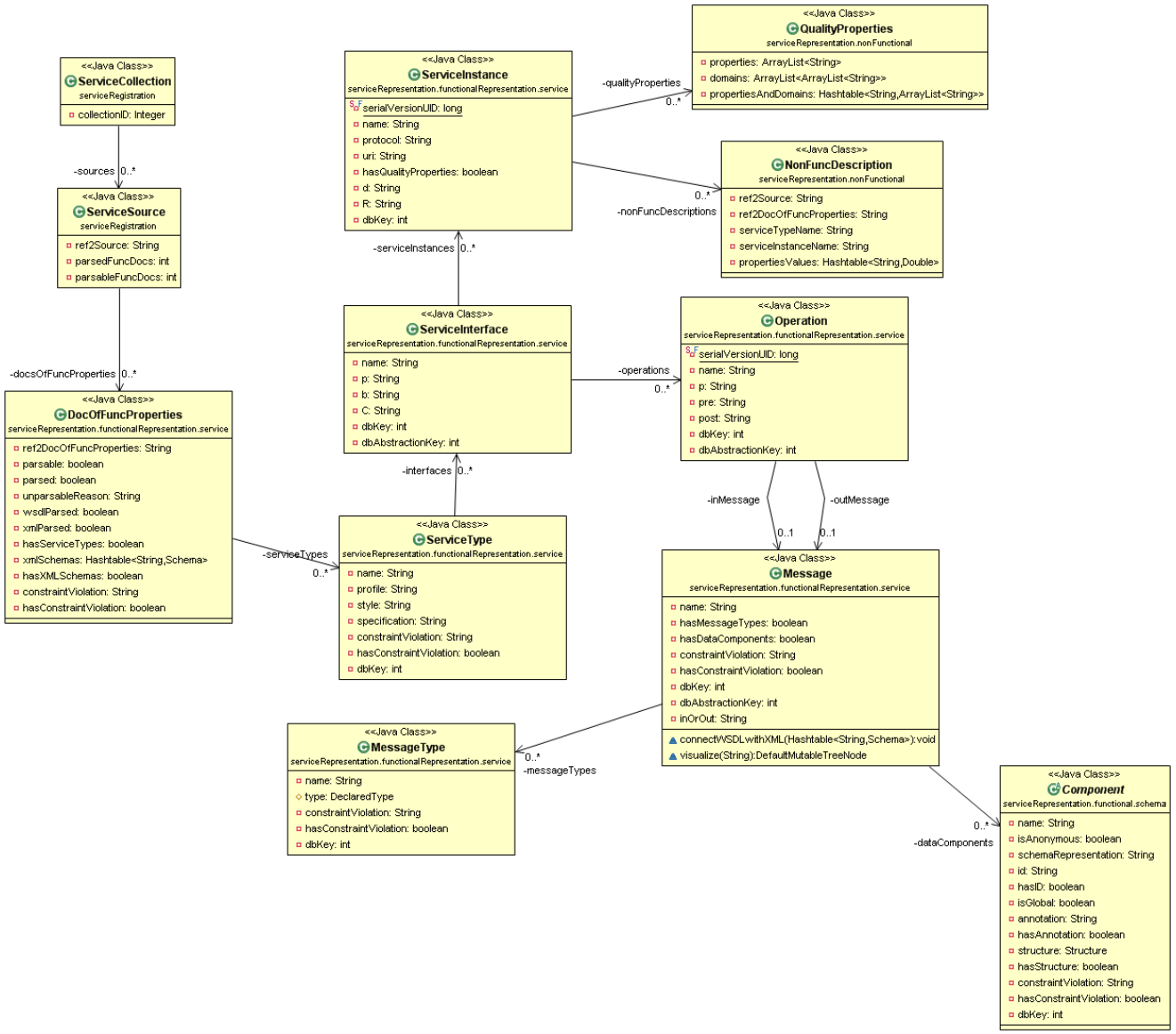

Figure 6.1: The classes of the service-oriented component model that represent services in main memory.

interfaces, operations and messages. The `NonFunctionalAbstraction` concept contains information concerning a set of service instances characterized with similar values for their quality properties. Both kinds of abstractions form tree-like hierarchies with abstractions at higher levels being composed as the "union" of a set of abstractions at the lower level.

The Database Model. We briefly describe the database model concerning the representation of services, service types, service interfaces and their components, service instances, abstractions and quality characterizations inside the `Service Base` that is used for persistent storage. We employ a relational DBMS, namely MySQL, for the storage and ultimate querying of the `Service Base`. We stress that we employ typical and standard features of practically all mainstream relational DBMSs both in terms of representation and querying; therefore the usage of other DBMSs as persistent storage and querying engines is straightforward.

Roughly, the database model comprises relations (a.k.a tables) that correspond directly to the concepts of the service-oriented component model. Note that in order to distinguish between the concepts of the service-oriented component model and the relations of the database model, we employ a different

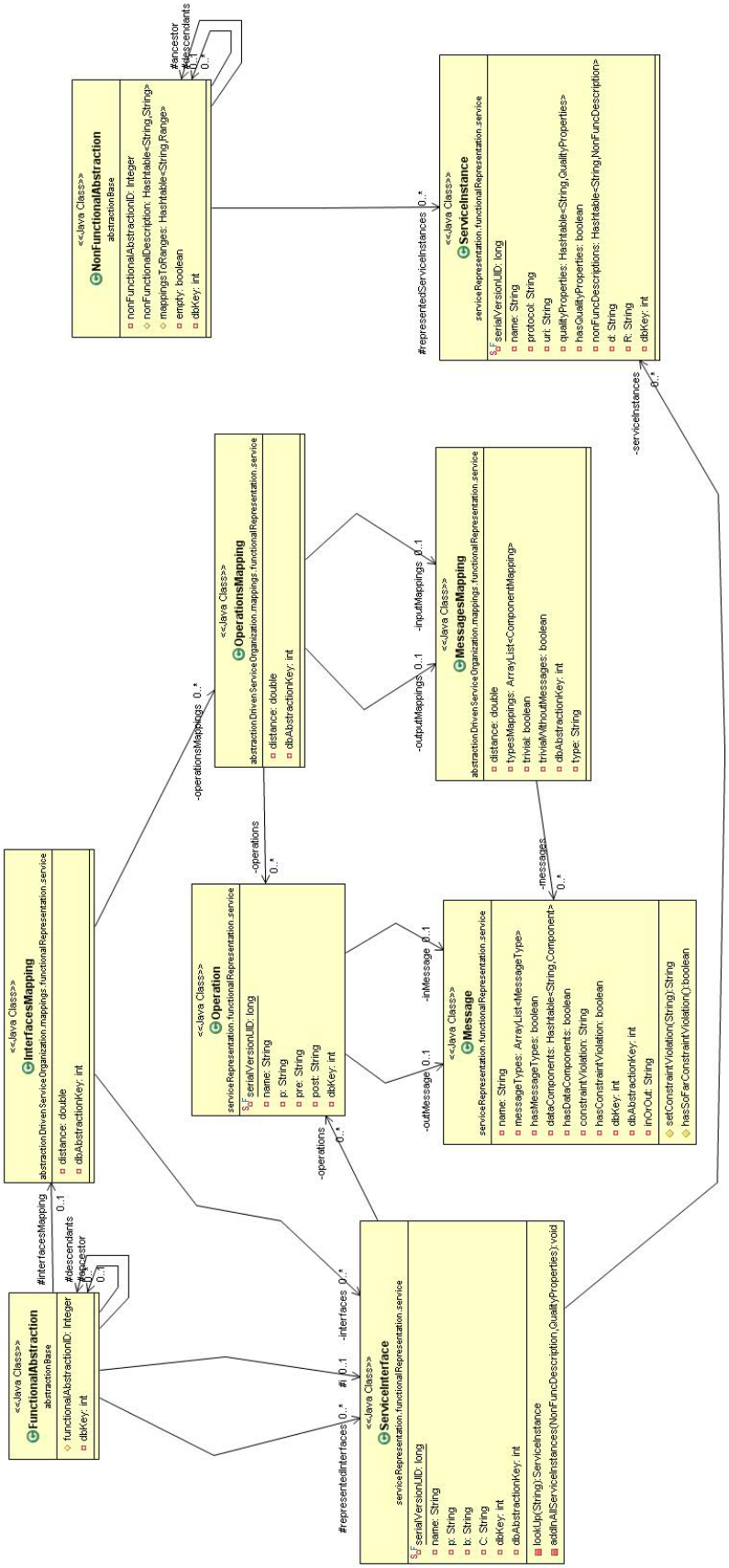

Figure 6.2: The classes of the service-oriented component model that represent service abstractions in main memory.

Figure 6.3: The basic relations (also known as tables) of the `Service Base` concerning service representation and their relationship to abstractions.

naming convention for the relations; relations are named with lowercase letters.

Figure 6.3 comprises the part of the database model that concerns services, their internal structure and their relationship to abstractions. The notion of aggregation in the object oriented paradigm is modeled via foreign key relationships in the relational paradigm. Observe the upper right part of the figure: there is a 1:M relationship between relations `servicetypes` and `serviceinterfaces`, representing the fact that a `ServiceType` object encompasses one or more `ServiceInterface` objects. This is modeled via a foreign key from relation `serviceinterfaces` to the primary key of relation `servicetypes`: specifically, attribute `SI_ST_ID` in relation `serviceinterfaces` is a foreign key (and thus, a subset of) attribute `ST_ID` in relation `servicetypes`. The same pattern appears consistently throughout the database schema.

The service representation part of the database schema is depicted on the top of Figure 6.3. Starting from right to left, service types include service interfaces that include operations that include messages that include parameters (relation message types). Service instances (bottom of the figure) are also linked to their respective service types for a foreign key.

A functional abstraction includes several interfaces that it represents via relation `representedinterfaces`. A non-functional abstraction is linked to service instances that it abstracts via relation `representedinstances`. A representative interface of a functional abstractions is captured via relation `representativeinterfaces`. A representative interface's decomposition in its parts is captured via the line of relations `representativeoperations` and `representativemessages`. The mappings between the parts of the abstraction's represented interfaces and the respective parts of the components of the representative interface of the functional abstraction are depicted in the middle "line" of the Figure 6.3.

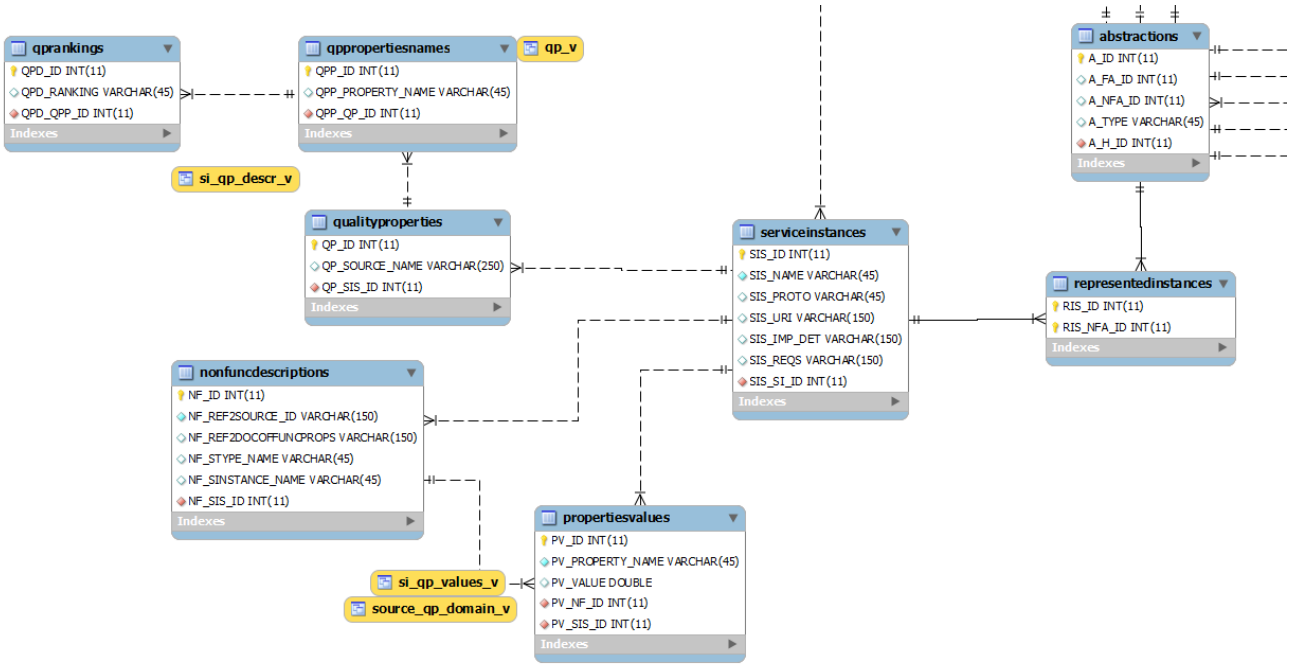

Figure 6.4: The basic relations (a.k.a. tables) of the `Service Base` concerning instances and their relationship to quality properties.

The part of the database schema that concerns service instances and their relationship to quality properties is given in Figure 6.4. Service instances are characterized by `nonfunctionaldescriptions` that state that a recommender source (`NF_Ref2Source_ID`) has characterized a service instance (`NF_SIS_ID`) for a property (`PV_Property_Name` in relation `propertiesvalues`) has a certain concrete value (`PV_Value`). At the same time, a source (`QP_Source_Name` in relation `qualityproperties`) assesses a service instance (`QP_SIS_ID`

in relation `qualityproperties`) for a set of properties (`QPP_Property_Name` in relation `qpropertynames`) with a domain of characterizations (like `low`, `high`, etc) (`QPD_Ranking` in relation `qprankings`).

To alleviate the burden of querying this complex composition of relations we have introduced a set of views. View `qp_v` lists all quality properties with their names. View `si_qp_descr_v` returns all combinations of quality properties, service instances, sources of recommendations, and domains of possible (high-level descriptions of) rankings. Domains are also captured by view `source_qp_domain_v` that returns all combinations of quality properties, sources of recommendations, and actual domains of concrete scores in terms of min and max values assigned to instances. Most importantly, view `si_qp_values_v` as returns all combinations of quality properties, service instances, sources of recommendations, and actual values of the recommendations.

Finally, in Figure 6.5 we gives the part of the database schema that concerns the abstractions, their properties and inter-relationships. As one can see in Figure 6.5 service collections include hierarchies that, in turn, include abstractions. We have already seen how representative interfaces are related to functional abstractions. To handle how quality properties are related to non-functional abstractions we introduce view `nfaqp_v` that abstracts relations `nfabstractionsranges` and `nfabstractionsnonfunctionaldescriptions`. The view returns, for each non-functional abstraction, and each of its quality properties, the lowest and highest value within which it ranges. The abstractions form hierarchies and relation `hierEdges` captures the mother-child ancestor-descendant relationship.

A particular point of interest is the representation of the inheritance relationship between (a) abstractions and (b) functional abstractions and non-functional abstractions in the database schema (Figure 6.2). The problem is of particular difficulty in databases since we want the following characteristics to be taken into consideration in the solution:

1) The solution must apply to a traditional DBMS without inherent support for inheritance, mainly for portability and interoperability issues

2) The querying must be as simple as possible in order to avoid complicate sequences of queries that, for example, probe a table for a particular mother class and then probe a second table for the characteristics of the subclasses

3) Foreign keys must be applicable – unfortunately, in traditional DBMSs foreign keys can be targeted towards a *single* relation. Therefore, in all cases of abstract coupling where a container is defined as a container for objects of a mother class and subsequently be populated by objects of the child classes, the respective foreign key must be targeted to a single table.

There are several possibilities for representing an inheritance hierarchy in relational terms:

1) Ignore the mother class and produce one table per subclass. Unacceptable as when want foreign keys to be directed to the mother class.

2) A lookup table for the mother class and auxiliary tables for the subclasses. The lookup table can have only the identifier as primary key along with a characterization of which subclass this tuple belongs to, or, it can also have the common fields too. The problem with this solution is the need for costly joins to compose the original information of a subclass object

3) A single table with all the characteristics of the subclasses. This solution is quick in response time and solves all the problems of querying easiness and lack of joins. On the down side, the relation becomes too empty (and thus space wasting) when the hierarchy is large and it is more difficult to maintain when the hierarchy is altered (the table's schema must be altered too).

In our case we had to deal with a small hierarchy of one mother class and two subclasses. Due to the small size of the class hierarchy, we made a judged decision to opt for the single table solution. An

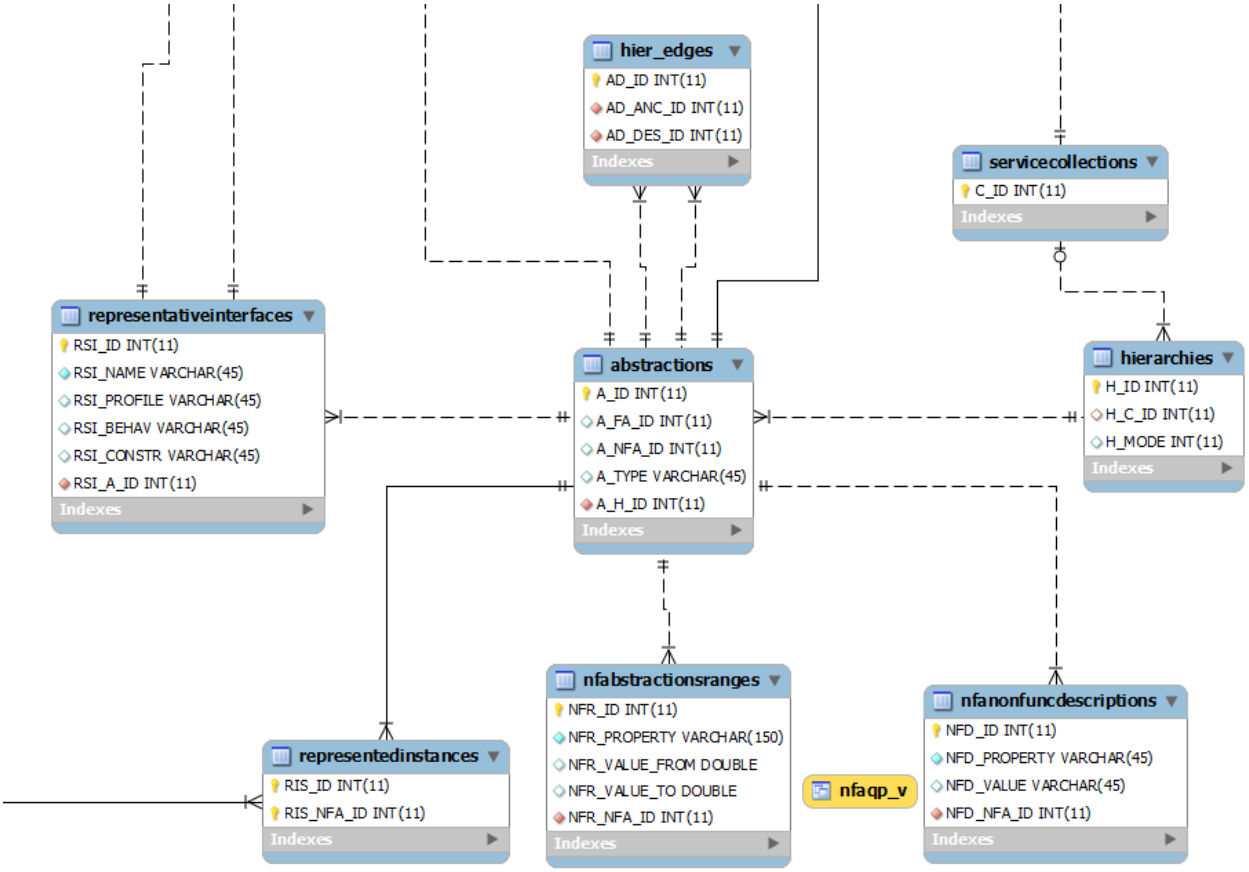

Figure 6.5: The basic relations (a.k.a. tables) of the `Service Base` concerning abstractions and their hierarchies.

extra advantage of our preference to the third solution is that the structure of the hierarchy is unlikely to change. Therefore, we produced a single relation `Abstractions` with all the characteristics of all three kinds of abstractions and link information concerning both the functional and non-functional abstractions (i.e., representative interfaces and quality properties) via foreign keys to this relation.

6.2. The Service Base Query Language

Hereafter, we present the Web Service Base Query Language (WSBQL). We begin with the main concepts of the language. Then, we give the SQBL syntax and the semantics of the WSBQL language.

6.2.1. Basic Concepts - Generalized Trees for Querying Services.

The querying of the `Service Base` requires the query author to think of the database as a generalized tree. As we will describe later, a generalized tree is a graph that resembles a tree a lot; however there are nodes that break the fundamental property that a tree's non-root node has exactly one father, and consequently, we use the –hopefully intuitive– term generalized trees.

To query the abstraction's part of the `Service Base` schema (i.e., to retrieve abstractions), we think of this part of the schema as a tree. The model that the query author has to keep in mind is depicted in Figure 6.6. We call this tree the *Generalized Tree of the Service Base at the Schema Level* and we textually detail the parts right away.

- A service collection contains several hierarchies of abstractions.

- A hierarchy contains several abstractions.
- An abstraction can be simply dealt as a generic abstraction, or, in more specificity, either as a functional abstraction or as a non-functional abstraction.
- Functional abstractions have representative interfaces.
  - These representative interfaces have operations.
  - Each operation has input and output messages.
  - Each message has a set of parameters.
- Non-functional abstractions are characterized by a set of quality properties.
  - Each such quality property, in the context of a non-functional abstraction, has a range of values and a user-friendly name.

A subtle point concerns the distinction between abstractions and their subclasses, functional and non-functional abstractions. The usage of abstractions in the tree is merely to allow the user to navigate over the tree, without delving in particular in the specifics of the abstractions.

Except for the *Generalized Tree of the Service Base at the Schema Level*, the `Service Base` involves service instances too. Instances are represented via the *Generalized Tree of the Service Base at the Instance Level*. As the abstraction part of the `Service Base` obeys the schema of the *Generalized Tree at the Schema Level*, its contents can form the *Generalized Tree of the Service Base at the Instance Level*.

6.2.2. WSBQL Syntax

Table 6.1 gives an example of an WSBQL query that queries the `Service Base` for functional abstractions that are characterized by the following characteristics:

- They have a representative interface whose name includes 'SMSSend'
- They have two operations with the following characteristics
  - The first operation has
    - a name which include the text 'sendMe'
    - an input message with two parameters: (a) a parameter with type 'String' and name 'Sender' and (b) a parameter with name 'IP'
  - The second operation has an output message with a parameter with name 'text'

The query returns to the user all the information concerning the functional abstractions that fulfill the aforementioned criteria.

The syntax of the language largely follows XQuery for several reasons. First, the generalized tree structure of the service base resembles the XML hierarchical structure of the text. Second, the learning curve is much milder if we follow an existing language. Third, there is always the potential of porting the service base to XML stores in the future, without much effort (if XML stores acquire the potential of scaling up to ultra large scale volumes of text). At the same time, the difference is mainly located in the return types that are intentionally restricted to a fixed set, in order to allow the complete representation of the returned data to full descriptions of abstractions (see next for a detailed discussion).

The general syntax of a WSBQL query is given in Table 6.2. The reserved words that distinguish the different parts of an SQBL query are presented with underlined format.

We discuss each of these parts separately in the sequel, after we have formally defined their constituent elements.

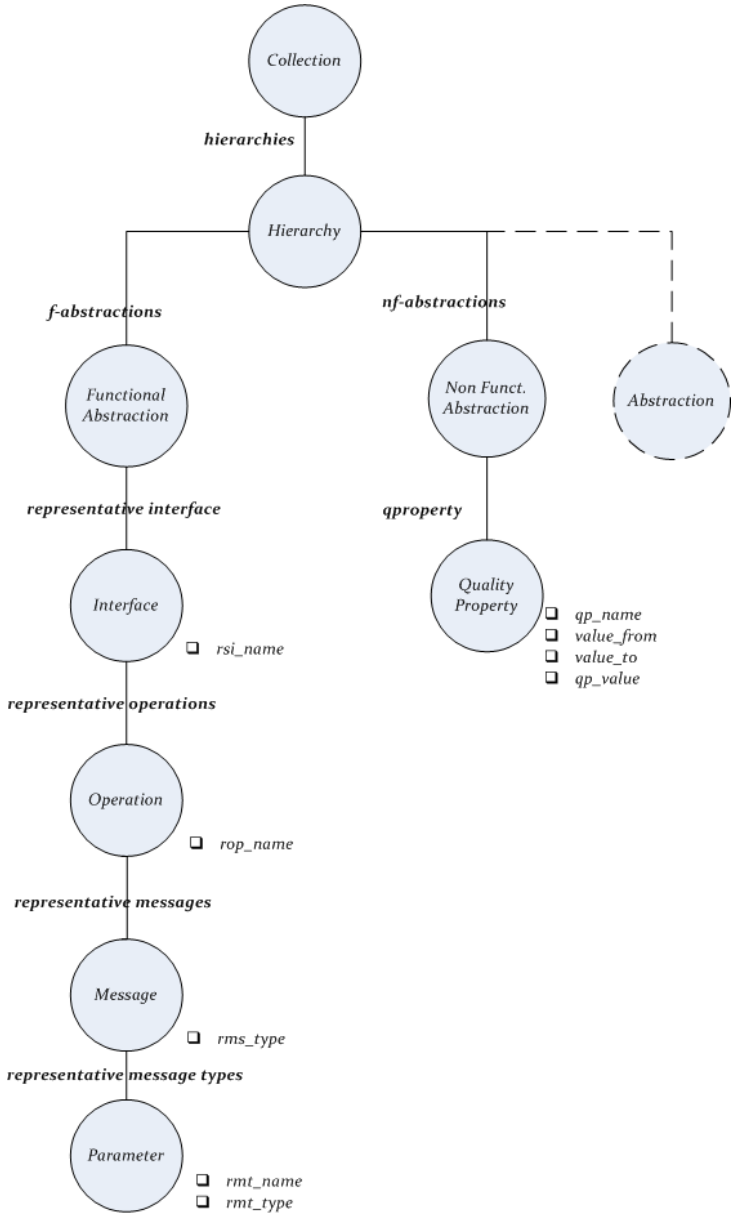

Figure 6.6: The tree that abstracts the structure of functional and non-functional abstractions at the schema level.

Variables. A variable is an alphanumeric string that begins with a dollar sign '$'.

Path Expressions. A path expression is of the form

$$\texttt{variable}_0\texttt{/edge}_1\texttt{/.../edge}_n$$

A path expression is well-defined if the sequence of edge names creates a linear path in the generalized tree of Figure 6.6.

Variable definition. A variable definition is of the form

`for variable in pathExpression` i.e., $\texttt{for variable in variable}_0\texttt{/edge}_1\texttt{/.../edge}_n$

It is worth noting here that variables have a type, defined by the last edge of their path expression. In our abstract notation, $\texttt{variable}_0$ has type $\texttt{edge}_n$.

Filters. A filter is an expression of the form

Table 6.1: A WSBQL query example.

```
let       $db = db(''localhost/mySB'')
for       $c in $db/servicecollections
for       $fa in $c/hierarchies/abstractions
for       $if in $fa/representativeinterfaces
for       $o1 in $if/representativeoperations
for       $o2 in $if/representativeoperations
for       $p1 in $o1/representativemessages/representativemessagetypes
for       $p2 in $o1/representativemessages/representativemessagetypes
for       $p3 in $o2/representativemessages/representativemessagetypes
where
          $if/rsi_name like %SMSSend% and
          $op1/rop_name like %sendMe% and
          $op2/rop_name = 'exactSearch' and
          $p1/rmt_name = 'Sender' and
          $p1/rmt_type = 'String' and
          $p2/rmt_name = 'IP' and
          $p3/rmt_name = 'text'
return
          Abstractions.representativeInfo
```

Table 6.2: The general syntax of a WSBQL query.

```
let       databaseSpecifier
          variableDefinitionArea
[where    filterList]
return    returnExpression
```

`variable/field` θ `value`, with $\theta \in \{ =, LIKE, <, >, <=, >=, <> \}$

Database specifier. A database specifier characterizes which database we will query. The syntax of a database specifier is

`databaseVariable = db(database)`

where *`databaseVariable`* is going to be used in subsequent variable definitions and *`database`* is a string with the name of the database that we query (as understood by the underlying DBMS).

Variable Definition Area. A list of variable definitions. Variable definitions are separated by newlines in the variable definition area.

Filter List. A list of filters. More than one filters are connected by `and` connectors in the filter list. The `where` clause is optional and consequently, the list may be empty; in this case we assume that a filter with semantics *true* is implicitly implied.

Return type. The return type dictates what the ultimate result will be in terms of main-memory representation and it is an expression of the form

`classType.returnType`

where:

`classtype` $\in$ {`Abstractions`, `F-Abstractions`, `NF-Abstractions`}

`returnType` $\in$ {`singleObject`, `RepresentativeInfo`}

6.2.3. WSBQL Semantics (and mapping to SQL)

The semantics of the query language are largely based on the correct definitions of the individual parts of a WSBQL query. We will employ the term *well defined* to refer to individual parts whose declaration by the user makes sense.

Well defined variables. Every variable definition in the variable definition area involves two variables, specifically, (a) the *declared variable* at the beginning of the definition and (b) an *auxiliary variable* at the beginning of the involved path expression.

Then, every variable has

- An abbreviated path, which is the one appearing in the variable definition area
- A full path that is produced if we replace the auxiliary variable in the path expression with its own full path

For this recursive definition to work, we need to define the full path for the auxiliary variable(s) that appear in the database specifier, which is the empty set.

A variable is *well defined* if its full path is a continuous path in the Generalized Tree of the service base at the Schema Level, starting at collections and ending at the type of the variable, as a simple line.

Well-defined filters. A filter is a triple of the form

$$\texttt{variable/field}\ \theta\ \texttt{value}, \text{ with } \theta \in \{ =, LIKE, <, >, <=, >=, <> \}$$

Assuming the variable to be of type T, filter is well defined if the field appearing in the filter's expression belongs to type T.

In the sequel, we assume that all variables and filters are well defined; if not, the query returns an error code and an empty result set.

Query semantics. The semantics of a query, i.e., the list of returned objects that correspond to the application of the query expression over an arbitrary service base are defined via the sequence of the following four steps.

1) The query takes as input the Generalized Tree of the Service Base at the Instance level.
2) We intend to annotate each node with a variable as prescribed in the `let` clause of the query. For every abstraction appearing in the `let` clause of the query, we produce all possible clones of its subtree with all the applicable combinations of variable assignments.
3) We pass every such clone via the set of filters prescribed in the `where` clause of the query. The semantics of the filter list are conjunctive; in other words, for a subtree to become part of the result, all the filters of the filter list must evaluate to true (see next). We call these subtrees, *survivor* subtrees.
4) For every survivor subtree, we compute its return graph of objects as prescribed from the `return` clause of the query.

Filter semantics. The semantics of a filter are as follows:

- The input to the filter is a subtree of an abstraction as previously defined.
- The filter annotates a node in the tree. The node which is annotated is the one resulting from the full path of the variable
- We replace the `variable/field` part of the filter's expression with the respective value of the node

- If the resulting expression evaluates to true then the input path is added to the result of the filter, i.e., its output; else nothing is added to the output.

A filter list is the conjunction of filters and each of them is applied to the appropriate node. A subtree survives if all its filters evaluate to true. If the filter list of a query is empty, we assume that a single filter with semantics *true* is added; thus all subtrees evaluate to true without further checking.

Query completion for the return type. The result of a query depends on the return type of the query. The return type of a query is a combination of two factors, namely (a) the *classtype* and (b) the *returntype*. The semantics of the query require that first we constrain survivors according to their *classtype* and then we compute the final result based on the *returntype*.

The *classtype* retains in the result only the survivor clones that belong to the appropriate class. Specifically:

- If the *classtype* is `Abstractions`, then all survivors are retained in the survivors set
- If the *classtype* is `F-Abstractions`, then only the survivors whose root is a functional abstraction are retained in the survivors set
- If the *classtype* is `NF-Abstractions`, then only the survivors whose root is a non-functional abstraction are retained in the survivors set

Once this step is completed, the final result of the query is computed. For every survivor, we return the following graph:

- If the return type is `singleObject` then we return only the object of the respective class in the object model (i.e., `FunctionalAbstraction` or `NonFunctionalAbstraction`) for the root of the survivor subtree
- If the return type is `RepresentativeInfo` and the class of the root is `FunctionalAbstraction`, then we return the graph produced by the `FunctionalAbstraction` object and its components belonging to classes `ServiceInterface`, `Operation`, `Message` and `Component` (corresponding to the *Parameter* of the Generalized Tree).
- If the return type is `RepresentativeInfo` and the class of the root is `NonFunctionalAbstraction`, then we return the object of this class as it appears in the object model.

6.3. State of the art in simulation of QoS in Web Service choreographies

Software architectures describe the structure of software systems at a high level of abstraction and have been devised as the appropriate design level to perform quantitative analysis. To this aim, several approaches have been proposed to integrate or combine performance analysis and software architecture specification. Various types of performance models and different specification languages have been considered and/or developed for the purpose. Some approaches refer to the entire software life cycle, whereas others refer to a certain software stage, usually the design specification one.

Simulation allows predicting software applications performance in different status and load conditions of the execution environment. The predicted results are used to provide feedback on the efficiency of the application. Service simulation provides a mechanism for building “light-weight” reference architectures by eliminating expensive IT components and without sacrificing the functionality required by consumer developers to build their applications hooks into the producer systems.

6.3.1. QoS evaluation by simulation

The synergy between Web service technology and simulation has been introduced in [MCS02], demonstrating that Web service processes can be simulated for the purpose of correcting/improving the design or even for making adaptive changes at runtime. This work focused on functional properties by simulating the execution of processes composed by Web Services and measuring their performance.

Simulation of Web Service process executions to measure QoS parameters has been introduced in [CSM+02]. In a later work [CMS+03], the authors focused on problems related to service based architectures specification, evaluation, and execution using Service Composition and Execution Tool (SCET). SCET allows to compose statically a WS process with WSFL and to generate a simulation model that can be processed by the JSIM simulation environment. In this work, WSFL has been enhanced to include QoS measures obtained by carrying out simulation tests.

[JDJ+10], in a work supported by S-CUBE EU project, use a discrete-event approach to simulate QoS in service-based orchestrations. Orchestrations are described as BPEL diagrams, which are then transformed into discrete-event actions. The simulation framework has been implemented by extending NS-2 network simulator.

[NM02] proposes a model-theoretic semantics as well as distributed operational semantics that can be used for the simulation, the validation, the verification, the automated composition and the enactment of DAMLS-described service based architectures. To provide a full service description, Narayanan and McIlraith use the machinery of situation calculus and its execution behavior described with Petri Nets. They rely on the simulation and modeling environment KarmaSIM to translate DAML-S mark-up to situation calculus and Petri Nets. In this work, three QoS properties are analyzed: reachability, liveness and the existence of deadlocks.

[MVRT05] presents a framework which targets the support of the development of self-optimizing, predictive and autonomic systems for WS architectures. It adopts a simulation-based methodology which allows predicting QoS properties in different status and load conditions.

[TCW+07] first give a fairly extensive overview to the role of modeling and simulation in service-oriented software development, by comparing and contrasting traditional distributed simulation and service-oriented simulation. Then a model-driven approach is proposed to dynamically generate simulation based on the specification written in the Process Specification and Modeling Language (PSML). Dynamic architecture in SOC is in fact a model-driven approach, and thus the proposed simulation approach can potentially support the simulation of dynamic architectures.

6.3.2. Automatic generation of test-beds for SOA

Since we aim at generating simulation models for, we investigated some of the approaches used to generate testing environments for compositions of web services.

Puppet (Pick UP Performance Evaluation Testbed) is an approach for the automatic generation of test-beds to empirically evaluate different QoS features of a Web Service under development. It is roughly described in [BAFP08]. The generation process exploits the information about the coordinating scenario, the services description and the specification of the agreements that the roles will abide. In particular, the work focuses on assessing that a specific service implementation can afford the required level of QoS (e.g., latency and availability) defined in a corresponding QoS specification for a composition of services (choreography/orchestration) in which the service under evaluation will play a role.

[GHLL06] reports the extensions and the application of a test-bed generator to MaramaMTE[1], for use in the domain of service-oriented architecture performance analysis. The aim of that work was to allow service compositions (static or dynamic ones) to be modeled and test-beds to be generated (again statically or dynamically) to assess the performance of both models and the resulting service-oriented

[1] https://wiki.auckland.ac.nz/display/csidst/MaramaMTE

systems. Formal models of service compositions are described at a high level using either BPMN or Tool Abstraction-based modeling approaches. These high-level formal compositions are extended with a lower-level service composition model. From the composition and load models MaramaMTE generates one or more performance test-beds, which are executed to stress-test the service compositions.

SOABench [BBD10] focuses on Web service compositions described as BPEL processes, running on execution environments consisting in a BPEL engine plus additional components such as service registry or enterprise service buses. The current implementation of SOABench provides support for gathering and computing the response time of processes and services (as measured by testing agents), the network traffic generated by a server and the number of threads created by a server. SOABench relies on Weevil (Wang et al. 2006), a framework that provides model-based configuration of testbeds, automated script construction, workload generation, experiment deployment and execution.

6.3.3. Simulation of labeled transition systems models

. The Coordination Delegates synthesis process is based on the Labeled Transaction System (LTS) formalism, which is normally used to model distributed systems. A number of works have dealt with the simulation of generic distributed systems and software architectures modeled as LTSs, as reported in [BS01], although none of them was explicitly targeted at choreographies of services.

[BIM98] describes a method for the automatic derivation of a queuing network model from a software architecture specification, described using the CHAM formalism (CHemical Abstract Machine). Informally, the CHAM specification of a software architecture is given by a set of molecules which represent the static components of the architecture, a set of reaction rules which describe the dynamic evolution of the system through reaction steps, and an initial solution which describes the initial static configuration of the system. The paper presents an algorithm to derive a QN model from the CHAM specification of a software architecture. It is based on the analysis of the LTS that represents the dynamic behavior of the CHAM architecture, and that can be automatically derived from the CHAM specification. The algorithm does not completely define the QN model whose parameters, such as the service time distributions and the customers arrival processes, have to be specified by the designer. The solution of the QN model is derived by analytical methods or possibly by symbolic evaluation. Parameter instantiation identify potential implementation scenarios and the performance results allow providing insights on how to carry on the development process in order to satisfy given performance criteria.

The approach proposed by [ABI01] concerns the derivation of QN models from LTS describing the dynamic behavior of software architectures. Starting from a LTS description of a software architecture makes it possible to abstract from any particular software architecture specification language. The approach assumes that LTSs are the only knowledge on the system that they can use. This means, in particular, that it does not use any information concerning the system implementation or deployment. This approach is an extension of (Aquilani et al. 2000), considering a finite state representation independent of a specific architectural description language and modeling more complex interaction patterns and synchronization constraints that can be represented by extended QN (EQN). Such EQN models the software concurrent execution and component interaction at the software architecture design level.

Bibliography

[ABI01] F. Aquilani, S. Balsamo, and P. Inverardi. Performance analysis at the software architectural design level. *Performance Evaluation*, 45(2):147–178, 2001.

[ARS12a] Marco Autili, Davide Di Ruscio, and Amleto Di Salle, editors. *CHOReOS Dynamic Development Model Definition*. Number Public Project Deliverable D2.1. The CHOReOS Consortium, October 2012.

[ARS12b] Marco Autili, Davide Di Ruscio, and Amleto Di Salle, editors. *Initial Architectural Style for CHOReOS Choreographies*. Number Public Project Deliverable D1.3. The CHOReOS Consortium, October 2012.

[Ast03] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall PTR, July 2003.

[BAFP08] Antonia Bertolino, Guglielmo Angelis, Lars Frantzen, and Andrea Polini. Model-based generation of testbeds for web services. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 266–282. Springer Berlin Heidelberg, 2008.

[BAP11] A. Bertolino, G. De Angelis, and A. Polini, editors. *Governance V&V policies and rules*. Number Public Project Deliverable D4.1. The CHOReOS Consortium, 2011.

[BAP12] A. Bertolino, G. De Angelis, and A. Polini, editors. *V&V tools and infrastructure – strategies, architecture and implementation*. Number Public Project Deliverable D4.2.2. The CHOReOS Consortium, 2012.

[BBC+12] C. Bartolini, A. Bertolino, A. Ciancone, G. De Angelis, and R. Mirandola. Non-functional analysis of service choreographies. In *Proc. of PESOS*. IEEE-CS, June 2012.

[BBC+13] Cesare Bartolini, Antonia Bertolino, Andrea Ciancone, Guglielmo De Angelis, and Raffaela Mirandola. Apprehensive qos monitoring of service choreographies. In *Proc. of the ACM Symposium on Applied Computing (SAC 2013)*. ACM, 2013. – to appear. Accepted on 5th Nov.2012.

[BBD10] Domenico Bianculli, Walter Binder, and Mauro Luigi Drago. Automated performance assessment for service-oriented middleware: a case study on bpel engines. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 141–150, New York, NY, USA, 2010. ACM.

[BBG08] S. Balsamo, M. Bernardo, and V. Grassi. Quantitative analysis of software architectures. 2008.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley, Boston, 2003.

[BIK+07] Moshiur Bhuiyan, M. M. Zahidul Islam, George Koliadis, Aneesh Krishna, and Aditya Ghose. Managing business process risk using rich organizational models. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 509–520, Washington, DC, USA, 2007. IEEE Computer Society.

[BIM98] S. Balsamo, P. Inverardi, and C. Mangano. An approach to performance evaluation of software architectures. In *Proceedings of the 1st international workshop on Software and performance*, pages 178–190. ACM, 1998.

[BMKM12] Felipe Besson, Paulo Moura, Fabio Kon, and Dejan Milojicic. Rehearsal: a framework for automated testing of web service choreographies. In *Brazilian Conference on Software: Theory and Practice*, Natal-RN, Brésil, September 2012. The research leading to these results has received funding from HP Brasil under the Baile Project and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet).

[BS01] Simonetta Balsamo and Marta Simeoni. Deriving performance models from software architecture specifications. In *In: Proceedings of the European Simulation Multiconference, Analytical and Stochastic Modelling Techniques*, 2001.

[CDF+11] Andrea Ciancone, Mauro Luigi Drago, Antonio Filieri, Vincenzo Grassi, and Raffaela Mirandola. Klapersuite: an integrated model-driven environment for non-functional requirements analysis of component-based systems. In *Proc of TOOLS*, pages 99–114, 2011.

[CHO11] CHOReOS Project Team. Choreos perspective on the future internet and initial conceptual model. www.choreos.eu, April 2011.

[CHO12a] CHOReOS Project Team, Public Project Deliverable D3.2.2. *CHOReOS Middleware Implementation: Second Version*, October 2012.

[CHO12b] CHOReOS Project Team, Public Project Deliverable D6.2. *"Passenger-friendly airport" services & choreographies design*, April 2012.

[CHO12c] CHOReOS Project Team, Public Project Deliverable D4.2.1. *V&V tools and infrastructure - strategies, architecture and first implementation*, March 2012.

[CHO12d] CHOReOS Project Team, Public Project Deliverable D4.2.2. *V&V tools and infrastructure - strategies, architecture and implementation*, October 2012.

[CMS+03] Senthilanand Chandrasekaran, John A. Miller, Gregory S. Silver, Budak Arpinar, and Amit P. Sheth. Performance analysis and simulation of composite web services. *Electronic Markets*, 13(2):120–132, 2003.

[CMSA02] Jorge Cardoso, John Miller, Amit Sheth, and Jonathan Arnold. Modeling quality of service for workflows and web service processes. *Journal of Web Semantics*, 1:281–308, 2002.

[CSM+02] Senthilanand Chandrasekaran, Gregory Silver, John A. Miller, Jorge Cardoso, and Amit P. Sheth. Web service technologies and their synergy with simulation. In *in Proceedings of the 2002 Winter Simulation Conference*, pages 606–615, 2002.

[DBP12] G. De Angelis, A. Bertolino, and A. Polini. Validation and verification policies for governance of service choreographies. In *Proc. of the 8th International Conference on Web Information Systems and Technologies (WEBIST 2012)*, Porto, Portugal, Apr. 2012. SciTePress.

[Del97] Marcio Eduardo Delamaro. *Mutação de interface: Um critério de adequação interprocedimental para o teste de integração.* PhD thesis, University of São Paulo – Physics Institute, SP, Brazil, 1997.

[DPB+11] A. Di Marco, C. Pompilio, A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta. Yet another meta-model to specify non-functional properties. In *Proc. of QASBA*, pages 9–16. ACM, 2011.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, first edition, 2009.

[GGvD10] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Evaluation of online testing for services: a case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '10, pages 36–42. ACM, 2010.

[GHLL06] John Grundy, John Hosking, Lei Li, and Na Liu. Performance engineering of service compositions. In *Proceedings of the 2006 international workshop on Service-oriented software engineering*, SOSE '06, pages 26–32, New York, NY, USA, 2006. ACM.

[GMRS08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. The common component modeling example. chapter KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability, pages 327–356. Springer-Verlag, Berlin, Heidelberg, 2008.

[GN12] Alfredo Goldman and Yanik Ngoko. On graph reduction for qos prediction of very large web service compositions. In *International Conference on Service Oriented Computing (SCC)*, pages 258–265, Hawai, USA, 2012. IEEE Press.

[GNM12] Alfredo Goldman, Yanik Ngoko, and Dejan Milojicic. An analytical approach to predicting qos
of web services choreographies. In *International WorkShop on Middleware for Grid and eScience*, page submitted, 2012.

[HBC+12] A. Ben Hamida, A. Bertolino, A. Calabrò, G. De Angelis, N. Lago, and J. Lesbegueries. Monitoring service choreographies from multiple sources. In *Proc. of 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012)*, volume 7527 of *LNCS*, pages 134–149. Springer, Sept. 2012.

[HRH01] Jonathan Hammond, Rosamund Rawlings, and Anthony Hall. Will it work? In *RE*, pages 102–109, 2001.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[Jac95] Michael A. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., 1995.

[JDJ+10] Y. Jamoussi, M. Driss, J. Jézéquel, H. Hajjami, and B. Ghézala. Qos assurance for service-based applications using discrete-event simulation. *International Journal of Computer Science*, 7(4), 2010.

[Jef11] Ron Jeffries. What is extreme programming? Available on: http://xprogramming.com/xpmag/whatisXP#test, 2011.

[KA92] B. Kirwan and L.K. Ainsworth. *A guide to task analysis*. Taylor and Francis, 1992.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[Lam78] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[MCS02] J.A. Miller, J. Cardoso, and G. Silver. Using simulation to facilitate effective workflow adaptation. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pages 177 – 181, april 2002.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.

[MVRT05] Emilio Mancini, Umberto Villano, Massimiliano Rak, and Roberto Torella. A simulation-based framework for autonomic web services. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops - Volume 02*, ICPADS '05, pages 433–437, Washington, DC, USA, 2005. IEEE Computer Society.

[NM02] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36:46–52, October 2003.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.

[PS02] Fabio Paternò and Carmen Santoro. Preventing user errors by systematic analysis of deviations from the system task model. *Int. J. Hum.-Comput. Stud.*, 56:225–245, February 2002.

[SC04] S-CUBE. Project web site: Secse.eng.it, 2004.

[SOA09] SOA4All Project Team, Public Project Deliverable D5.3.1. *First Service Discovery Prototype*, September 2009.

[TCW+07] W.T. Tsai, Zhibin Cao, Xiao Wei, Ray Paul, Qian Huang, and Xin Sun. Modeling and simulation in service-oriented software development. *Simulation*, 83(1):7–32, January 2007.

[Tea10] CHOReOS Project Team. Choreos state of the art, baseline, and beyond - public project deliverable d1.1, December 2010.