Large Scale Choreographies for the Future Internet

ICT IP Project

Deliverable D5.2

Specification of the CHOReOS IDRE

http://www.choreos.eu

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

Deliverable Number	:	D5.2
Title of Deliverable	:	Specification of the CHOReOS IDRE
Nature of Deliverable	:	Report
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	VA
Contractual Delivery Date	:	30/09/2011
Contributing WP	:	WP5
Editor(s)	:	Amira Ben Hamida (EBM)
Author(s)	:	Amira Ben Hamida (EBM), James Lockerbie (CITY), Antonia Bertolino, Guglielmo De Angelis (CNR), Nikolaos Georgantas, Animesh Pathak (INRIA), Rokas Bartkevicius (NME), Pierre Chatel (THA), Marco Autili, Massimo Tivoli, Davide Di Ruscio (UDA), Apostolos Zarras (UOI), Felipe Besson, Carlos Eduardo Moreira dos Santos, Daniel Cukier, Leonardo Alexandre Ferreira Leite, Gustavo Oliva, Yanik Ngoko(USP).
Reviewer(s)	:	Gianmarco Panza (CEFRIEL), Valérie Issarny (INRIA)

Abstract

This deliverable focuses on the design of the CHOReOS Integrated Development and Runtime Environment, aka CHOReOS IDRE, based on the supporting solutions developed within WP2, WP3 and WP4 during CHOReOS' 1st year. The document provides an overall description of the IDRE components, their respective functionalities and the integration dependencies between them, thereby defining the integration points between the components developed in WP2-3-4.

Keyword list

Choreography, Cloud, Grid, IDRE, Integration, Development, Governance, Middleware, Monitoring, Runtime, Service, Service Discovery, Service Access, Service Composition, TDD, V&V

Document History

Version	Changes	Author(s)
1.0	Outline	Amira Ben Hamida, Apostolos Zarras, Fabio Kon, Hugues Vincent
2.0	Agreed Outline Input from UDA, UOI, USP Added a general view for the whole IDRE	Amira Ben Hamida, Apostolos Zarras, Fabio Kon, Daniel Cukier, Gustavo Oliva, Yanik Ngoko
2.1	Updated version with USP entries	Amira Ben Hamida, Daniel Cukier, Gustavo Oliva, Carlos Eduardo Santos, Felipe Besson, Yanik Ngoko
2.3	Inputs from CITY, CNR Modification to section on synthesis processor Adding a high level scenario	Amira Ben Hamida, Guglielmo Deangelis, James Lockerbie,
2.4	Input from USP, UOI, Inria	Felipe Besson, Apostolos Zarras, Carlos Eduardo Moreira dos Santos, Daniel Cukier, Leonardo Alexandre Ferreira Leite, Animesh Pathak
2.5	Input from THALES, UDA, CNR Editing outline	Pierre Chatel, Amira Ben Hamida, Guglielmo De Angelis, Marco Autili, Massimo Tivoli, Davide Di Ruscio
2.5.1	Input from CNR	Guglielmo Deangelis
2.6	Input from USP, CNR, CITY	Guglielmo De Angelis, Gustavo Oliva, James Lockerbie, Felipe Besson, Fabio Kon, Perdo Leal
2.7	Input from UOI, EBM	Apostolos Zarras, Amira ben Hamida
2.8	Monitoring subsystem Reconfiguration for service substitution	Amira ben Hamida, Apostolos Zarras,
3.0	Input from Inria, USP Outline updated wrt middleware specification detailed in D3.1 Added Introduction, ULS challenges, IDRE Overview	Nikolaos Georgantas, Animesh Pathak, Felipe Besson, Fabio Kon, Perdo Leal
3.1	Outline changed Updates by Inria and UOI on middleware Conclusion	Nikolaos Georgantas, Apostolos Zarras, Animesh Pathak, Amira Ben Hamida
3.2	Updates from Inria on middleware sections Comments from Inria integrated for governance and DSB sections	Nikolaos Georgantas, Amira Ben Hamida

Document Review

Review	Date	Ver.	Reviewers	Comments
Outline	15/09/11	1.0	All authors	OK
Draft	10/10/11	2.0	All authors	Revision
QA	02/11/11	3.0	Nikolaos Georgantas, Gianmarco Panza	Revision
PTC	02/11/11	A	Valérie Issarny	Editorial comments

Glossary, acronyms & abbreviations

Item	Description
AoSBM	Abstraction-oriented Service Base Management
BPEL	Business Process Execution Language
C&E	Composition & Estimation
CEP	Complex Event Processor
CS	Client/Server
DOW	Description of Work
DPWS	Device Profile for Web Services
DSB	Distributed Service Bus
EAI	Enterprise Application Integration
ESB	Enterprise Service Bus
FI	Future Internet
GA	Generic Application
IDRE	Integrated Development and Runtime Environment
IoBS	Internet of Business Services
IoS	Internet of Services
IoT	Internet of Things
IoTS	Internet of Things-based Services
KB	Knowledge Base
LSB	Lightweight Service Bus
PS	Publish/Subscribe
QoS	Quality of Service
SCA	Service Component Architecture
SOA	Service Oriented Architecture
SOC	Service Oriented Computing
TD	Things Discovery protocol
TS	Tuple Space
UML	Unified Modeling Language
VBA	Visual Basics for Applications
WP	Work Package
WSDL	Web Service Description Language
XSB	eXtensible Service Bus
XSC	eXecutable Service Composition
XSD	eXtensible Service Discovery

Table of Contents

Introduction

1. Introduction

Sketching the specification of the CHOReOS IDRE is a major step in the lifecycle of the project. In particular, such a specification shall prescribe the integration points among the various constituting pieces that are developed in WP2-3-4.

Reading Key

The purpose of this deliverable is to provide a first version of the specification of the CHOReOS IDRE, based on the results of WP2-3-4 at the end of Year 1. The proposed specification captures and conveys the significant architectural decisions along with their associated architectural drivers. We describe each component and initially decompose it into subcomponents when possible. Furthermore, subcomponents, interfaces and integration dependencies are defined for all the top-level components of the IDRE, although these definitions are called to evolve as the development of components progresses.

This deliverable is relevant to the CHOReOS RTD work packages WP2 to WP5. In general, the target audience includes: component providers, users, and any person inside or outside the CHOReOS project interested in learning about the internal processing of the CHOReOS IDRE.

This deliverable presents the technological fundamentals, guidelines and details for the implementation of the CHOReOS IDRE out of the components being developed in related work packages WP2-3-4.

Design Paradigms

The CHOReOS IDRE is dedicated to Future Internet environments and will use services from ultra large scale infrastructures. Design decisions need to be adopted in order to make the IDRE architecture suitable for such environments. Modularity and separation of concerns, which are essential aspects of the SOA paradigm, offer the capability of designing the system from the integration of services and components. Such systems are more adaptable to a highly distributed and dynamic context. For the design of the IDRE and the elements composing it, we fully rely on a modular and flexible SOA architecture.

Document structure

The remainder of this deliverable is organized as follows:

- Section 2 sketches an overview of the CHOReOS IDRE and surveys how it answers the FI requirements.
- Sections 3 to 9 present in detail the main subsystems of the IDRE and their subcomponents, where:
  - Section 3 is dedicated to the CHOReOS *Development Environment* (developed within WP2),
  - Sections 4 to 7 to the CHOReOS *Runtime Environment* (aka Middleware that is developed within WP3), and
  - Sections 8 and 9 to the *Governance and V&V framework* (developed within WP4) embedded in the IDRE.
- Section 10 concludes the document.

2. CHOReOS IDRE General Architecture

The CHOReOS IDRE aims at gathering the technological contributions issued from the CHOReOS WP2-3-4. Furthermore, the CHOReOS IDRE will be tested and assessed based on the use cases that will be developed in WP6-7-8.

Figure 2-1 illustrates the key integration role of WP5, depicting the various CHOReOS work packages and their respective scopes and dependencies.

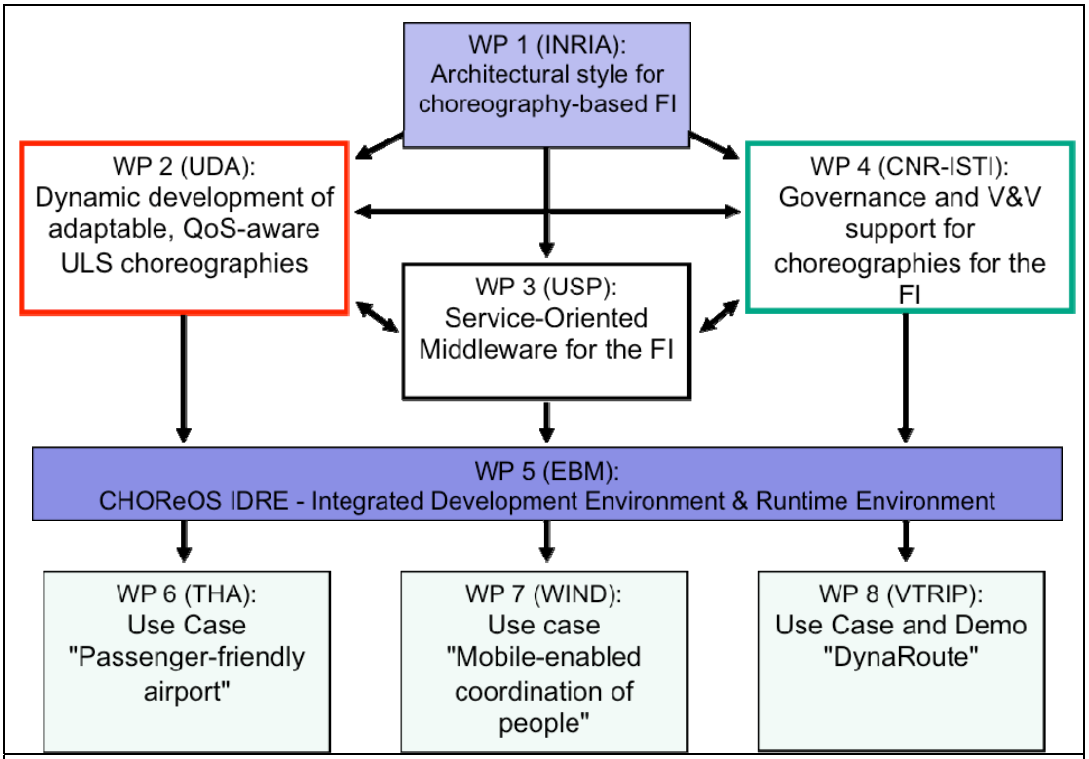


Figure 2-1. CHOReOS Work Packages Overview

2.1. CHOReOS IDRE Overview

The CHOReOS IDRE relies on a modular service-oriented architecture where a number of top-level coarse-grained components/subsystems are integrated in order to support the overall *development*, from design to implementation, together with *deployment* and *execution*, of *services choreographies in the FI*. Still, note that those top-level components are developed within WP2-3-4, while WP5 concentrates on their integration.

Figure 2-2 gives an overview of the specification of the CHOReOS IDRE, depicting the IDRE's top-level components that are developed within WP2-3-4, together with the main integration dependencies between them.

Hereafter, we briefly describe each of the IDRE top-level components (from top to bottom, left to right on the figure):

- **CHOReOS Development Environment:** The CHOReOS project adopts a *model-driven choreography development process*. First, this process allows for the specification of user requirements, thanks to dedicated *Requirements Specification Tools* (e.g., ontology for non-functional requirements and service quality measures, user task models, choreography patterns). The final output of the requirements specification activity is a *choreography specification* (in the BPMN2 language), which serves as input to the next phases of the overall process. Second, the *Synthesis Processor* allows for the automated synthesis of the coordination delegates that will coordinate the collaboration among the participant (choreographed) services so as to implement the specified choreography. These coordination delegates are deployed and

executed on top of the CHOReOS *Middleware* (i.e., CHOReOS XSC, XSD, XSA and Cloud & Grid Middleware discussed below). Third, the development process ends with the *Choreography Analyser* that performs the scalability analysis of the choreography in order to make it cope with the FI requirements. The components related to the CHOReOS Development Environment are further presented in Section 3.

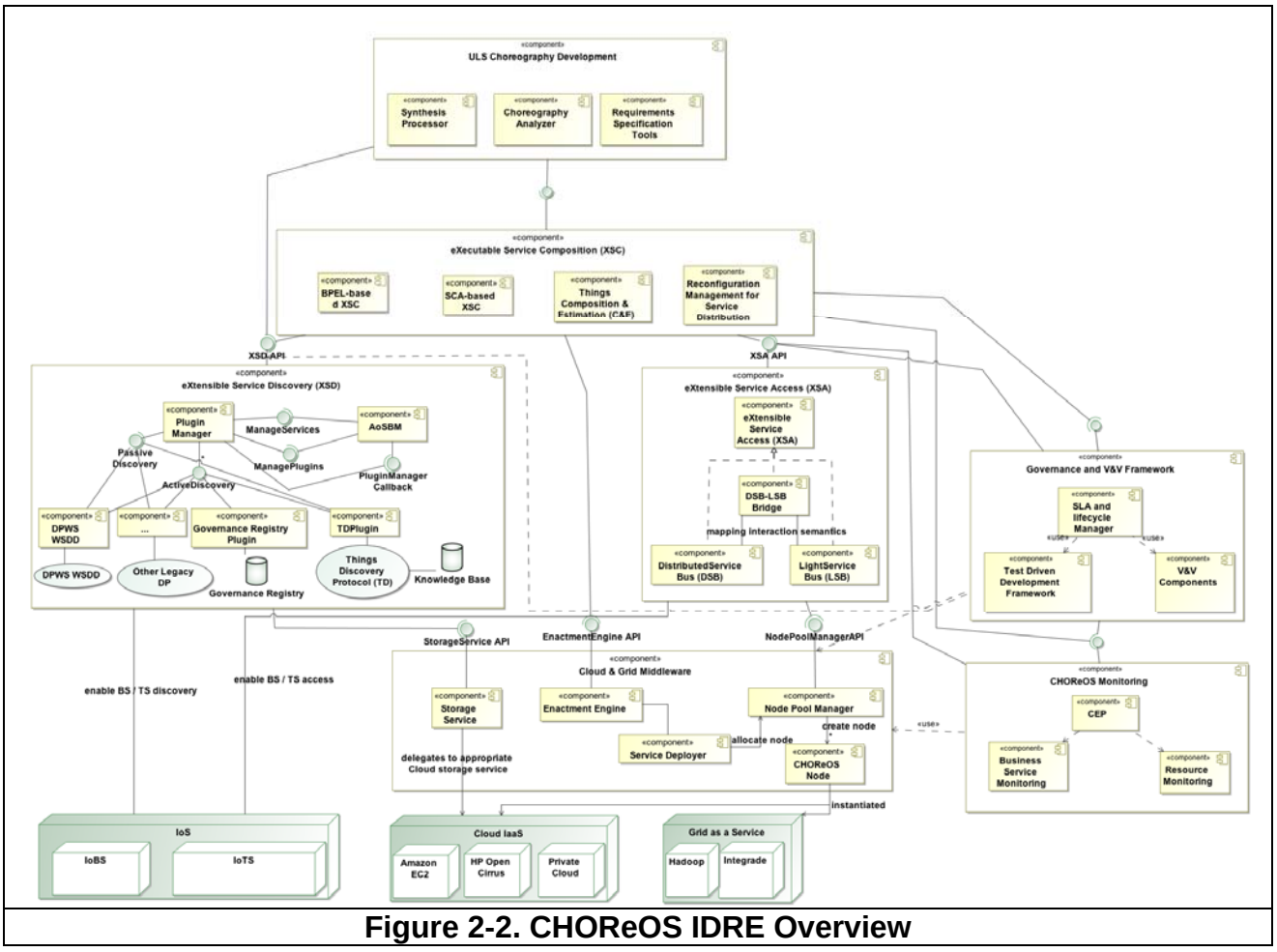


Figure 2-2. CHOReOS IDRE Overview

- **eXecutable Service Composition (XSC):** Service choreographies in CHOReOS are supported by different execution platforms so as to cope with the diversity of service technologies found in a ULS environment. Specifically, *BPEL-based XSC* enables the implementation of coordination delegates using BPEL, while *SCA-based XSC* supports the implementation of coordination delegates using SCA. In a complementary way, the *Thing-based Composition & Estimation* component deals with the composition of Thing-based services to handle requests for interacting with the physical world. The CHOReOS XSC relies also on the *Reconfiguration Management for Services Substitution* component for dynamically substituting services that no longer respect their initial service level agreements. The XSC components are addressed in Section 4.
- **eXtensible Service Discovery (XSD):** The CHOReOS IDRE provides a multi-protocol service discovery service. Actually, it relies on an *Abstraction-oriented Service Base Management (AoSBM)* that stores and classifies in a suitable way an important amount of services data. This base is populated by an extensible mechanism, the *Plugin Manager.* The latter is responsible of extending the service discovery to both business services and FI things, by plugging domain-specific discovery protocols like, e.g., the *Governance Registry for Business services* and the *Things Discovery Protocol (TDP)***.** The XSD-related components are presented in Section 5.

- **eXtensible Service Access (XSA):** CHOReOS *eXtensible Service Access* is based on an enhanced service bus paradigm to overcome the heterogeneity of the FI. This paradigm is represented by the *eXtensible Service Bus*, which is realized by a *Distributed Service Bus (DSB)* that sustains interaction with Business services and a *Light Service Bus (LSB)* that enables interactions with Thing-based services. These two buses are bridged thanks to an integration mechanism, the *DSB-LSB Bridge.* The interfaces and functionality of XSA are sketched in Section 6.
- **Cloud and Grid Middleware:** The CHOReOS IDRE benefits from the Cloud and Grid technologies. Indeed, we rely on a highly distributed and flexible Cloud and Grid infrastructure that embeds the *Enactment Engine, Node Pool Manager, Service Deployer* and *Third Party IaaS Providers* so as to enable exploitation of the Cloud and Grid technologies for realizing choreographies as well as middleware services. These components are discussed in Section 7.
- **Governance and V&V Framework:** The CHOReOS *Governance and V&V framework* realizes a set of governance and V&V activities that are dedicated to administrating and assessing ULS choreographies. Precisely, the *SLA and lifecycle manager* manages the lifecycle of relevant resources such as services, service level agreements, and choreographies. The *V&V Components* further perform the testing of services before their involvement in choreographies, and also online testing of services and choreographies at runtime, during their operation. The *Test Driven Development* Framework operates a series of tests on the choreographies in a complementary way. The governance and V&V framework is the focus of Section 8.
- **CHOReOS Monitoring:** The monitoring functionality enables the assessment of the good behaviour of running Business services in the CHOReOS context. Once services are deployed on top of the DSB, *CHOReOS Monitoring* performs runtime assessment of service level agreements and control of communications within a choreography, thanks to the *Business Service Monitoring*. Moreover, the hardware resources of the Cloud are monitored using *Resource Monitoring*. Finally, monitored data serves detecting contract violations at the level of the *Complex Event Processor (CEP).* The definition of the monitoring component is addressed in Section 9.

2.2. CHOReOS IDRE Responses to the FI Requirements

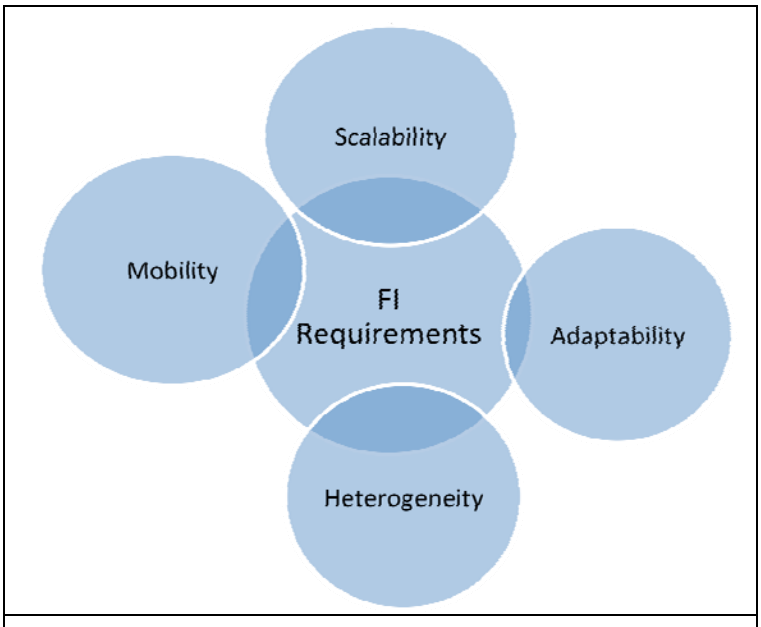


Figure 2-3. Future Internet Challenges

The CHOReOS IDRE is intended to be deployed and used in ultra large scale environments. According to Deliverable D1.2, the IDRE must in particular cope with the following FI challenges (see Figure 2-3):

- **Scalability** regarding the number of services, resources and users involved;
- **Mobility** of devices and users:
- **Adaptability** of the IDRE and choreographies of services;
- **Heterogeneity** of services and resources.

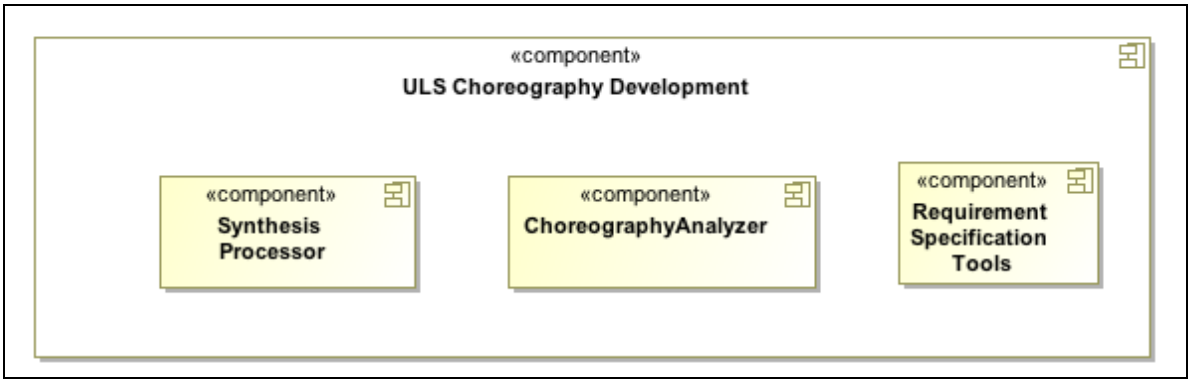
As detailed in the associated Deliverables D2.1, D3.1 and D4.1, the above challenges are accounted for in the design of all the constituting elements of the IDRE. The interested reader is referred to those documents for detail about the IDRE constituents, while the present document concentrates on the definition of their interfaces and inter-dependencies.

The design of the CHOReOS IDRE fully adheres to the principles of modularity and separation of concerns. In particular, the IDRE components presented in this deliverable provide their functionality as services through exported interfaces. Any service of the CHOReOS IDRE components can then be exposed for use by third parties.

CHOReOS Development Environment

3. ULS Choreography Development

As depicted below, the basic components/subsystems of the *ULS Choreography Development* subsystem are: the *Requirements Specification Tools,* the *Synthesis Processor*, and the *Choreography Analyser*. In what follows, we describe for each component: the related composition, interfaces and functionality and integration dependencies with other components.



3.1. Requirements Specification Tools

The *Requirements Specification Tools* subsystem (see Figure 3-1) is mainly responsible for enabling domain experts to specify functional and quality requirements on services and service-based applications, and in turn, to enable the domain expert to produce a first draft choreography specification (see Deliverable D2.1).

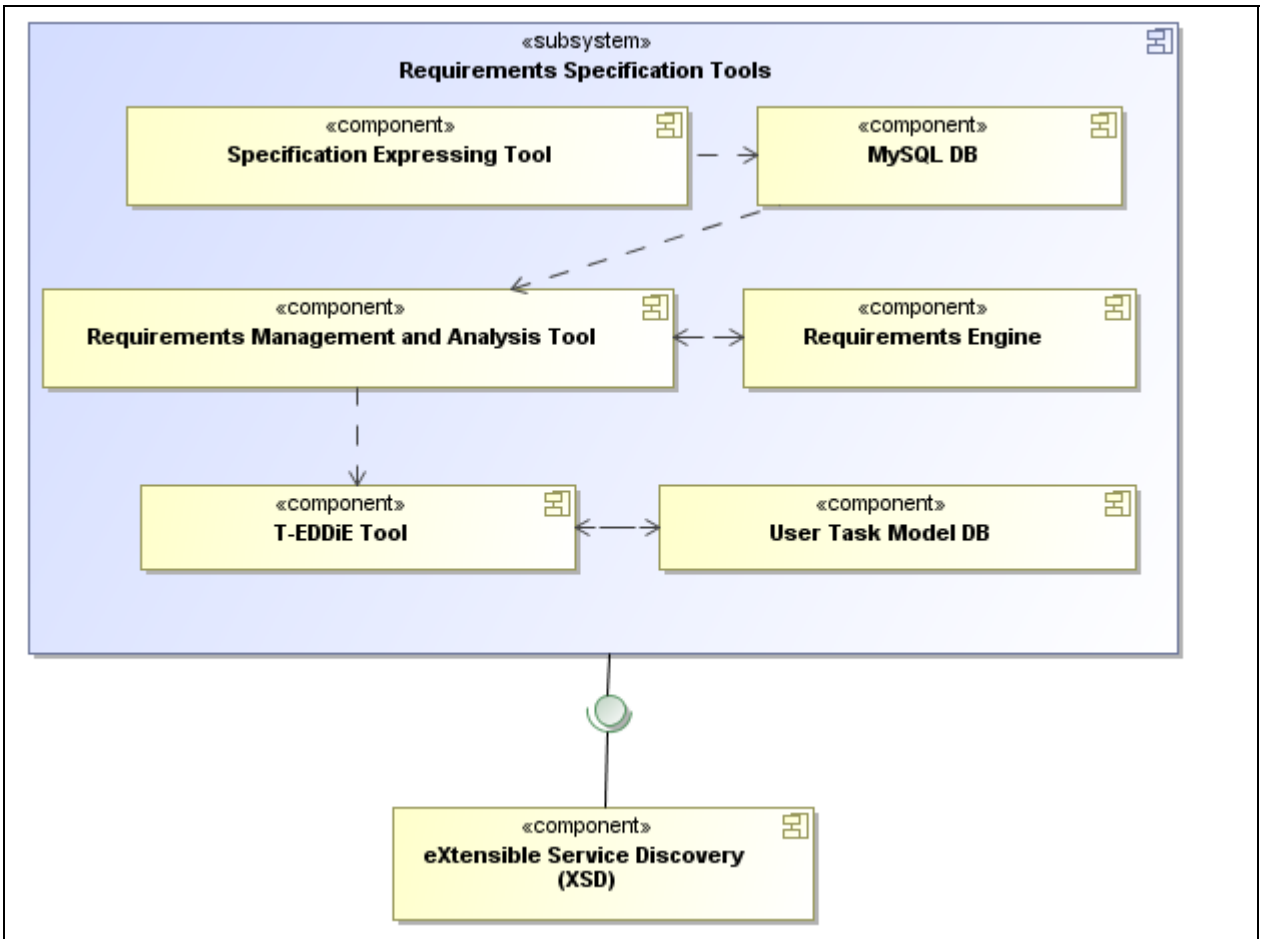


Figure 3-1. Requirements Specification Tools Architecture

Subcomponents

- **Specification Expressing Tool and DataBase:** The responsibility of this component is to provide the domain expert with service consumer requirements and associated attributes. The service consumer specifies requirements, or user needs, using a structured approach facilitated by mobile tools – such as the iPhone app (application) that has been developed for the CHOReOS project. These requirements and attributes represent the inputs to the overall domain expert specification process, of which there can be many service consumers with many user needs. The expressed requirements are recorded in a *MySQL DataBase* along with attributes for quality, priority and situation (longitude, latitude and time). Associated with the expression of service consumer requirements is a quality model, which relates the user requirements on service-based applications to Quality of Service (QoS) on services aggregated in these applications.
- **Requirements Management and Analysis Tool:** This component is responsible for providing the domain expert with requirements management and analysis functions implemented in an MS Excel front end developed for the CHOReOS project. The Excel-based tool accesses the MySQL requirements database and provides functions implemented in the Excel VBA (Visual Basic for Applications) project for the domain expert to manipulate the data. For example, they can edit and amend service consumer requirements and also add their own requirements. The domain analyst is able to generate different views of the data set by filtering the requirements, for example selecting the last 100 requirements entered into the database (based on time stamps). These functions are provided to help the domain expert to pull out individual requirements in order to form a set of requirements for choreography.
- **Requirements Engine:** This component is responsible for executing a matching and grouping algorithm to cluster the service consumer and domain expert expressed requirements. A 'calculate similarity' algorithm, derived from work undertaken on the SeCSE project (http://www.secse-project.eu/), enables two text strings, i.e., requirements, to be compared for similarity using natural language processing techniques. These techniques use the WordNet [MBFGM90] on-line lexicon, which provides word senses, and definitions, which disambiguate terms, and also adds semantic relations between terms with similar meanings to make the similarity calculation more accurate. The calculate similarity web service is called from the Excel tool by posting a SOAP (Simple Object Access Protocol) 1.1 envelope via an HTTP request written in VBA code. The output from this component is grouped requirements for choreographies.
- **T-EDDiE Tool and User Task Model Database:** This component is responsible for matching the requirements on the choreography specification to user task models [PS02] using the T-EDDiE tool [ZKM-TR]. A set of CTT task models, describing structured activities that are often executed during the interaction with a system, will be defined for CHOReOS and stored in a database. The task-based algorithm can use links between knowledge about the user's problem domain and the solution space of classes of service to formulate a set of terms to parse and match to task models. Finally, the prioritized quality-based requirements and user task models are then associated with choreography strategies, which are expressed in the form of patterns by the choreography designer. The final output of the domain expert specification will be a first draft choreography specification and a set of associated requirements to inform the discovery of abstract services.

Interfaces and Functionality

The *Requirements Specification Tools* are not defining external explicit interfaces, yet.

Integration Dependencies

The *Requirements Specification Tools* are at the beginning of the specification process. They are self-contained and do not have dependencies to external components, while the first draft specification that is produced is used as input by other components of the *Choreography Development* subsystem.

3.2. Synthesis Processor

The *Synthesis Processor* subsystem (see Figure 3-2) is mainly responsible of synthesizing the coordination delegates (see Deliverable D1.3) that are in charge of suitably coordinating the services participating to the choreography.

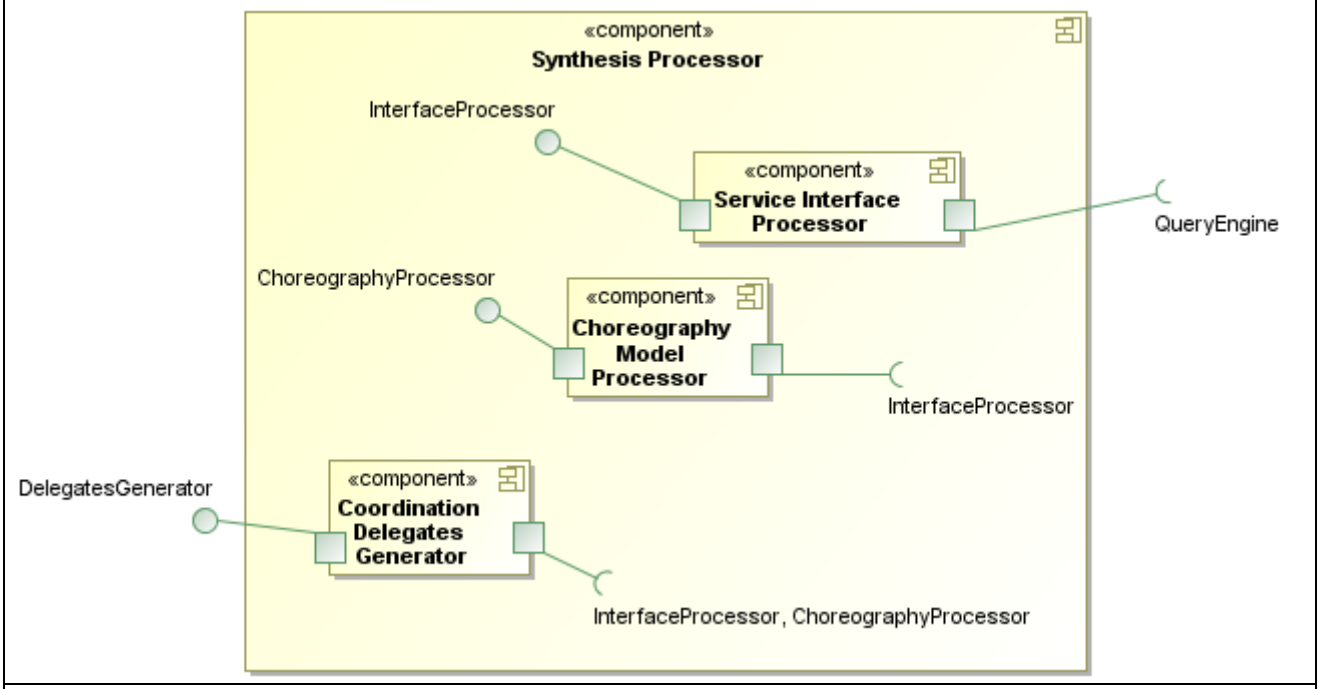


Figure 3-2. Synthesis Processor Architecture

Subcomponents

- **Service Interface Processor:** This component is responsible to process a service interface description (e.g., a WSDL), and automatically derive a partial ordering relation among the different service operations. This partial ordering relation is represented as an automaton, called *Behaviour Protocol automaton* that models the service interaction protocol. This automaton explicitly models also the I/O data of the service operations. More precisely, the states of the behaviour protocol automaton are service execution states and the transitions, labelled with operation names plus I/O data, model possible service interactions with its environment. For instance, the service interface description can be given as a WSDL (only signature) or as a WSCI (signature plus interaction protocol). Thus, two cases can be distinguished. The latter case concerns standard model-to-model transformation to translate the service interface description into an automata-based notation. In the former case, the behaviour protocol is obtained through synthesis and testing stages. The synthesis stage is driven by data type analysis, through which we obtain a preliminary dependency automaton that can be optimized by means of heuristics. Once synthesized, this dependency automaton is validated through testing against the service to verify conformance, and finally transformed into an automaton defining the behaviour protocol. The Service Interface Processor provides the *InterfaceProcessor* and requires the *QueryEngine* interface.

- **Choreography Model Processor:** The main responsibility of this subsystem is to support the processing of the automata-based specification of the choreography (see Deliverables D1.3 and D2.1). The processor learns the interaction flow globally defined by the choreography and projects it, in a peer-style fashion, among the different participants according to the "local" roles that they play to fulfil the global choreography, hence obtaining a peer-style specification of the choreography. As detailed in Deliverable D1.3, the notions of role and operation abstract the notions of participant and task, respectively, of a BPMN2 choreography specification. The *Choreography Model Processor provides* the *ChoreographyProcessor* interface and *requires* the *InterfaceProcessor* interface**.**
- **Coordination Delegates Generator:** The main responsibility of this subsystem is to support the generation of the actual code of the coordination delegates. At a high-level, the approach starts from (i) the BPMN2-based choreography model(s) and from (ii) the set of services discovered by the XSD. Input (i) comes from the refinement of the CTT models and choreography patterns obtained by means of a transformational process applied to the goals and requirements specification (see Section 3.1). Input (ii) comes from the exploitation of the *Abstractions-oriented Service Base Management (AoSBM)* mechanisms described later. Thus, as described in Deliverable D2.1, the synthesis process assumes that the services into the registry/base have been discovered so that they satisfy the local (to the service) functional and non-functional requirements that have been specified for the choreography and, hence, can be considered as potential candidates to participate in the global choreography process. Finally, the choreography synthesis produces the coordination delegates that will be then managed by the *XSC* component for choreography realization purposes (see Section 4), hence accessing the participant services through the *XSA* subsystem. The *Coordination Delegates Generator* provides: the *DelegatesGenerator* Interface, and requires the *ChoreographyProcessor* and *InterfaceProcessor* interfaces.

Interfaces and Functionality

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
InterfaceProcessor	BuildDependencyAutomaton	wsdl: URI	dfModel: DataFlowModel	Derives all the syntactic I/O data dependencies from a WSDL description and represents them into an automaton (the dependency automaton)
	ValidateDependencyAutomaton	dfModel: DataFlowModel, testCases: URI	dfModel: DataFlowModel	By means of testing, refines the dependency automaton to keep only the syntactical I/O dependencies that are semantically correct
	DA2BPA	dfModel: DataFlowModel	automaton: LTS	Performs model-to-model transformation to translate the validated dependency automaton into its corresponding behaviour protocol automaton
ChoreographyProcessor	BuildChoreographyLTS	bpmn2Diagram: URI	automaton: ChoreoLTS	Performs model-to-model transformation to translate a BPMN2 Choreography Diagram into a Choreography LTS (suitable for coordination delegates synthesis purposes)
DelegatesGenerator	CoordinationDelegatesSynthesis	sList: List<LTS>, c: ChoreoLTS	cdList: List<URI>	Given (i) a list of LTS-based service behavioural descriptions and (ii) a LTS-based specification of the choreography, derives, for each coordination delegate, a description (e.g., XML-based) of the delegate's coordination logic.

Integration Dependencies

External Interfaces required for this component

Interface Name	External Component
QueryI	Abstraction-oriented Service Base Management

3.3. Choreography Analyser

The *Choreography Analyser* component is mainly responsible for analysing either a serialized BPMN2 choreography specification or a set of coordination delegates in order to provide measures concerning scalability prediction and change impact. The *Choreography Analyser* component consists of the following subcomponents (see Figure 3-3): *Scalability Predictor, Stability and Interdependency Analyser, M2M Transformer, and Graph Analyser.*

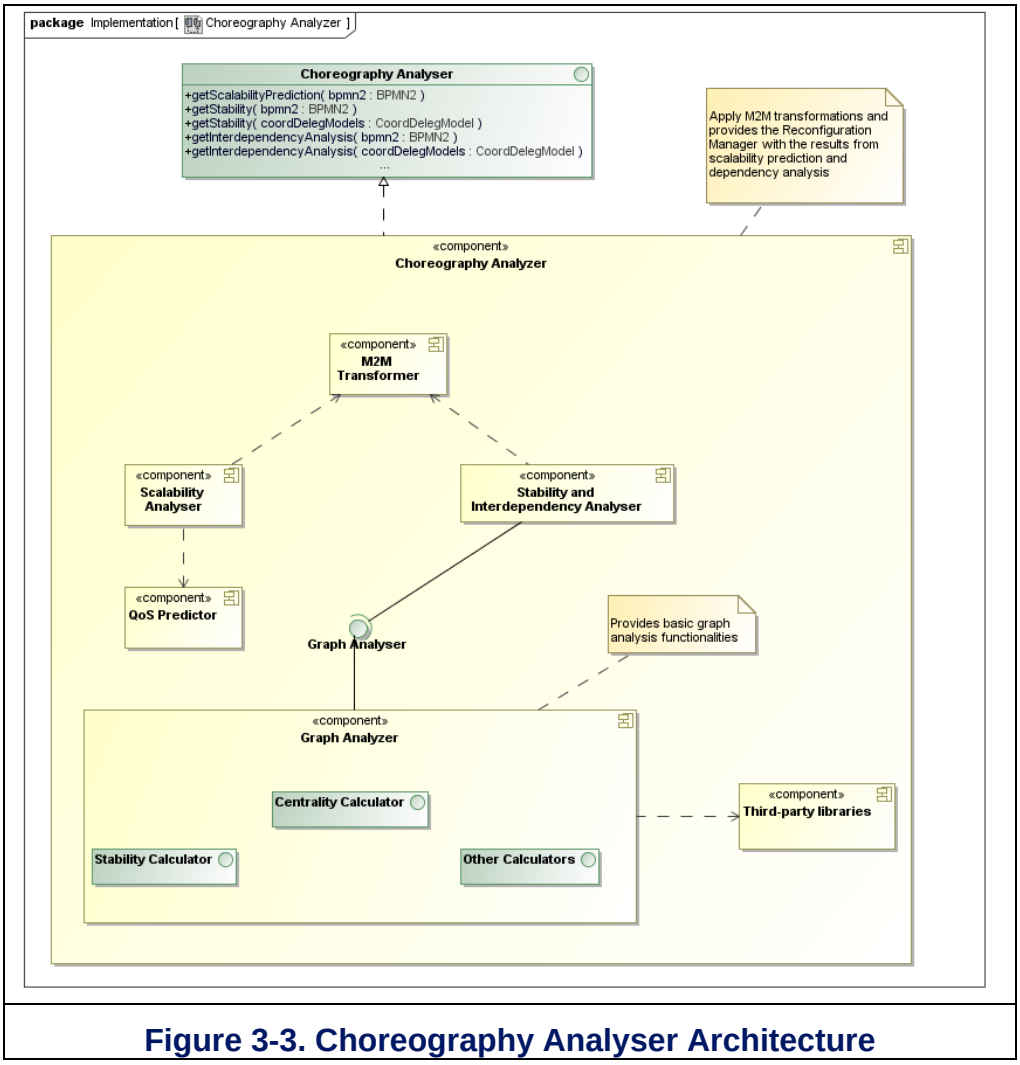


Figure 3-3. Choreography Analyser Architecture

Subcomponents

- **Predictor:** The QoS predictor in CHOReOS aims at estimating the behaviour of choreographies regarding QoS parameters: service response time, capacity, reliability, availability of a composition, etc. The main function of this component consists in taking a set of coordination delegates and providing related QoS estimation. The predictor component is based on an internal graph representation of the choreography (service dependence graph) that must be analysed. Therefore, a transformation of coordination delegates into a graph representation is required.
- **Scalability Analyser:** The role of this component consists in providing information on QoS scale. While the *QoS Predictor* aims at capturing choreography QoS information assuming knowledge on the Web Services, the *Scalability Analyser* aims at predicting the evolution of these QoS values. To this end, QoS information that has an impact on the scalability (related to the physical architecture or to the user behaviour) must be taken into account. Initially, we consider the rate of user requests as the main parameter.

- **Stability and Interdependency Analyser:** This component is primarily responsible for performing change impact analysis based on existing dependencies between participants and existing dependencies between services. This component relies on the M2M Transformer component to obtain the dependency graph. Additionally, the analyser relies on the Graph Analyser component to calculate dependency-centric measures.
- **Graph Analyser:** This component is responsible for calculating stability, centrality, and other measures from a directed graph.
- **M2M Transformer:** This component is responsible for extracting a participant-dependency graph from a choreography model represented in a BPMN2. The component is also responsible for extracting a service-dependency graph from a set of coordination delegates.

Interfaces and Functionality

The Choreography analyser subsystem provides the following interfaces and functionality:

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
QoSPredictor	extract_planning	coordDelegates:List<URI>	plan_execution: Planning	This operation takes a set of coordination delegates and transform them into an internal execution planning, which represents dependencies between services and QoS values associated to service execution
	getQos	plan_execution:Planning	qos_report: Qos Report	This method takes an input plan and outputs related QoS parameters. The QoS report provides the worst expectation, the mean expectation and the best expectation for each QoS dimension
ScalabilityAnalyser	getScalesVector	plan_execution:Planning scaling: ScalingParameters	scaling_report: ScaleVector Report	This method uses the *Qos Predicor* and the scaling parameters to estimate the scaling of an execution plan in different settings
StabilityAndInterdependencyAnalyser	analyseParticipantDependencies	bpmn2: BPMN_Model	participantDependenciesReport: DependencyReport	Calculates all centrality measures and stability for participants in a BPMN choreography model.
	analyseServiceDependencies	coordDelegates:List<URI>	serviceDependenciesReport: DependencyReport	Calculates all centrality measures and stability for services in a Coordination Delegate model
	getStability	bpmn2: BPMN_Model	stability:double	Calculates the stability of the Choreography model based on dependencies among participants
	getStability	coordDelegates:List<URI>	stability:double	Calculates the stability of the choreography based on dependencies among services. Dependencies are inferred by

				analyzing the coordination logic from delegates
	getButterflyServices	coordDelegates:List<URI>, inDegreeThreshold: double	butterflyServices: List <ServiceIdentifier>	Analyses the service dependency graph and identifies butterfly services Dependencies are inferred by analyzing the coordination logic from delegates
	getSensitiveServices	coordDelegates:List<URI>, outDegreeThreshold: double	sensitiveServices: List <ServiceIdentifier>	Analyses the service dependency graph and identifies sensitive services Dependencies are inferred by analyzing the coordination logic from delegates
	getHubServices	coordDelegates:List<URI>, degreeThreshold: double	hubServices: List <ServiceIdentifier>	Analyses the service dependency graph and identifies hub services Dependencies are inferred by analyzing the coordination logic from delegates
M2MTransformer	extractParticipantDependenciesFromBPMN	bpmn2: BPMN_Model	dependencyGraph:DependencyGraph	Extracts the participant dependency graph from a BPMN2 diagram
	extractServiceDependenciesFromCoordDelegates	coordDelegates:List<URI>	dependencyGraph:DependencyGraph	Extracts the service dependency graph from the coordination logic of the delegates
GraphAnalyser	calculateCentrality	dependencyGraph:DependencyGraph, centralityType:Enum<CentralityType>	centralityResult	Calculates the given centrality (degree, eigenvector, etc.) of the dependency graph
	calculateStability	dependencyGraph:DependencyGraph	stabilityResult	Calculates the stability of the dependency graph

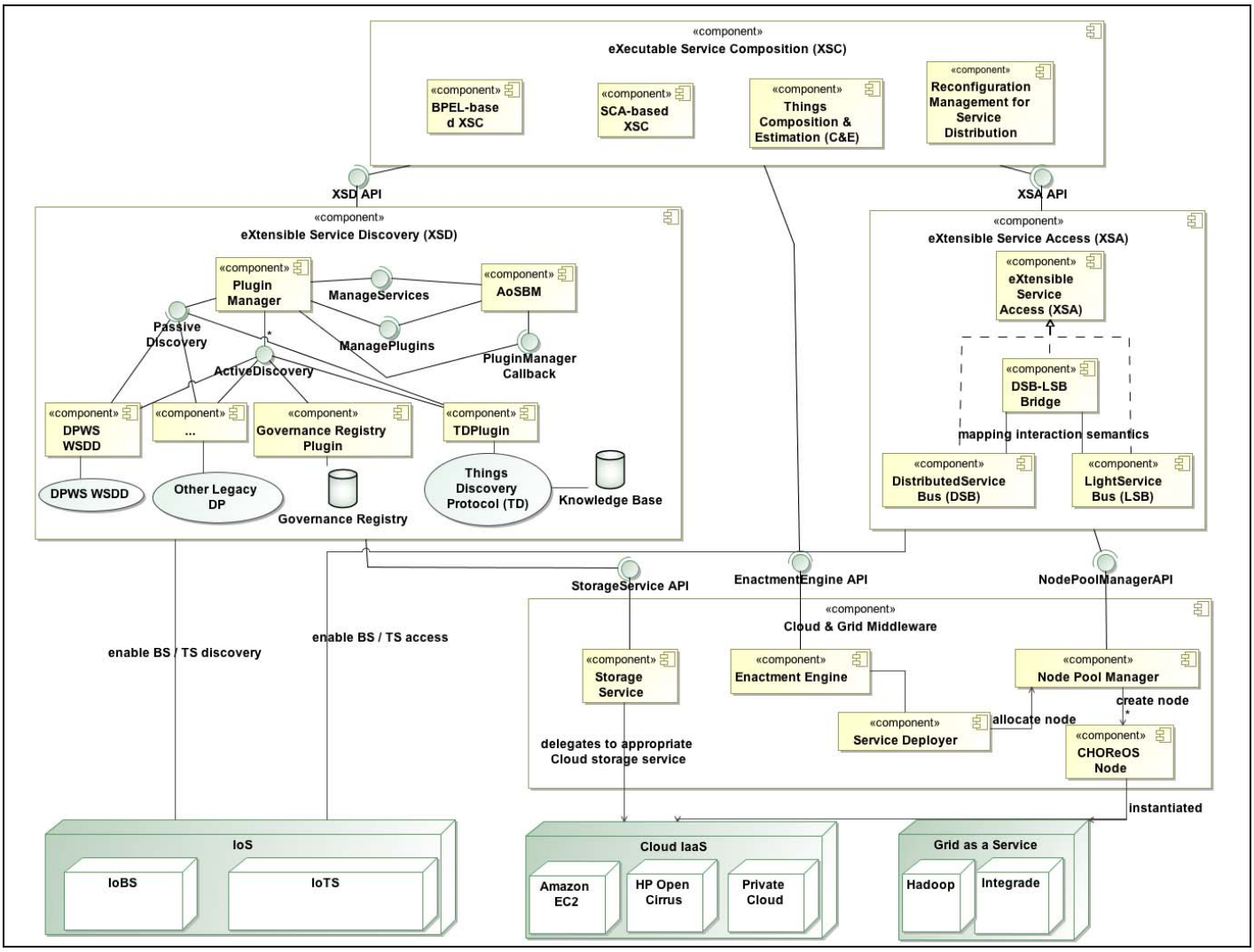
Integration Dependencies

There is no integration dependency foreseen at the moment, since this is an analysis (utility) component that is to be used by other components in the system (e.g., Reconfiguration Manager) so as to enhance and support their functionality.

CHOReOS Service-Oriented Middleware for the Future Internet

The specification of the CHOReOS *Middleware* is the subject of Deliverable D3.1, which introduces four main subsystems, as depicted below and further detailed in the following sections:

- **eXecutable Service Composition (XSC)** offers the mechanisms that allow the composition of services in the FI (i.e., integrating Business and Things based services), and in particular support the deployment and execution of *Coordination Delegates* generated by the CHOReOS *Choreography Development* subsystem.
- **eXtensible Service Access (XSA)** sustains interaction with heterogeneous services in the FI.
- **eXtensible Service Discovery (XSD)** supports the registration and lookup of services in the FI.
- **Cloud & Grid Middleware** provides the mechanisms enabling the deployment of FI services (including services provided by the IDRE) over a cloud and grid infrastructure.



4. eXecutable Service Composition (XSC)

The main responsibility of the eXtensible Service Composition - XSC subsystem is to support the deployment and enactment of CHOReOS choreographies. Figure 4 1 presents a high level view of XSC, which decomposes into the following subsystems:

- *BPEL-based XSC* and *SCA-based XSC* allow for the execution of Coordination Delegates produced by the *Choreography Developmen*t subsystem (see Section 3). As such, choreographies are not directly inputted into the BPEL- and SCA-based XSC, only the specifications of coordination delegates that enact these choreographies.
- *Things Composition and Estimation - C&E* deals with computing a composition of Thing-based services in response to a query for physical information.
- *Reconfiguration Management for Service Distribution* addresses service substitution at runtime to face the dynamics of the FI.

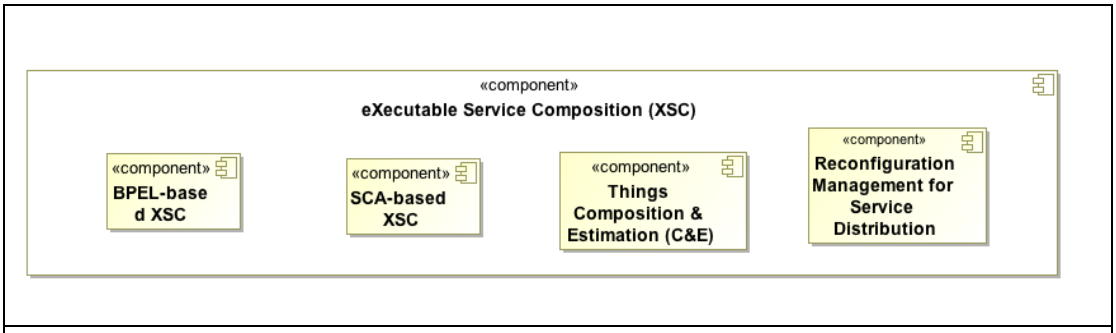


Figure 4-1. Architecture of the eXecutable Service Composition

4.1. BPEL-based XSC

The CHOREOS *BPEL-based XSC* (see Figure 4-2) enables Business services composition, enactment and monitoring.

Specifically, the *BPEL-based XSC* enables the implementation of *Coordination Delegates* using BPEL so that Coordination Delegates get executed on top of a BPEL engine, as in particular embedded in the CHOReOS DSB.

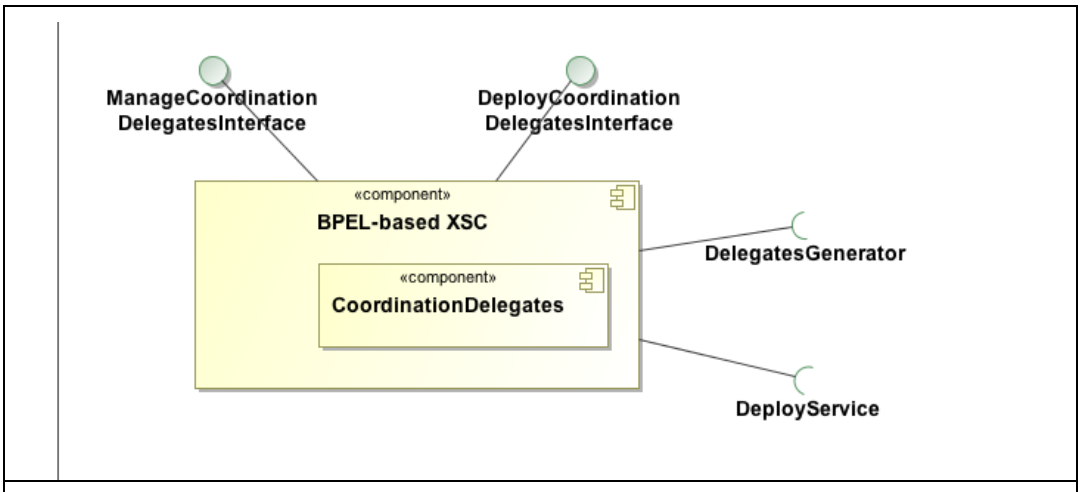


Figure 4-2. Architecture of the BPEL-based XSC

Subcomponents

As mentioned above, the *BPEL-based XSC* embeds the BPEL implementation of the Coordination Delegates that are synthesized by the *Choreography Development* subsystem (see Section 3).

Interfaces and Functionality

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
DeployCoordinationDelegateInterface	deployCoordinationDelegate deploycoordinationDelegates	URI cd URI[] cds	String String[]	Provides functionality to deploy BPEL Coordination Delegates based on into the runtime. To this end, it includes a first method to deploy a single Coordination Delegate and a second one to batch deploy multiple Coordination Delegates at the same time. Both methods accept a URI identifying the Coordination delegates, and return a single (or multiple) deployment identifier(s) that uniquely identifies(y) each deployed delegate.
ManagementCoordinationDelegatesInterface	removeCoordinationDelegate removeCoordinationDelegates getCoordinationDelegateStatus	String id String id[] String id	Boolean Boolean[] String	Provides control facilities over the choreography runtime for BPEL-based extensible service execution. It provides a method to “undeploy” (remove) a coordination delegate according to its identifier. Both methods return Boolean(s) to assess the success of the operation. A third method is defined to get the current status of a Coordination Delegate in the runtime based on its identifier.

Integration Dependencies

Interface Name	External Component
Delegates Generator	Coordination Delegates Generator (Development Environment)
Deploy Service	Distributed Service Bus

4.2. SCA-based XSC

The *SCA-based XSC* subsystem (see Figure 4-3) enables the implementation of Coordination delegates using SCA. Through this implementation, this subsystem provides support for a lightweight deployment and extensible enactment of choreographies composing both (Web) Business and Thing-based services.

Although our SCA runtime provides direct facilities (SCA *bindings*) to connect to Business and Thing-based services, it can also leverage other parts of the CHOReOS Middleware, namely the *eXtensible Service Discovery* (see Section 5) and the eXtensible Service Access (see Section 6) subsystems. Indeed, bindings for accessing the *Distributed Service Bus (DSB)* for Business Services and *Light Service Bus (LSB)* for Things will be implemented and will extend the scope of targets accessible to the runtime. These dependencies are detailed hereafter.

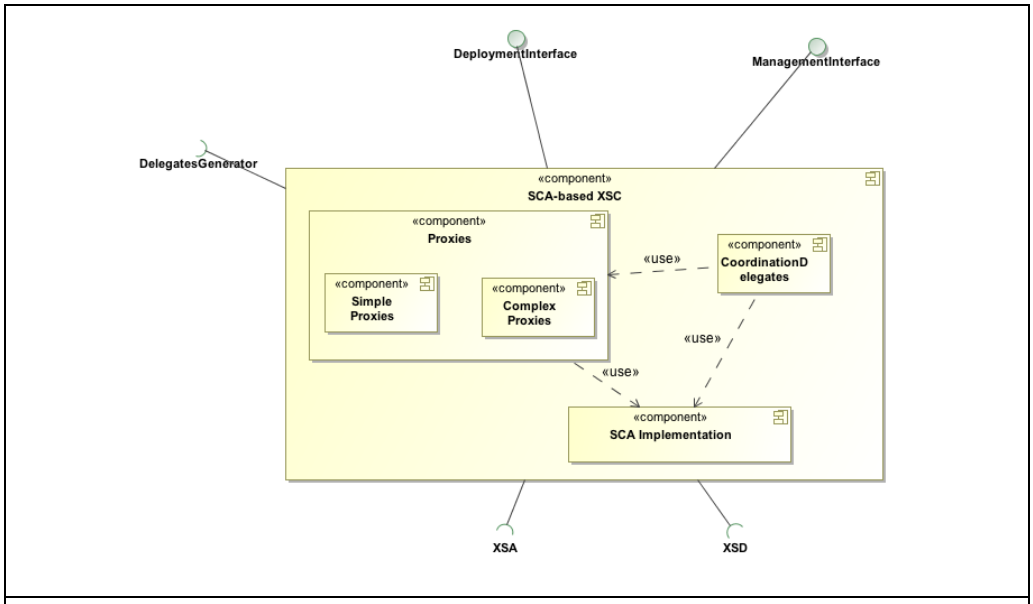


Figure 4-3. Architecture of the SCA-based XSC

Subcomponents

- **Coordination delegate implementation artefacts**: Coordination Delegates (as provided to the middleware by WP2, according to choreography specifications) are implemented in the SCA-based XSC using SCA artefacts (components, composites and bindings).
- **Proxies:** they are CHOReOS specific and rely on SCA bindings to link to actual Business and Thing-based services at runtime. *Simple proxies* are used to alleviate access mechanism and protocol heterogeneity constraints from the Business logic. *Complex proxies* are needed when dealing with protocol discrepancies.
- **SCA Implementation**: SCA is an industry standard created by major software vendors including IBM and Oracle. It provides a model for composing applications that follow Service-Oriented Architecture principles. The specific implementation used in the CHOReOS runtime is OW2 FraSCAti[1].

[1] https://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti

Interfaces and Functionality

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
DeploymentInterface	deployCoordinationDelegate deploycoordinationDelegates	CoordinationDelegate cd CoordinationDelegate[] cds	String String[]	Provides the first access point to deploy Coordination Delegates into the runtime. To this end, it includes a first method to deploy a single Coordination Delegate and a second one to batch deploy multiple Coordination Delegates at the same time. Both these methods return a single (or multiple) deployment identifier(s) that uniquely identifies(y) each deployed delegate.
ManagementInterface	removeCoordinationDelegate removeCoordinationDelegates getCoordinationDelegateStatus	String id String id[] String id	Boolean Boolean[] String	Provides control facilities over the choreography runtime for SCA. In its current state, it provides a method to "undeploy" (remove) a coordination delegate according to its identifier. Another method to do so by batch is also defined. Both methods return Boolean(s) to assess the success of the operation (if *false*, then unsuccessful). Finally a third method is defined to get the current status of a Coordination Delegate in the runtime based on its identifier. Status is a String whose values currently include *"deployed"*, *"undeployed"*, *"error"*, *"unknown"*. It may evolve to provide monitoring features such as methods returning data according to identified QoS dimensions.

Integration Dependencies

Interface Name	External Component
DelegatesGenerator	Coordination Delegates Generator (CHOReOS Development environment)
XSD	eXtensible Service Discovery
XSA	eXtensible Service Access

4.3. Things Composition and Estimation

The main responsibility of the *Things Composition and Estimation* - *C&E* subsystem (see Figure 4-4) is to support the deployment and enactment of CHOReOS choreographies in the context of the Internet of Thing-based Services. The *C&E* component accepts sensing/actuation requests from the outside world and returns their results (see Deliverable D3.1 for detail). It is in charge of coordinating the entire composition process, by relying heavily on semantic information stored in the *Knowledge Base* (see Section 5.6). In addition, *C&E* interacts with *Things Discovery* (see Section 5.3) in order to gain information regarding which Things are available within the current network topology.

Subcomponents

- **Expansion:** This expands the initial query by replacing each term in the query with an equivalent expression, found by traversing the *Domain Ontology* in the *Knowledge Base*.
- **Mapping:** This takes all dataflows produced by the expansion step and maps them to the actual network topology by interacting with the discovery interface of *XSD* and specifically the things discovery functionality. It also interacts with the *Device Ontology* in the *Knowledge Base* that models real world devices, to complement any information found to be missing during the discovery process.
- **Optimal mapping selection:** Once all feasible dataflows have been mapped (in the mapping phase), *C&E* must choose one dataflow to enact. Here, therefore, a dataflow that is (in some predefined way) optimal is found and passed on to the execution block, below.
- **Execution:** Now that the best composition of services has been determined for the query, it can be deployed using relevant components of the *XSC*, i.e., *BPEL- or SCA-based XSC*. Then, in the execution step, the services are actually accessed using the *LSB*, see Section 6.4) and the result is returned (or stored).

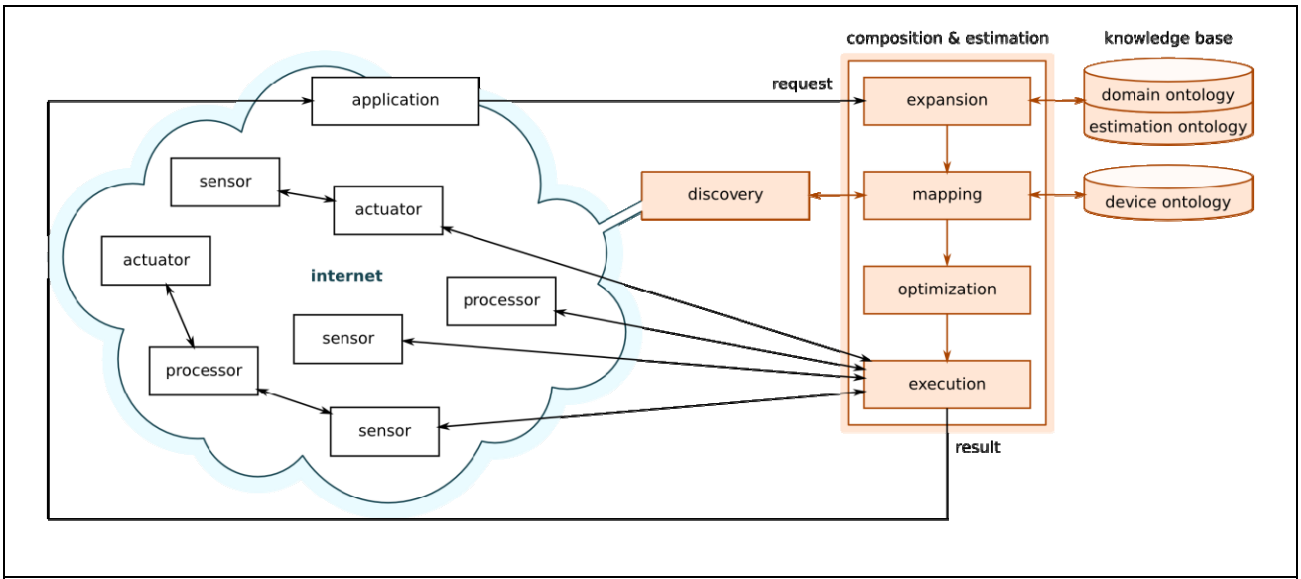


Figure 4-4. Internal Structure of Composition and Estimation component, and IoT Knowledge Base

Interfaces and Functionality

A preliminary API for the C&E component is as follows:

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ThingRequestExecutor	ExecuteRequest	Request (String, format to be detailed)	RequestResult (format to be detailed)	Executes a request for sensing/Actuating

Integration Dependencies

Interface Name	External Component
ThingsDiscovery	Things Discovery
KnowledgeBaseQuery	Knowledge Base
DeploymentInterface	BPEL- / SCA-based XSC

4.4. Reconfiguration Management for Service Substitution

Although the specification of the reconfiguration support for service substitution is going to be delivered in M24, we provide in Figure 4-5 a preliminary view of the architecture of this subsystem to facilitate the integration of the CHOReOS IDRE. The *Reconfiguration Manager* component is responsible for substituting choreographed services that no longer satisfy the choreography requirements. We assume that the services belong in specific functional/non-functional abstractions that were used to develop the choreography.

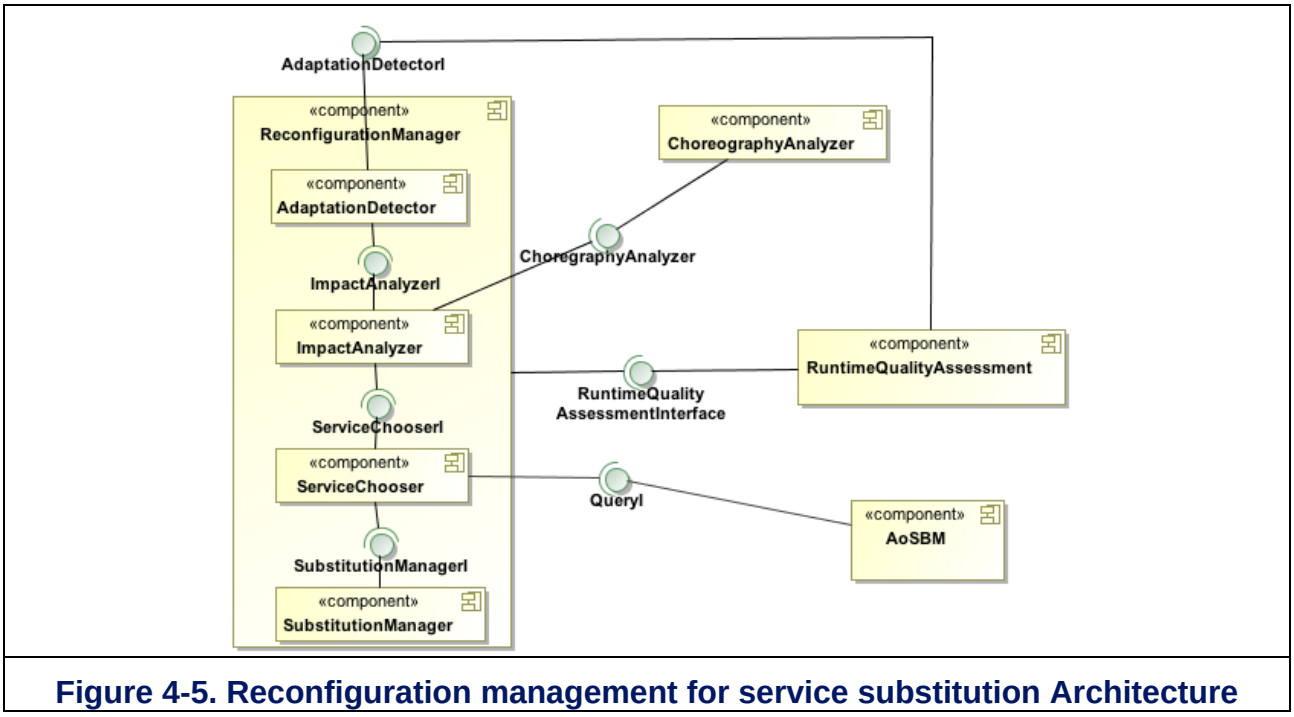


Figure 4-5. Reconfiguration management for service substitution Architecture

Subcomponents

- **Adaptation Detector:** this component detects the need to substitute a service with another one and notifies the *Impact Analyzer* component. The detection involves the *RuntimeQualityAssessment* component of the CHOReOS *Governance Infrastructure*. Hence, we have an intra-dependency with the *Impact Analyzer* that does not introduce a major integration issue. Moreover, we have an interdependency with the *RuntimeQualityAssessment* component that is a critical integration point.
- **Impact Analyzer:** this component finds choreography entities that may be affected by the change and performs appropriate actions (e.g., notify, block, etc.). Following, the component notifies the *Service Chooser* component. Hence, we have an intra-dependency between the aforementioned components, which is not a major integration issue. Finding the affected entities may involve the *ChoreographyAnalyser* component of WP2. Thus, this is an inter-dependency that may be a critical integration point.
- **Service Chooser:** This component finds a substitute service using the *AoSBM* component of the CHOReOS *Service Discovery* subsystem. Following, it notifies the *Substitution Manager* component that actually makes the change. Therefore, here we have an intra-dependency that does not introduce a major integration issue. On the other hand, we have an inter dependency with the CHOReOS *AoSBM* that is a critical integration point.
- **Service Substitution Manager:** This component actually substitutes the service with the substitute identified by the *Service Chooser*.

Interfaces and Functionality

Provided Interface Name	Operations List	Input Parameters	Output Parameters	Responsibility
AdaptationDetector	SubstituteService	sr:SubstitutionRequest	void	This method is called when there is a need to substitute a service for another one. The substitution request contains the necessary information about the change situation, the target service, the abstractions that represent it, the choreography, etc.
ImpactAnalyzer	AnalyzeImpact	sr:SubstitutionRequest	void	This method is called to analyse the impact of a service substitution
ServiceChooser	ChooseService	sr:SubstitutionRequest	substitute:Service	This method serves for choosing a substitute service using the ServiceBaseManagement component
SubstitutionManager	SubstituteService	sr:SubstitutionRequest, substitute:Service	void	This method performs the actual substitution

Integration Dependencies

Interface Name	External Component
Runtime Quality Assessment Interface	Runtime Quality Assessment
QueryI	AoSBM
ChoreographyAnalyser	ChoreographyAnalyser

5. eXtensible Service Discovery (XSD)

The *eXtensible Service Discovery – XSD* subsystem provides a unified solution to CHOReOS service discovery in the Internet of Services at large, i.e., comprising the Internet of Business Services (IoBS) and the one of Things-based Services (IoTS). The *XSD* architecture is depicted in Figure 5-1. The *XSD API* comprises a *Discovery* interface and a *Federation* interface. Through the *Discovery* interface, both Business Services and Thing-based Services can be registered and looked up. Moreover, services from heterogeneous environments can be registered via the extensible internal architecture of XSD based on *discovery protocol plug-ins*. Finally, multiple instances of *XSD* can be integrated via their *Federation* interfaces, hence providing an extensible distributed collaborative architecture that ensures scalability in the ultra large scale FI.

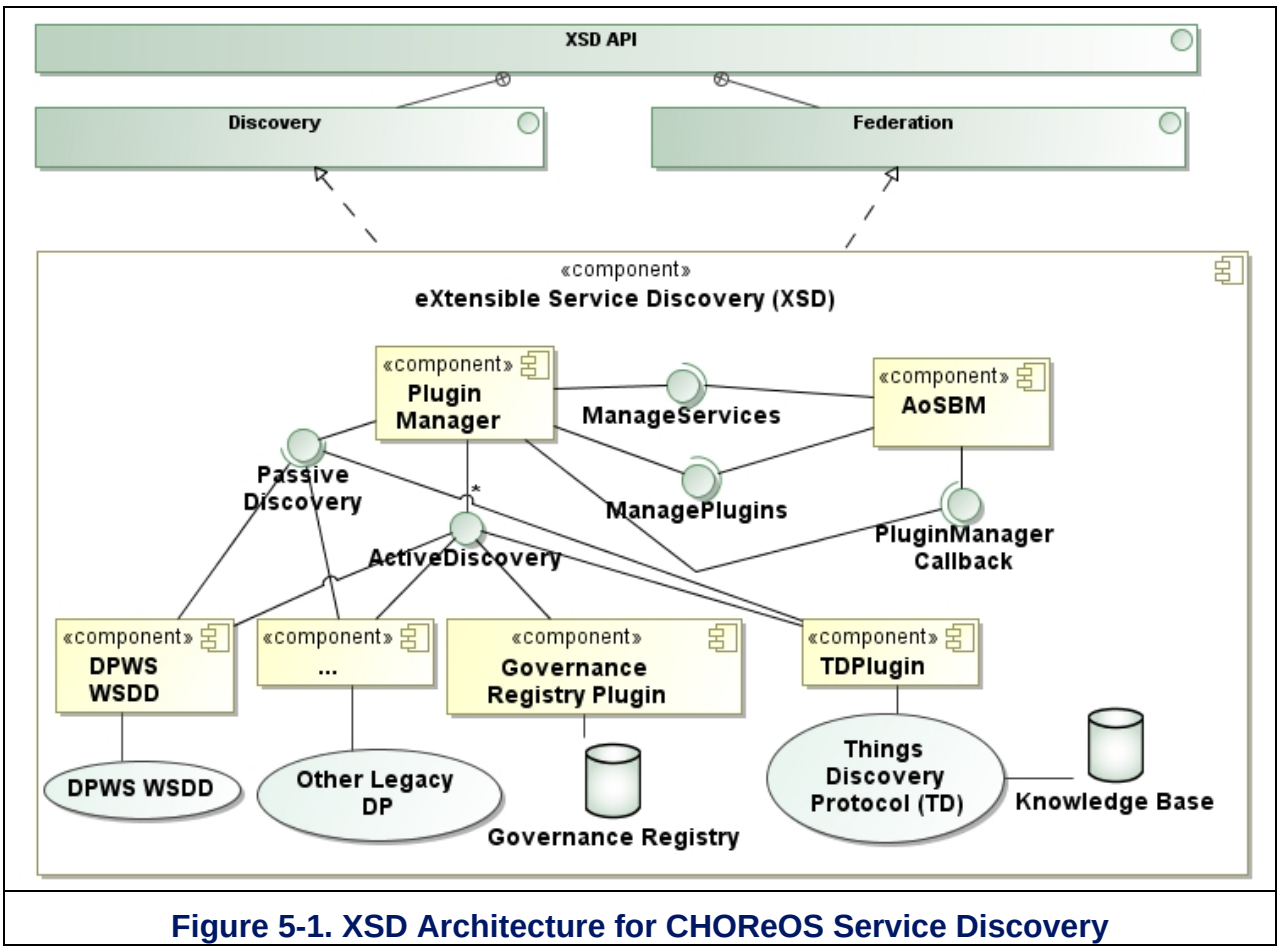


Figure 5-1. XSD Architecture for CHOReOS Service Discovery

Subcomponents

The main subcomponents of XSD are: *Abstraction-oriented Service Base Management (AoSBM), Plugin Manager, Plug-ins, Governance Registry, Things Discovery (TD), Things Discovery Protocol, Knowledge Base.* Those components are further defined in the following sections.

Interfaces and Functionality

XSD supports the traditional CRUD operations (Create, Read, Update and Delete) of persistent storage. The operations that modify the stored service-related information (i.e., Create, Update and Delete) are performed through the *ExecuteRegistrationQuery* command, which returns an acknowledgement from the repository. Meanwhile, the remaining operation (*Read*) is performed through the *ExecuteLookupQuery* command, which returns service-related information and metadata. The operations provided by *XSD* are presented in more detail in the table below.

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
Discovery	ExecuteLookupQuery	ReadQuery (format not defined)	QueryResult (service, metadata)	Sends a lookup query, returns metadata about discovered services
	ExecuteRegistrationQuery	Create, Update or Delete Query (format not defined)	Service Repository Acknowledgement	Creates, updates or removes service information and metadata upon service registration; returns an ack message from the service repository.
Federation	To be defined.	To be defined.	To be defined.	Enables federation of multiple XSD instances.

The operations of the Discovery interface are generic registration and lookup operations, which are refined according to the particular type of service/information that is registered or looked up. For these refined operations, see *Abstraction-oriented Service Base Management* (*AoSBM*) (Section 5.1), *Governance Registry* (Section 5.2), *Things Discovery* (Section 5.3).

Integration Dependencies

None.

5.1. Abstraction-Oriented Service Base Management

The *Abstraction-Oriented Service Base Management* (*AoSBM*) component (see Figure 5-2) provides functionality that allows (see Deliverable D2.1 for further details):

- The *retrieval of information* about available services from multiple sources.
- The *Organization* of the information about services with respect to functional and non-functional abstractions.
- The *Browsing* and *Querying* for services.

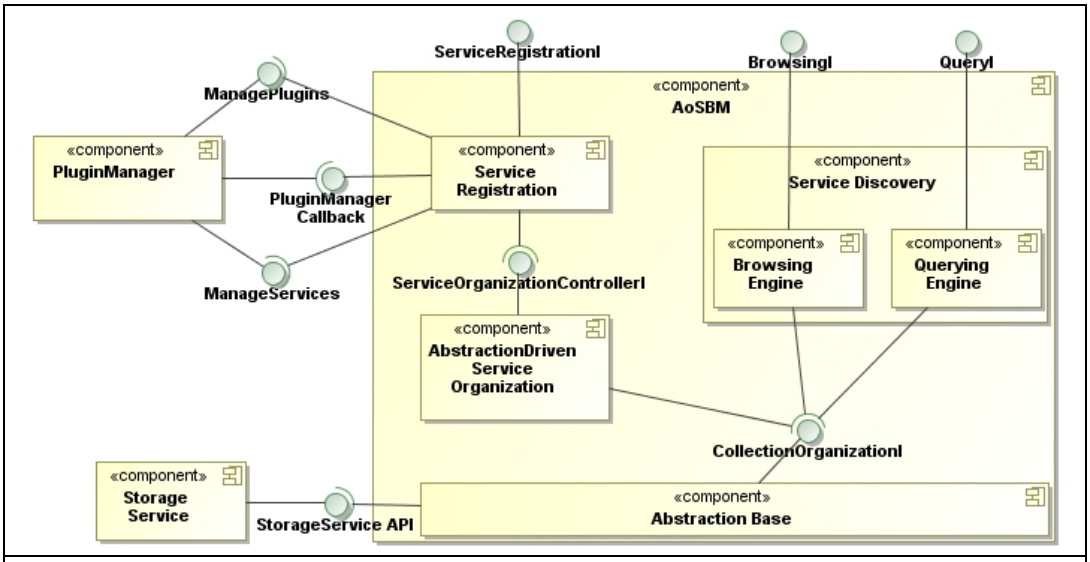


Figure 5-2. Abstraction-Oriented Service Base Management

Subcomponents

- **Service Registration:** The main responsibilities of this component relate to the Creation/Refreshing of service collections via the Plugin Manager component; the collections are stored in the Abstractions Base component, which is discussed below. Based on the above, the Service Registration subsystem has an inter-dependency with the Plugin Manager component, which constitutes and integration issue and certain intra-dependencies with other AoSBM components that are not significant for the integration.
- **Abstraction Driven Service Organization:** This component organizes services information with respect to functional/non-functional abstractions that are consequently stored using the Abstraction Base component. This component contains an intra-dependency with the Abstractions Base component and does not introduce any major integration issue.
- **Service Discovery:** This component is responsible for abstraction-driven browsing and querying, realized respectively by the Browsing Engine and the Querying Engine components. It exploits the contents of the Abstraction Base component. Thus, we have only an intra dependency with Abstraction Base that is not a major integration issue.
- **Abstraction Base:** This component stores information about services and service abstractions in a storage unit. Initially this storage unit may be centralized. Nevertheless, we further foresee to possibility to exploit the Cloud infrastructure as a storage media. In this case there may be an inter dependency between the component and the *Cloud Storage Service*, which is a critical integration point.

Interfaces and Functionality

Provided Interface Name	Operations List	Input Parameters	Output Parameters	Responsibility
ServiceRegistrationI	ExecuteRegistrationQuery	s:Service, sdpID:int	void	Registers a service to an SDP
	Produce	mode:int col:ServiceCollection	void	This method accepts as input the set of the services of the collection. Moreover, the method accepts as input the value of the mode in which the services of the collection will be organized. In any case, the ultimate goal of the execution process of the method is to organize the services of the collection into functional and non-functional hierarchies of abstractions.
	Update	mode:int col:ServiceCollection	void	This method accepts as input the necessary information about the service collection. The method follows alternative execution processes depending on the value of the mode. In any case, the ultimate goal of the execution process of the method is to update the existing functional and non-functional hierarchies of abstractions that organize the services of the collection.
BrowsingI	LaunchBrowser	void	void	Launches a browser that allows to browse abstraction hierarchies
	KillBrowser	void	void	Kills a browser
	HandleBrowsingEvent	e: Event	void	Handles a browsing event
QueryI	ExecuteLookuoQueryQuery	qex: QueryExpression	fas:FunctionalAbstraction [0..*] nfa:NonFunctionalAbstraction[0..*] s:Service[0..*]	This method accepts as input a query expression and consists of: • a service collection, • an (optional) expression over the properties of the service type and interface of the services of the collection, • an (optional) expression over the properties of the services of the collection, • an (optional) expression involving a list of functional or non-functional abstractions, annotated with an execution tag that restructures the result in groups,

				• an (optional) list of features (of the service type/interface or the service) that we want to see exported as part of the answer.
PluginManagerCallback	Notify	s:Service[0..*]		Enables communication with the *Plugin Manager*

Integration dependencies

Interface Name	External Component Providing the interface
ManageServices	Plugin Manager
ManagePlugins	Plugin Manager
Storage Service API	Storage Service

5.2. Governance Registry for Business Services

The *Governance Registry* for Business Services eases the retrieval and querying of Business services throughout their lifecycle, from design- to run-time. The functionalities offered by the *Governance Registry* refine the *XSD Discovery* interface. During the service lifecycle, governance activities such as V&V, Test-Driven Development and QoS runtime assessment are supported in the registry. In order to fulfil this, the governance registry interacts with the V&V components, the TDD framework and the SLA and lifecycle manager presented in Section 8. The whole governance functionality is fully addressed by the CHOReOS Governance Framework presented in Deliverable D4.1.

In Figure 5-3, we illustrate the *Governance Registry* for Business Services and its dependencies with the other middleware components. Via the associated plugin, services of the *Governance Registry* populate the *Abstraction-oriented Service Base*.

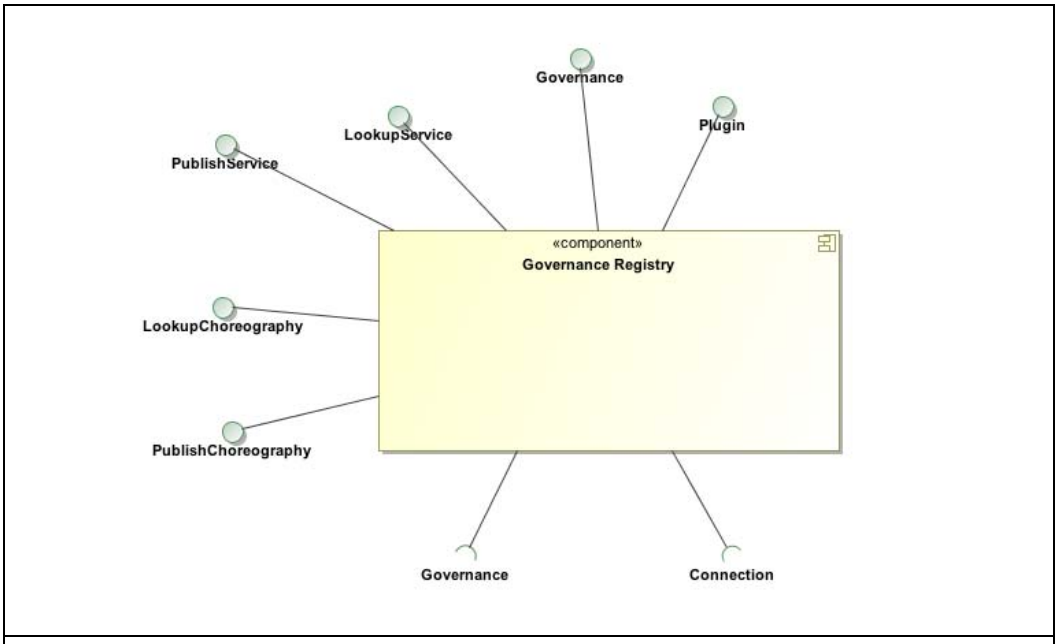


Figure 5-3. Architecture of the Governance Registry

Subcomponents

As discussed above, the *Governance Registry* for Business Services enables service and choreography discovery and publication. Moreover, it supports TDD, V&V and SLA and lifecycle management. The latter includes the SLA creation and negotiation and finally the runtime quality evaluation. The *Governance Registry* exposes the interfaces detailed in the following section, except for the Governance Interface that is detailed in Section 8.

The Governance Registry depends on the Connection Interface provided by the DSB in order to get synchronized with the distributed service bus. Indeed, the Distributed Service Bus internally relies on a naming service where the relevant data about services is stored in a synthetic way. Services location, their identifications and their names are stored in a unique lightweight registry. The naming service on top of the bus is populated automatically at runtime. It relies on the Java Business Integration specification and Application Programming Interfaces that are dedicated for handling service discovery within the ESB middleware. More advanced functionality is also supported such as: endpoint registration, endpoints and data query and components logging.

Interfaces and Functionality

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
PublishService	ExecuteRegistrationQuery	Service		The PublishService Interface provides the functionality of publishing a service.
LookupService	ExecuteLookupQuery	String serviceName	String serviceIdentifier	The LookupService Interface provides the functionality of querying a service.
PublishChoreography	publishChoreography	URI choreography		The PublishChoreography Interface provides the functionality of publishing a choreography specification.
LookupChoreography	lookupchoreography	ChoreographyIdentifier cIdentifier	URI choreography	The LookupChoreography Interface provides the functionality of publishing a choreography specification.

Integration Dependencies

Interface Name	External Component
Governance	SLA and Lifecycle Manager V&V Components
Connection	Distributed Service Bus

5.3. Things Discovery and Things Discovery Protocol

Things Discovery is the part of XSD functionality that concerns Thing-based Services (see Figure 5-1). It relates closely to the *Things Discovery Protocol*, which can be used to register and look up Things.

For instance, Things employ the *Things Discovery Protocol* whenever they register themselves with the networking environment. This protocol is further employed (via a dedicated discovery protocol plug-in) for populating the *Abstraction-oriented Service Base* with Thing-based Services. Then, in the process of resolving a sensing/actuation request, the *Composition & Estimation module (C&E)* (see Section 4.3) interacts with *Things Discovery* to determine which Things are best suited to answer the query at hand. Hence, *Things Discovery* and *Things Discovery Protocol* offer identical interfaces for registering and looking up Things. These register and lookup actions are assisted by semantic information stored in the Knowledge Base (see Section 5.6).

When being accessed by either a Thing or by C&E, *Things Discovery* and *Things Discovery Protocol* support and refine the operations of the *XSD Discovery* interface. In particular, the refinement concerns handling not only traditional relational queries but also probabilistic queries and continuous queries (see Deliverable D3.1 on the CHOReOS middleware).

Subcomponents

None.

Interfaces and Functionality

The operations provided by Things Discovery and Things Discovery Protocol are listed below.

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ThingsDiscovery, ThingsDiscoveryProtocol	ExecuteLookupQuery	ReadQuery (format not defined)	QueryResult (service, metadata)	Sends a (discrete/continuous) lookup query to a probabilistic service repository, returns metadata about discovered devices
	ExecuteRegistrationQuery	Create, Update or Delete Query (format not defined)	Service Repository Acknowledgement	Creates, updates or removes service information and metadata upon probabilistic service registration, returns an ack message from the service repository.

Integration Dependencies

As already pointed out, Things Discovery and Things Discovery Protocol make use of the semantic Knowledge Base of IoT concepts while performing their actions.

Interface Name	External Component
KnowledgeBaseQuery	Knowledge Base

5.4. Plugin Manager

The *Plugin Manager* provides unified access to service discovery protocols present in the CHOReOS environment and acts as a bridge between them. Via the *Plugin Manager*, the *Abstraction-oriented Service Base* gets populated with services discovered by these heterogeneous service discovery protocols. The *Plugin Manager* interfaces with these protocols via dedicated *plug-ins*. Individual discovery protocols can be integrated in the *XSD* runtime system by implementing the interfaces expected from a discovery *plug-in*.

Subcomponents

None

Interfaces and Functionality

The Plug-in Manager provides the following main interfaces and functionalities[2].

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ManagePlugins	AddPlugin	Plug-in information	Boolean status	Registers a *Plug-in* with the *Plugin Manager*.
PassiveDiscovery	RegisterService	ServiceDecription	Boolean status	*Plug-ins* can call this to deliver descriptions of individual services
ManageServices	RetrieveService	TargetDiscoveryEnvironment	ServiceDecription	The *Abstraction-oriented Service Base Management (AoSBM)* can call this to actively retrieve descriptions of individual services from the target discovery environment
	RetrieveServiceCallback	TargetDiscoveryEnvironment, CallbackReference, Reference to ServiceDescription store	Boolean status	The *Abstraction-oriented Service Base Management (AoSBM)* can call this to set up a callback for delivery by the *Plug-in Manager* of new service descriptions from the target discovery environment when these are available

Integration Dependencies

Active plug-ins retrieve new service descriptions upon demand of the *Plugin Manager*. Furthermore, after *RetrieveServiceCallback* is called by the *AoSBM*, the *Plugin Manager* calls *AoSBM's* callback when there are new service descriptions available. Hence, we have the following dependencies:

Interface Name	External Component
PluginManagerCallback	Abstraction-oriented Service Base Management.
ActiveDiscovery	Plug-in

[2] See Deliverable D3.1 for a more detailed interface definition of the *Plugin Manager* as outcome of previous research work currently being extended in CHOReOS. We opt for providing here only the essential functionality as the new specification is not yet final.

5.5. Plug-in

Plug-ins enable access to diverse service discovery protocols. *Plug-ins* may be active, i.e., they retrieve new service descriptions upon demand of the *Plugin Manager*, or passive, i.e., they deliver service descriptions to the *Plugin Manager* when these are available in the networking environment. As already pointed out, individual discovery protocols can be integrated in the *XSD* runtime system by implementing the interfaces expected from a discovery *Plug-in*.

Subcomponents

None.

Interfaces and Functionality

An active plug-in provides the following interface and operation[3].

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ActiveDiscovery	GetService	Details of services	Boolean status	The *Plugin Manager* can call this to retrieve descriptions of individual services

Integration Dependencies

As already discussed, passive plug-ins deliver service descriptions to the Plug-in Manager when these are available, which creates the following dependency:

Interface Name	External Component Providing the Interface
PassiveDiscovery	Plugin Manager

[3] See [D3.1] for a more detailed interface definition of a plug-in as outcome of previous research work currently being extended in CHOReOS. We opt for providing here only the essential functionality, as the new specification is not yet final.

5.6. Knowledge Base

The *Knowledge Base* is a component comprising a global ontology that represents the IoT-based real world and related querying and reasoning mechanisms. The global ontology consists of a set of ontologies that model devices and their functionalities, along with the related domain of physics. Physics are at the core of the real world representation, as they allow the approximation and estimation of functionalities provided by Things. More specifically, the global ontology includes three sub-ontologies:

- **Device Ontology:** The *Device Ontology* models Things, which are devices that may exist in the underlying network.
- **Domain Ontology:** The *Domain ontology* is a Physics and Mathematics Ontology that models information regarding real world physical concepts and their relationships.
- **Estimation Ontology:** The *Estimation Ontology* describes estimation models ("linear interpolation", "Kalman filter", "naive Bayesian learning", etc.), the equations that represent them, the services that implement them, etc.

Subcomponents

None.

Interfaces and Functionality

The operations provided by the Knowledge Base are listed below.

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
KnowledgeBaseQuery	AccessOntology	• *AccessRequestType:* Determines access to the ontology. An access can be Read, Update, or Delete • *Information:* Information to access. This maps, at a lower level, to ontology triples, i.e., {subject, predicate, object}. A subject/object is an ontology concept or instance. A predicate is a relationship. • *Condition:* A set of conditions that the information to access should satisfy. A condition has an attribute, an operator and a value. A condition also maps to an ontology triple.	*AccessResult:* The result of the query to the *Knowledge Base*	Executes a query to the *Knowledge Base*

Integration Dependencies

None.

6. eXtensible Service Access (XSA)

The *eXtensible Service Access - XSA* provides a unified solution to CHOReOS access in the Internet of Business Services (IoBS) and Internet of Things-based Services (IoTS) domains. The *XSA* architecture is depicted in Figure 6-1. It relies on the service bus paradigm.

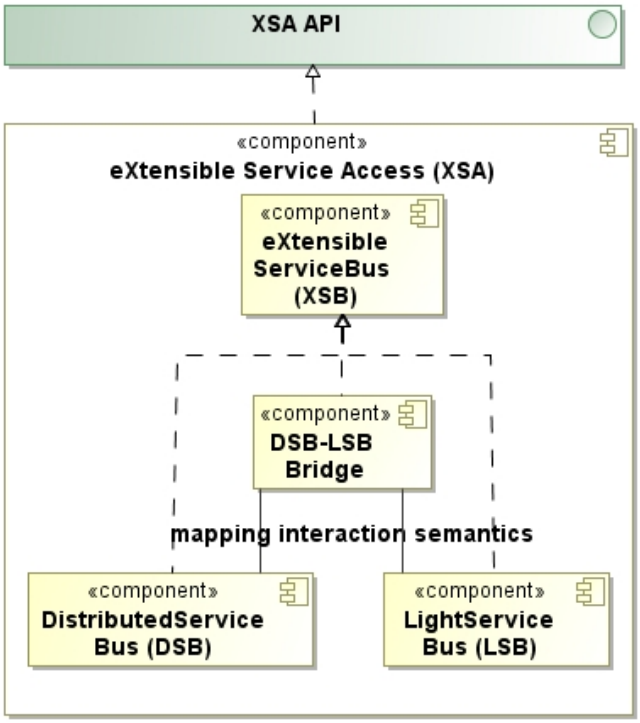


Figure 6-1. XSA Architecture for CHOReOS Service Access

Subcomponents

- **eXtensible Service Bus (XSB):** This is an abstract component that is refined by a number of derived components inside the XSA architecture. XSB prescribes the high-level semantics of a new service bus protocol that provides special support for the seamless integration of heterogeneous interaction paradigms;
- **Distributed Service Bus (DSB)** and **Lightweight Service Bus (LSB):** These are concrete realizations of XSB; and
- **DSB-LSB Bridge**: It also inherits from XSB and provides bridging between the two aforementioned concrete buses.

Interfaces and Functionality

The *XSA API* essentially comprises a *Communication interface*, through which services can interact by employing a service bus, and a *Deployment interface*, through which services can be deployed on the bus.

We opt for keeping these interfaces abstract at this level. These interfaces are refined by *XSA* subcomponents as discussed in the following sections. In particular, the *Communication interface* includes the *GA interface* for interoperable communication, which is presented in Section 6.1.

Integration Dependencies

XSA depends on the *Cloud & Grid Middleware* component in the case that certain *XSA* service bus resources are provided by the latter component.

Interface Name	External Component
NodePoolManager	Cloud & Grid Middleware

6.1. eXtensible Service Bus

The purpose of the *eXtensible Service Bus – XSB* is to enable multi-protocol access to both Business Services of the IoBS domain and Thing-based Services of the IoTS domain, as well as cross-domain access. In particular, it enables interoperability among heterogeneous interaction paradigms of both domains, while conserving as much as possible their semantics.

XSB is an abstract bus that prescribes only the high-level semantics of the common bus protocol. This semantics follows the GA abstraction paradigm (see Deliverable D1.3 on the CHOReOS architectural style).

A number of concrete components for CHOReOS Service Access are derived from *XSB*, as depicted in Figure 6-1.

Subcomponents

None.

Interfaces and Functionality

The operations provided by XSB supporting the GA semantics are listed below.

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
GA	Post	coupling, scope, data, lease		Produces a data in the networking environment according to the semantics set by the parameters coupling (i.e., strong, weak, very weak and any), scope (i.e., addressing scope), and lease (i.e., lifetime of the data)
	set_get	coupling, scope	scope, handle	Sets up reception resources at the middleware
	get_async	handle, callback	data	Enables asynchronous reception of multiple pieces of data
	get_sync	handle, policy, timeout	data	Executes synchronous reception of a single piece of data
	notify	callback, scope, data		Enables asynchronous reception, and is called by the middleware
	end_get_async	callback		Closes a channel for asynchronous reception
	end_set_get	handle		Closes a reception channel

Integration Dependencies

None.

6.2. Distributed Service Bus

The *Distributed Service Bus - DSB* provides support for accessing business services in the CHOReOS context. In the CHOReOS Project, we rely on the Petals DSB to ensure this functionality. In this section, we present the native DSB realization, while in the next section we present the XSB-over-DSB realization, which combines the native DSB with the additional XSB features.

In Figure 6-2, we present in a very synthetic way without delving into details the main interfaces of the *Distributed Service Bus*. The details and main concepts of Petals DSB are described in Deliverable D3.1 and in the JBI Specification.

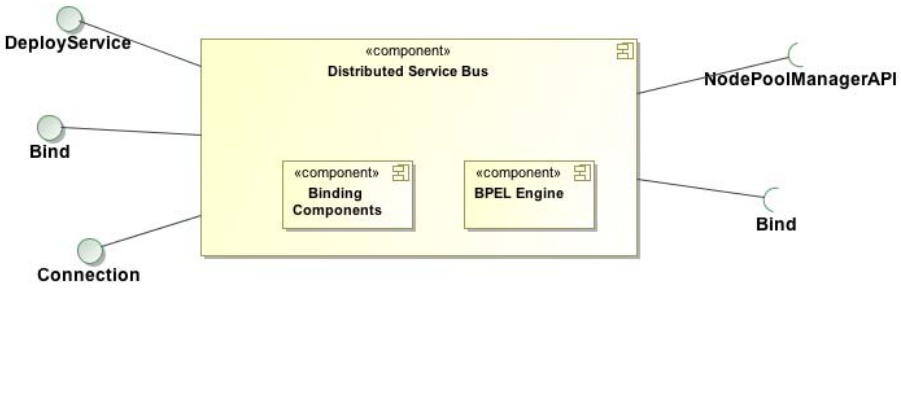


Figure 6-2. Architecture of the CHOReOS Distributed Service Bus

Subcomponents

- **Binding Components:** They are specific JBI components enabling the business services to connect from the bus to distant services deployed on other execution environment and vice versa. They are implemented as BC-SOAP components (Binding Components supporting the SOAP Protocol).
- **BPEL-Engine**. This component provides the composition and enactment capabilities for services involved in choreographies. In order to enact a choreography, the DSB can instantiate several distributed BPEL engines. The distribution concerns are handled thanks to the *Distributed Service Bus* capabilities.

Interfaces and Functionality

Provided Interface Name	Operations Name	Input Parameters	Output Parameters	Responsibility
DeployService	storeBPEL	BPELFile	-	Stores a BPEL Process
Bind	bind	ServiceEndpoint se	-	Enables distant services to bind to the bus and access others services.
Connection	connect	-	-	Uniform interface that enables the access to the DSB nodes.

Integration Dependencies

DSB depends on the CHOReOS *Cloud & Grid Middleware* in order to take benefit from the hardware resources provided by the cloud. *DSB* has the following dependency:

Interface Name	External Component
NodePoolManagerAPI	Node Pool Manager (Cloud & Grid Middleware)

6.3. XSB-over-DSB

XSB-over-DSB is a concrete bus realization of *XSB* and its GA semantics, targeting the IoBS domain, hence, enabling access to heterogeneous Business Services. In particular, it is an extended version of *DSB* (see Section 6.2), where the GA semantics is conveyed on top of the *DSB* common bus protocol.

Subcomponents

Not identified for the moment.

Interfaces and Functionality

XSB-over-DSB provides all the XSB GA operations (see Section 6.1) concerning communication. In addition, it provides all the native DSB operations (see Section 6.2) concerning deployment (i.e., plugging) of various Business Services on the bus.

Integration Dependencies

XSB-over-DSB depends on the *Cloud & Grid Middleware* component in the case that certain DSB service bus resources are provided by the latter component (see Section 7).

6.4. Light Service Bus

Light Service Bus - LSB is a lightweight concrete bus realization of *XSB* and its GA semantics, targeting the IoTS domain, hence, accounting for its dynamics and resource constraints while enabling access to heterogeneous Things. In particular, the GA semantics is conveyed on top of a substrate protocol that is suitable for the IoTS domain. As discussed in Deliverable D3.1, we are currently in the process of electing this protocol, which will most probably be DPWS (see companion of Deliverable D3.1 for our survey on DPWS).

Subcomponents

Not identified for the moment.

Interfaces and Functionality

LSB provides all the XSB GA operations (see Section 1) concerning communication. In addition, it will provide (under the conditional election of DPWS) all the native DPWS operations (see companion of Deliverable D3.1) concerning deployment (i.e., plugging) of various Things on the bus.

Integration Dependencies

None.

6.5. DSB-LSB Bridge

DSB-LSB Bridge realizes bridging between the DSB and LSB buses. As detailed in Deliverable D3.1, there are three cases requiring a bridging solution, namely, native-DSB to XSB-over-DSB, native-DSB to LSB, and XSB-over-DSB to LSB.

Subcomponents

Not identified for the moment.

Interfaces and Functionality

A *DSB-LSB bridge* does not implement any proper interface, it rather plugs into the interfaces exposed by the DSB and LSB buses.

Integration Dependencies

None.

7. CHOReOS Cloud and Grid Middleware

The *Cloud and Grid Middleware* subsystem (see Figure 7-1) provides basic services that support computational and storage intensive tasks performed either by the CHOReOS middleware subsystems, or by the choreographies that are built on top of the CHOReOS middleware. In order to avoid complexity in this integration part, the Grid Computing Interface used internally is not presented (details are in Deliverable D3.1).

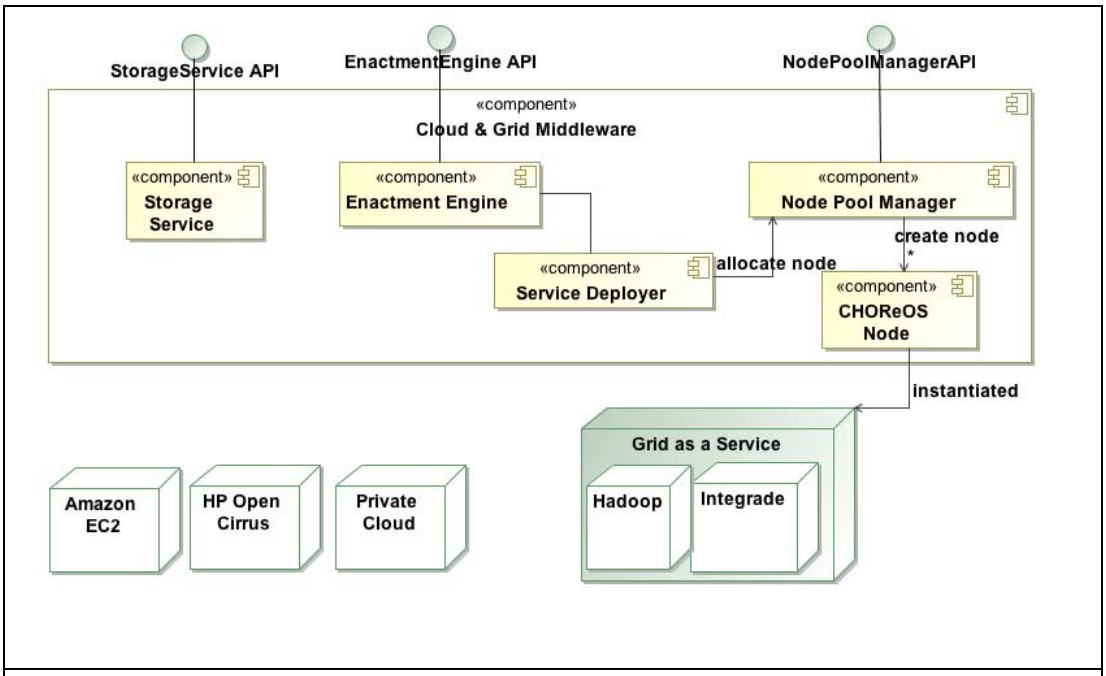


Figure 7-1. Cloud and grid architecture

Subcomponents

- **Service Deployer:** *The Service Deployer* API is defined using a RESTful API. This Service will deploy a given service to be later used by a choreography. Most deployment details are hidden from the client, such as in which machine to deploy the service and number of instances. The client may provide hints as to the expected resource consumption for each instance, and request non-mandatory increases or decreases in available processing power (which means modifying the number of instances according to an estimate).

  The *Deployer* is capable of deploying two kinds of services:

  - POWS - java web services
  - BPEL Orchestrations (not recursively; needed services must already be available)

  To deploy the service, the *Service Deployer* needs the corresponding code in executable form (e.g., war files, bpel specs, etc).

  The *Deployer* will also be able to remove (undeploy) these services.

- **Node Pool Manager:** The *Node Pool Manager* API is defined using a RESTful API. This API can be implemented using a pure HTTP service or using Java JAX-RS, which basically converts a Java class to a REST service. The main entity created by this API is the *Node*. Each node represents a logical machine created and managed by the CHOReOS Middleware. A node can be created in any cloud service supported by the middleware. This will be transparent to those that use the *Node Pool Manager* interface. Each node also has a set of configurations. Each configuration is related to a

recipe, which will install a bunch of software in this node. This installation will be provisioned by the configuration management tool used by the *Node Pool Manager*.

- **Storage Service:** Choreographies running within CHOReOS nodes may need to store, process and share large amounts of data. In order to scale up and down the number of responses and the storage space, the Cloud component will provide this service. The CHOReOS *Storage Service* will implement the Simple Storage Service implemented by Amazon, which is becoming a standard for storage services.
- **Third-party Cloud IaaS Providers:** In order to have a high-scalable infrastructure capable of responding to a large number of requests, the CHOReOS Middleware will be able to use third-party Infrastructure as a Service (IaaS) clouds. If the private infrastructure runs out of resources or is absent, the system can be deployed in public IaaS clouds. Aiming to avoid vendor lock-in, a layer is used to abstract vendor-specific features making it possible to deploy choreographies in a variety of providers. This abstraction is also important to support different software used in different private clouds.

  Operations like virtual machine start up and shut down will be made by the *Node Pool Manager* subsystem in IaaS cloud providers through an abstraction layer. Once a virtual machine is up and running, secure shell (SSH) network protocol is used regardless of the provider. Using this protocol and appropriate software, the virtual machine is configured with the necessary software and configuration to run services and play roles in the choreography. This way, it is possible to use many kinds of private, public and hybrid IaaS clouds to enact a choreography hiding the complexity of cloud-specific APIs and features from other subsystems like the *Service Deployer*.
- **The Enactment Engine:** This subsystem is responsible for setting up the choreography. For that, it needs to receive as input the following artefacts from the CHOReOS synthesis process:
  - Set of coordination delegates defining the interaction logic among the choreography services (e.g., in BPEL or jar files);
  - Services related to the coordination delegates:
    - Already existing service URIs;
    - Binary code (e.g., war, jar, aar files) of the services that are not already deployed.

  Provided the above input, the *Enactment Engine* will:
  - Instantiate the coordination delegate objects using the *Service Deployer*;
  - Deploy services that are not running using the *Service Deployer*;
  - Register services that are not in the Service Registry (the deployed binary codes);

Thus, the *Service Deployer* will make the services available through the DSB, abstracting details such as Petals Service Assembly specification. A preliminary API is defined in the next page, but this component will be developed in fact during the second year of the project.

Interfaces and Functionality

The *Cloud and Grid* components provides the following interfaces and functionality:

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ServiceDeployerAPI	POST	Service service	servicelink: URI	Deploys a new service
	GET	-	services: List<Service>	Lists deployed services
	PUT	Service service, String id, Integer factor	Status: ok, not found or error	Modifies performance of an existing service by factor
	DELETE	String id	Status: ok, not found or error	Undeploys an existing service
NodePoolManagerAPI	POST	Node node *(Node attributes are described below)*	nodelink:URI	Creates a new node
	DELETE	String id	Status: ok, not found or error	Deletes an existing node
	PUT	Node node, String id	Status: ok, not found or error	Modifies an existing node
	GET	String id	Node:Node	Gets existing node details
	POST	String nodeID, String configID	Status ok, not found or error	Deploy an existent configuration to an existing node.
	DELETE	String nodeID, String configID	Status ok or error	Undeploy a configuration in a node.

	GET /nodes/{id}/configs	String nodeID	Status ok with a list of configurations, status not found or error	Lists all configurations deployed in a node.
EnactmentEngineAPI	deployCoordinationDelegate	URI coordinationDelegate	URI deployedCD	The responsibility of this interface is to provide the first access point to deploy Coordination delegates into the runtime. Once deployed, their status can be acquired and they can also be undeployed using their unique IDs. It can also deploy and undeploy new services using a binary serviceFile.
	undeployCoordinationDelegate	String id	Boolean	
	deployService	File service	URI service	
	undeployService	String id	Boolean	
	getCoordinationDelegateStatus	String id	String	Status is a String which values currently include “deployed”, “undeployed”, “error”, “unknown”. It may evolve to provide monitoring features such as methods returning data according to identified QoS dimensions.
	getServiceStatus	String id	String	
StorageServiceAPI				This service will implement based on the API defined by Amazon S3 API.

The search criteria is of the form:

{field1}={value1}&{field2}={value2}&...&{fieldN}={valueN}
where fields are all available attributes in nodes (see Nodes attributes bellow).

The Node Attributes are as follows:

Name	Required	Type	Description
cpus	yes	Integer	Number of cpus in a node.
ram	yes	Integer	Amount of RAM memory (in MB).

storage	yes	String	Amount of allocated storage space (in GB).
hostname	no	String	The node hostname.
IP	no	String	The node IP address (if applicable).
so	yes	String	The operating system to be installed in the node. The SO is related to the bootstrap image that will be used to create the node. Nodes are, in essence, a SO installation plus a set of configurations.
zone	No	String	The zone attribute can be any String. Nodes with the same zone name will be prioritized to be created in the same cloud infrastructure.

Integration Dependencies

Interface Name	**External Component**
Discovery	eXtensible Service Discovery

CHOReOS Governance and V&V

8. Governance and V&V Framework

The CHOReOS *Governance and V&V Framework* is presented in the Work Package 4 of CHOReOS Description of Work. It gathers the functionalities responsible *for Testing Driven Development*, the *V&V*, the *SLA and Lifecycle Management* and finally the *Business Service Monitoring*.

In this section, we only present the SLA and Lifecycle Management, the V&V Functionality and the TDD Framework, while the Governance Registry and the Business Service Monitoring are assigned to separate components according to their respective concerns that are distinct but complementary from the other governance activities. The *Governance Registry* component is specifically presented as part of the *eXtensible Service Discovery* subsystem in Section 5, and the *Business Service Monitoring* component is addressed as part of the CHOReOS *Monitoring subsystem* in Section 9.

8.1. SLA and Lifecycle Manager

The *SLA and Lifecycle Manager* is responsible for offering the capabilities that ease the management of the resources that are governed. These can be a service, a choreography, a policy, or a service level agreement. Meanwhile, it manages the roles and responsibilities of the users of the *Governance Framework*, by assigning credentials. In Figure 8-1, we present the components of the *SLA and Lifecycle manager* related to the Governance Framework.

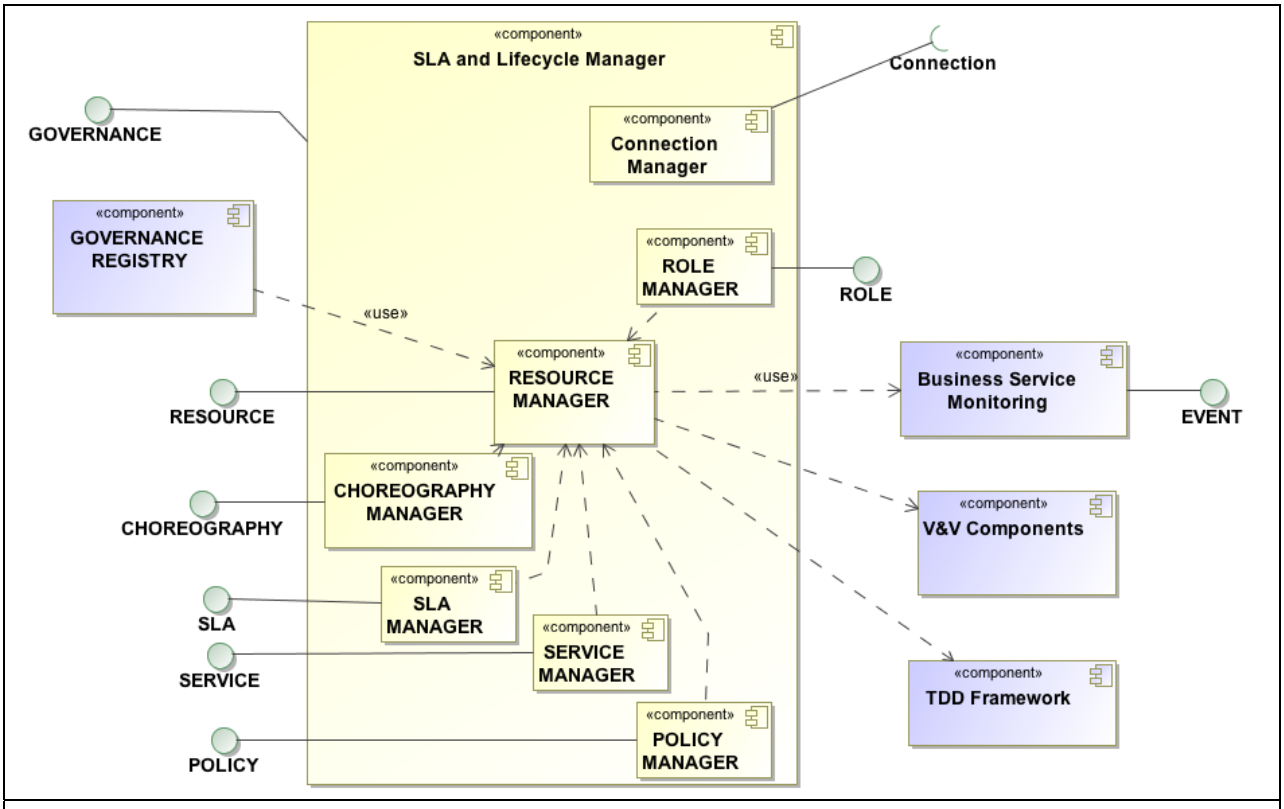


Figure 8-1. SLA and Lifecycle components

Subcomponents

- **Resource Manager:** The *Resource Manager* is a generic component that provides capabilities for the management of the lifecycle of the XML Resources. These are handled by the Governance and V&V Framework and can be a Service Description, a Choreography Specification, an SLA, or a Policy. The *Resource Manager* provides the Resource interface. The *Resource Manager* is connected to the *Business Service Monitoring* that is presented in section 9.1. It is also connected to the *Governance Registry* that provides resource discovery functionality (Section 5.2).
- **Service Manager:** The *Service Manager* is a component providing an interface called Service. The Service interface provides abilities for managing the lifecycle of a service. A service can be created, described, and finally retired.
- **SLA Manager:** The *SLA Manager* provides an SLA interface. It provides a list of functionalities facilitating the management of the lifecycle of an SLA such as the creation, reading, negotiation, lookup of the SLA templates attached to a service and finally the termination of an SLA.
- **Policy Manager:** The *Policy Manager* component provides a Policy interface. It is responsible for the management of the lifecycle of a policy. Offered functionalities are the creation, addition, storage and removal of a Policy.
- **Choreography Manager:** The *Choreography Manager* is responsible for providing the needed functionality for dealing with the governance of a choreography. This component may depend on the design tool for choreographies if provided.
- **Role Manager:** The *Role Manager* component is responsible for providing facilities for the creation, modification and deletion of the governance roles and responsibilities. It provides the Role interface.
- **Connection Manager:** The *Connection Manager* is responsible for synchronizing the Governance and V&V Framework with the middleware runtime environment. Functionalities such as the connection, disconnection, synchronization and the lookup of the available execution environments are offered. The *Connection Manager* requires the Connection Interface of the DSB (Section 6.2).

Interfaces and Functionality

Interface Name	Operations List	Input Parameters	Output Parameters	Responsibility
Resource	Publish getResource subscribe	URL address, ResourceType type String resourceId EventType event	 Document SubscribeResponse	The Resource Interface Is a generic API providing capabilities for managing any XML Resource.
Service	findServices findTechnicalInterfaces addService retireService DescribeService	FindServices parameters FindTechnicalInterfaces parameters ServiceIdentifier serviceId ServiceDescription ServiceDescription	FindServicesResponse FindTechnicalInterfacesResponse	The Service Interface provides capabilities for managing a service lifecycle.
Event	findEventProducers getEventProducer findAddressesOfEventProducers	FindEventProducersRequest body QName idEventProducer List<String> topicExpressions	FindEventProducersResponse ServiceType List<EPR>	The Event interface manages functionalities related to the events produced by the Governance framework.
SLA	create read findSLATemplate negociateSLA terminateSLA	List<SLO> String SLAIdentifier String ServiceIndentifier SLATemplate template SLATemplate template	Document Document <List> SLATemplates	The SLA interface offers functions to manage the creation, lookup, negotiation and termination of an SLA.
Connection	synchronize connectToEnvironment unconnectToEnvironment getAllEnvironment	 String endpointAddress String endpointAddress	 ExecutionEnvironmentInformationType ExecutionEnvironmentInformationType List<ExecEnvironmentInformationType>	The Connection interface facilitates the connection of the Governance framework to the execution environment.
Policy	create addPolicy	 PolicyIdentifier pIdentifier,	 Event	The Policy interface provides functionality of managing the lifecycle of a

	modifyPolicy deletePolicy	PolicyParameters PolicyIdentifier pIdentifier		Policy.
Choreography	addChoreography deleteChoregraphy publishChoreography retrieveByName retrieveByType retrieveByRole retrieveByTask unsubscribeChoreography	Choreography Specification String Choreography Specification String Choreography Type Role Specification Task Specification boolean	Boolean Boolean Boolean Choreography Specification Choreography Specification Choreography Specification Choreography Specification Boolean	The choreography interface provides the ability of dealing with the lifecycle of a choreography: publication, retrieval, and removal from the choreography registry.
Governance	getResources getService getPolicy getSLA getChoregraphy	URI choreography	List<Resource> List<Service> List<Policy> List<SLA> List<Choreography>	The Governance interface provides a generic API for accessing the Governance and V&V framework.
Role	createRole modifyRole addCredential retireCredential retireRole getRole	Role role Role role Credential c Credential c RoleIdentifier rIdentifier RoleIdentifier rIdentifier		The Role interface provides facilities for managing the governance roles.

Integration Dependencies

The SLA and Lifecycle manager depends on the following components:

External Interface Name	**External Component**
Connection	Distributed Service Bus (eXtensible Service Access)

8.2. V&V Components

The *Governance framework* provided by the CHOReOS project implements a comprehensive strategy for managing both choreographies and services. In addition, as deeply discussed in Deliverable D4.1, the project will put special emphasis on governance aspects related to choreography-oriented V&V activities by defining policies, and rules governing (e.g., enabling, regulating, etc.) them.

As presented in Deliverable D4.1, the *Governance Registry* is important component for SOA Governance. Indeed, service providers as well as choreography designers are expected to publish services and choreography descriptions on such registries. Consumers looking for services and choreographies by the role they aim to play can refine their search according to non-functional concerns: for example performance, results of the V&V sessions, usage frequency, ratings, etc.

In this chapter we present the current results achieved in WP4 by providing a high-level view of the main CHOReOS IDRE components that will be devoted to enable Governance and V&V activities. Thus, the rest of the chapter reports the preliminary design of the Governance V&V framework within the CHOReOS IDRE.

With reference to Figure 8-2, in the following we provide a preliminary description of the main components of the Governance V&V framework designed for the integration into the CHOReOS IDRE, with their offered/required interfaces. In line with what is specified in the DOW, information about component interfaces SHOULD NOT (i.e. "NOT RECOMMENDED") be considered either as complete or definitive, yet.

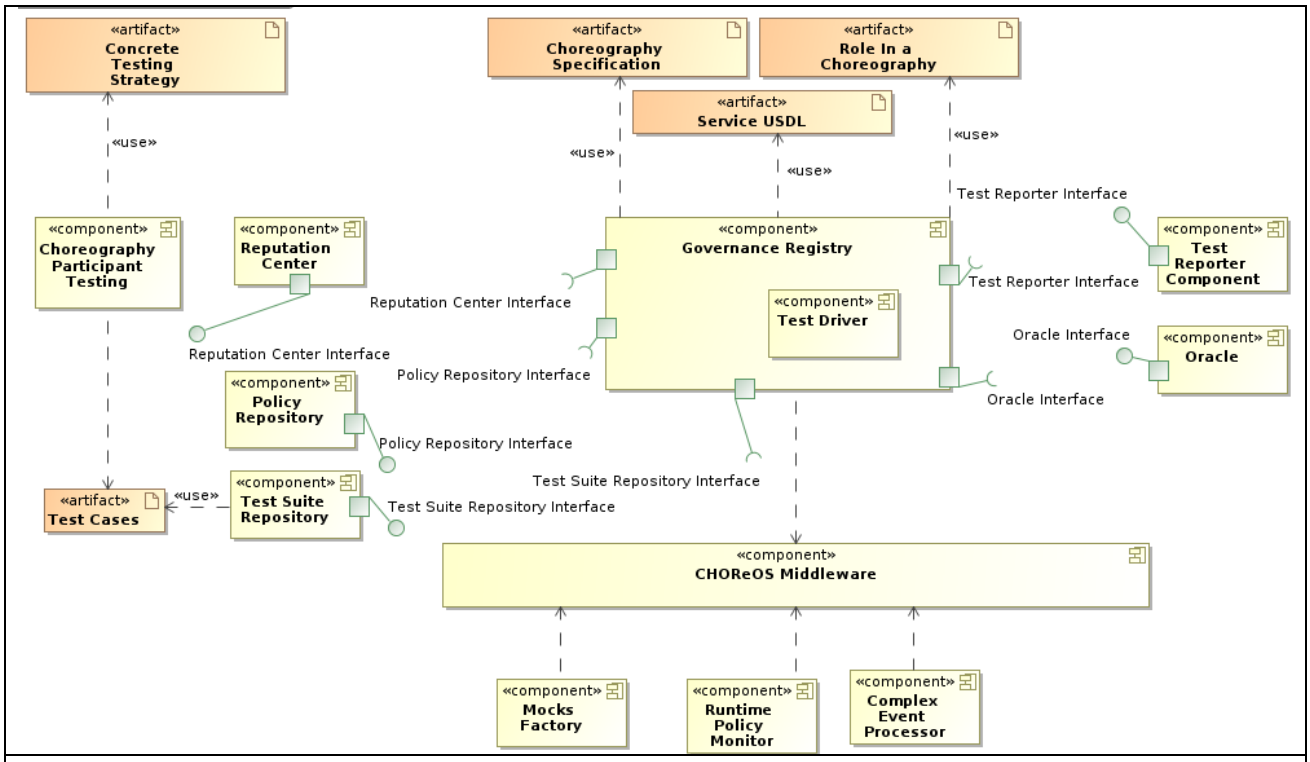


Figure 8-2. Preliminary Components of the V&V Framework

Subcomponents

The CHOReOS IDRE will enhance the service registry with testing functionality as detailed in Chapter 3 of Deliverable D4.1. The registry, which is presented in Section 5 will provide mechanisms to govern the management of installed testing handlers. Handlers are mechanisms permitting to modify a service registration procedure with additional functionalities. In particular testing handlers activate testing sessions on services for which a registration request, or a modification of the associated entry, is received.

- **Test Driver:** This component, of which several different instances will be available and dispersed over the IDRE, permits to retrieve and execute a test suite on a service. The driver is activated by the testing handler, which will provide the necessary information to identify the service to test and the test suite to execute.
- **Oracle:** This component permits to assess if the outcome of a test invocation made by the test driver is acceptable with respect to what was expected.
- **Test Reporter Component:** This component permits to store the results of launched testing sessions. Information reported can be used by governing authorities to put in place inclusion/exclusion policies for services (and their providers), based on the results. In principle only services successfully passing the testing sessions should be admitted.
- **Mocks Factory:** This component permits to derive proxies and mocks necessary to test services willing to participate to a choreography. Created proxies and mocks permit to assess also how a service is able to interact with the roles specified in the choreography.
- **Reputation Center:** This components logs information concerning reputation of services belonging to the CHOReOS Governance Framework. This information is useful to improve the service selection process and to put in place policies concerning service lifecycle activities related to those services made available by the corresponding service provider. Also, the *Reputation Center* is in charge of evaluating reputation metrics for “enactable” choreography. The evaluation is being based on the evaluations of single reputations for the possible participating services. At the time of this writing, the component *Reputation Center* needs to be further refined. Specifically, the concrete operations exported by this interface are still under design.
- **Choreography Participant Testing:** This component permits to automatically derive test cases from choreography specification. Specifically, this component derives a choreography-oriented test suite that can assess if a service can actually play a role when integrated in a given choreography.
- **Test Suites Repository:** This component permits to store and index test suites so that they can be executed to assess running services.

Interfaces and Functionality

Provided Interface Name	Operation Name	Input	Output	Responsibility
Test Driver Interface	checkService	Service Endpoint, Choreography Specification, Role	Test Result Specification	Retrieves a test suite by the role foreseen in a choreography, and it executes such test suite on the specified service
Oracle Interface	getExpectedResult	TestCase	Test Result Specification	Computes and returns an expected result for the given test case
Test Reporter Interface	notifyTestResult	Test Result Specification, Destination	boolean	Notifies the result of a test over the middleware
	storeTestResult	Test Result Specification	boolean	Stores the result of a test
	storeTestResult	Test Result ID	boolean	Stores the result of a test
Mocks Factory Interface	generateMocks	Choreography Specification, Role	Mocks EndPoint List	Generates and deploys moks for the roles specified by a choreographies. It returns a list of endpoints where the mocks are deployed.
	generateStubs	Choreography Specification, Role, Service Endpoint	Stubs EndPoint	Generates and deploys a stubs for the services playing a given roles in a choreographies. It returns the endpoint where the stub is deployed.
Choreographies Participant Testing Interface	generateTestSuites	Choreography Specification	Test Case Specification	Generates a set of Test Cases from a Choreography Specification.
Test Suites Repository Interface	storeTestSuites	List of Test Case Specification	boolean	Stores a Test suite into the repository
	retrieveTestSuites	Choreography Specification, Role	List of Test Case Specification	Retrieves the Test suites into the repository that belongs to a given role of a choregraphy

Integration Dependencies

Interface Name	External Component
Governance Registry Interfaces	Governance Registry
Middleware Interfaces	Cloud & Grid Middleware

8.3. Test-Driven Development Framework

The main goal of Rehearsal, the CHOReOS testing framework, is to support TDD (Test-Driven Development)[4],[5] of web service choreographies. Using the framework, a choreography designer or developer will be able to perform a series of tests to guide the choreography development. TDD is performed in a testing, or offline, environment where some of the concrete services may not be available. To achieve that, Rehearsal provides mechanisms for emulating real services or a part of the choreography by using mocks, which is a TDD standard practice. In addition, the framework provides mechanisms for applying unit and integration tests to create or adapt services to perform choreography roles properly. TDD is performed in a testing, or offline, environment and the tests written will serve both as an executable specification of the choreography behaviour and as a means for V&V at design time. Later, at runtime, the same tests may be used with the online system to verify the proper behaviour of the choreography in the production environment. These later tests are supported by the V&V components, which are described previously.

The internals and externals dependencies of the framework components are presented in Figure 8-3. Each component provides features for the developer to apply multiple levels of automated tests on the choreographies following the TDD principles. The Rehearsal framework components and its features with their respective APIs are presented below.

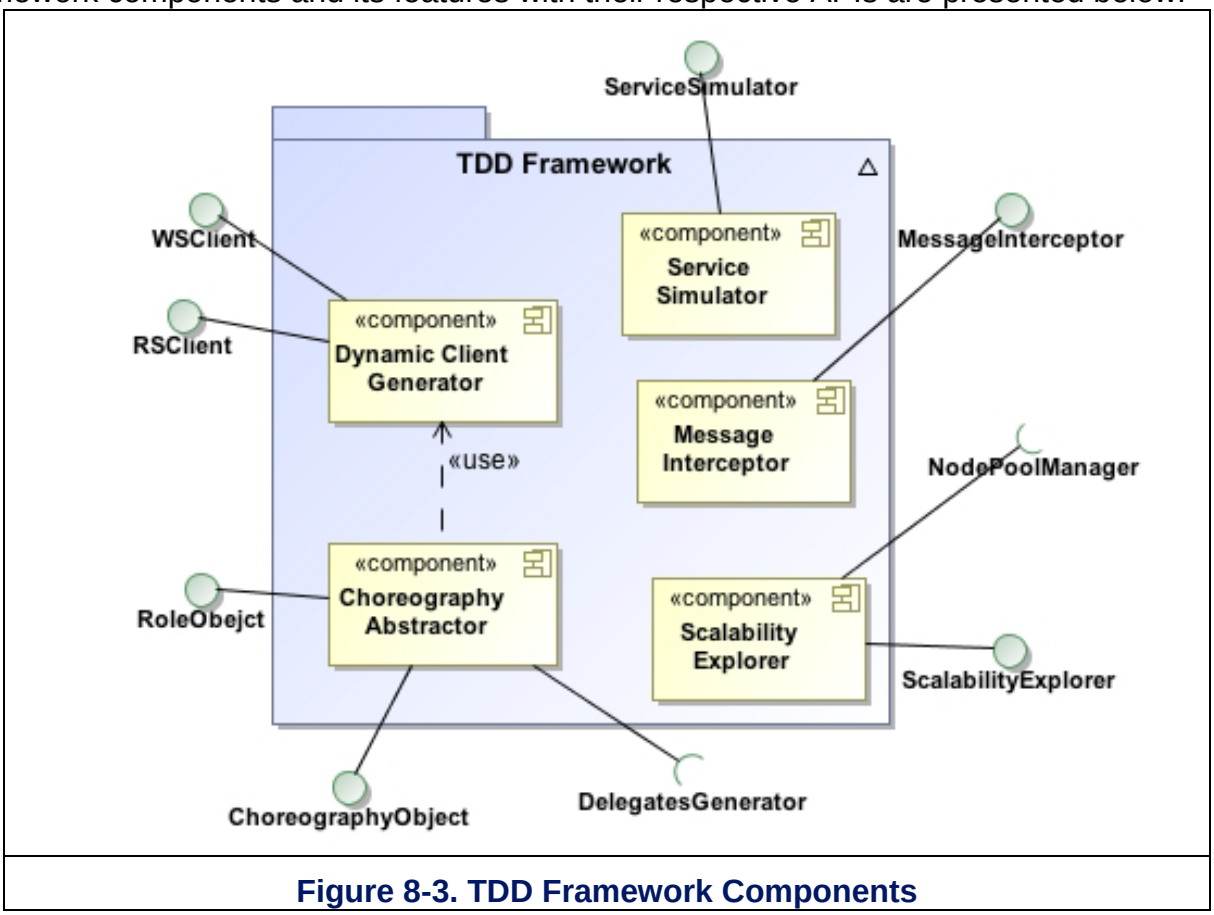


Figure 8-3. TDD Framework Components

Subcomponents

- **Dynamic client generator:** *The Dynamic Client Generator* component supports the dynamic generation of web service clients, which is a framework feature to facilitate the invocation of web services, greatly simplifying testing code. Given a web service interface (in WSDL), its operations can be invoked dynamically using the *WSClient*

[4] David Astels. Test-Driven Development: A Practical Guide. Prentice Hall, New Jersey, 2003.
[5] Kent Beck. Test-Driven Development: by example. Addison-Wesley, Boston, 2003.

class. To avoid XML manipulation, the Rehearsal framework provides the Item object. This object can be used as an operation parameter, or an operation response. To interact with RESTful services, the RSClient class provides mechanisms to apply HTTP operations on the resources of the service under test.

- **Message interceptor:** The framework supports the execution of integration tests by validating the exchange of messages among services. The framework will provide a feature for: (i) intercepting and collecting messages; and (ii) retrieving the messages collected for later validation.
- **Service simulator:** This component supports mocking of web services, providing mechanisms for mocking all operations of a service that is not available or cannot be tested offline. Also, in a development environment, the choreography may need to be adapted to replace a real service with a mock service. This component is based on the *MockFactory* component (see section 8.2).
- **Scalability explorer:** The *Scalability Explorer* component supports scalability testing by providing features to simulate the choreography and the service participants in different scales and analyse how scalable they are. To verify how scalable a choreography or a service can be, the developer needs to exercise it in different scales. These executions are compared to determine the level of scalability. Therefore, the developer has to execute the same process in different scales and compare the obtained results.

  The *@ScalabilityTest* and *@Scale* annotations assist the developer to achieve that. A method that has the *@ScalabilityTest* annotation will be executed by the framework in different scales. This is done by changing the parameters with the *@Scale* annotation for each execution. The annotations are described below:

```
@ScalabilityTest(
scalabilityFunction=<linear, exponential, quadratic, and
                    logarithmic>,
maxIncreaseTimes=<an integer telling how many times to increase
the parameters>)
```

  The Rehearsal framework will execute test scenarios multiple times, in each execution; the parameters that are specified with the *@Scala* annotation will be scaled out automatically. The function specified on *scalabilityFunction* parameter will determine how the parameters will scale during the tests execution. The possible functions are linear, exponential, quadratic, and logarithmic. The framework will increase the parameters a number of times defined on the *maxIncreaseTime* parameter.

  After running different scales of a test scenario, the framework collects the results and show how scalable the choreography is in fact.

  The code bellow describes a simple example that verifies the scalability of a service that is a calculator:

```
1.  @ScalabilityTest(scalabilityFunction="quadratic",
                     maxIncreaseTimes=5)
2.public Long testTheScalabilityOfCalculator(Service calculator,
                              WSClient serviceDeployer,
                              @Scale Integer numberRequests,
                              @Scale Integer cloudInstanceType) {
3.    serviceDeployer.put("/service/1", calculator, 1,
cloudInstanceType)
4.    WSClient wsClient = new WSClient(calculator.getWsdl());
5.    Item message = new Item();
6.    message.setContent("1+1");
7.    ReportMultipleRequests report = wsClient.multipleRequest(
numberRequests,
                                              10, "calculate",
                                              message);
8.    return report.getAverageTime();
9. }
```

Initially, the method configures the node where the Calculator service is to be located (line 3), using the *Service Deployer* component. In lines 7 and 8, it creates the content that will be sent. Then, it requests *numberRequests* times the Calculator service, with the content message. In the report object, there are information about responses and average response time, which is the value returned by the method. After that, the framework will take the *numberRequets* and *cloudInstanceType* parameters, both defined with the *@Scale* annotation, double their values, and execute the method again with these new values. This will be repeated 5 times, as defined by the annotation. Finally, the framework will take the values returned by these 5 executions and will compare them to determine the level of scalability of the service based on this process.

- **Choreography abstractor:** This component supports a framework feature for abstracting the choreography elements such as roles, services, and messages into Java objects. During test writing, these objects can be inputs for the previously described APIs. For example, the endpoints extracted from a Role instance (see below) can be dynamically invoked using WSClient or mocked using the Mocking service feature. This component interacts with the *Synthesis Processor* through the *DelegatesGenerator* API to create the objects described previously.

  Using the features presented above, Rehearsal aims at providing to the choreography developer mechanisms for applying unit, integration, conformance, acceptance, and scalability testing on services and choreographies. Further information about the framework will be provided in deliverable D4.2.

Interfaces and Functionality

Interface Provided Name	Operations List	Input	Output	Responsibility
RoleObject	Choreography <<constructor>>	bpmn2Diagram: URI	choreography: Choreography	Creates an instance of Choreography
	findRole	name:String id:int	role:Role	Retrieves a specific Role element
	Request	operationName:String arguments:(String... or Item)	response: Item	Invokes a choreography operation
Choreography object	Choreography <<constructor>>	bpmn2: BPMN2Diagrams	choreography: Choreography	Creates an instance of Choreography
	findRole	name:String id:int	role:Role	Retrieves a specific Role element
	Request	operationName:String arguments:(String... or Item)	response:Item	Invokes a choreography operation
Service simulator	Mock <<constructor>>	wsdl:String	mock:Mock	Creates an instance of Mock
	returnFor	operationName:String parameters: (String.. or Item) objectReturned:Item		Defines the object returned for the operation
	Crash	operationName:String		Configures the mock to simulate a crashing behavior
Message interceptor	Intercept	messageName:String	interceptor: Interceptor	Intercepts a specific message
	From	role:Role	interceptor: Interceptor	Intercepts messages sent by a role
	Intercept	messageName:String	interceptor: Interceptor	Intercepts a specific message
RSClient	RSClient <<constructor>>	baseURI:String basePath:String port:int	rsClient: RSClient	Creates an instance of RSClient
	HTTP operation (get, post, put or delete)	path:String parameters:Map<String, String>	String	Applies a HTTP operation over a resource
WSClient	WSClient <<constructor>	wsdl:String	wsClient: WSClient	Creates an instance of WSClient
	Request	operationName:String arguments:(String... or Item)	response: Item	Invokes a service operation
	multipleRequests	number_requests: Integer, time_interval_between_requests: Long	result: MultipleRequests Report	Requests multiple times a service operation. The MultipleRequestReport stores the

		operation: String, content: Item		different responses and the response average time.
	getAllOperationNames		operationNames: List<String>	Retrieves all operation names

Integration Dependencies

Interface Name	**External Component**
DelegatesGenerator	Synthesis Processor
Node Pool Manager	Cloud and Grid Middleware

9. CHOReOS Monitoring Subsystem

The CHOReOS Monitoring subsystem is part of the governance and V&V framework. By adopting a modular vision where the monitoring functionality is in a separate subsystem, the governance framework will continue to ensure other governance activities in the absence of the monitoring subsystem.

The *Business Service Monitoring* relies on an Event based Architecture (WS-Notification) for triggering the events from the probes deployed on the middleware and the Business Service monitoring. As part of the *Governance and V&V Framework*, the *Monitoring* subsystem interacts with the SLA and lifecycle manager presented in section 8.1 and the V&V components of section 8.2. It also interacts with the Resource monitoring addressed in section 1.1. For readability reasons we present the monitoring subsystem in a separate section.

In Figure 9-1, we depict the CHOReOS *Monitoring* most significant components. Actually, the monitoring functionality gathers data from both the *Distributed Service Bus* and the CHOReOS *Cloud and Grid* subsystem. Then, raw data from the services, the enacted choreographies, and the hardware resources are sent to the *CEP (Complex Event Processor)*. The latter is responsible for interpreting the collected information. It monitors and analyses the messages exchanged through the CHOReOS Middleware (i.e., primitive events), and inferring complex events. When a match with a complex event is detected (e.g., violation of more than one policy), the event is notified into specific channels of the middleware.

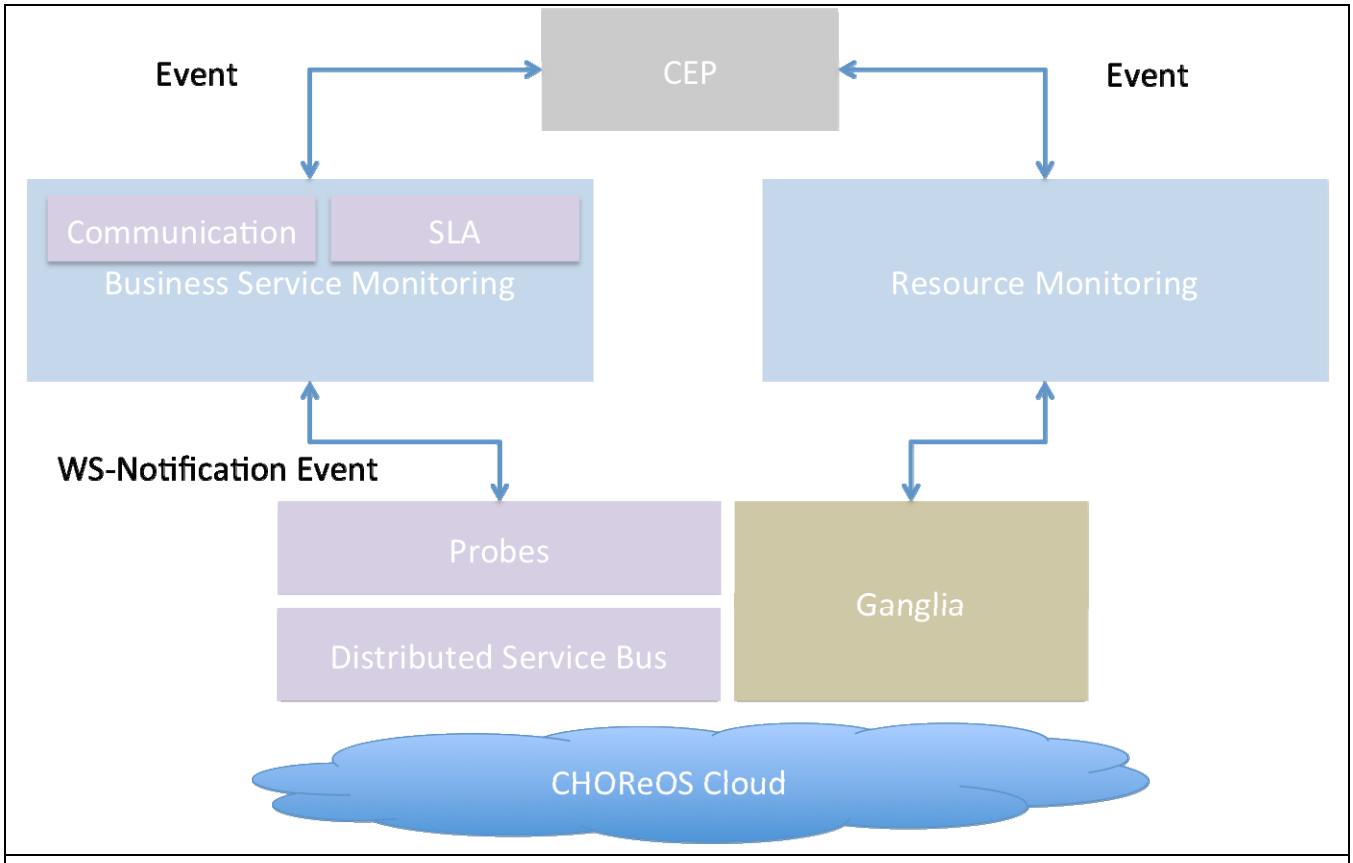


Figure 9-1. CHOReOS Monitoring Subsystem

9.1. Business Service Monitoring

The *Business Service Monitoring* is responsible for providing the monitoring functionality that relates to the services and choreographies. Indeed, it handles two main types of data: service data and communication data. In Figure 9-2, we depict the composition of the *Business Service Monitoring*.

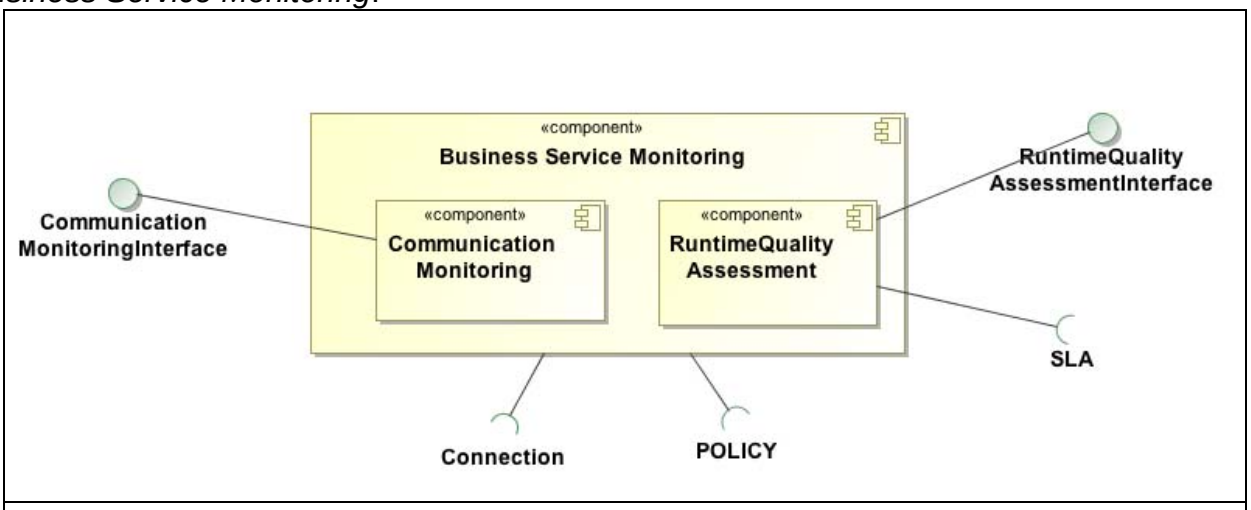


Figure 9-2. Business Service Monitoring

Subcomponents

- **Communication Monitoring Service:** The *Communication Monitoring* Service is responsible for gathering data about the interactions that occur between several services involved in collaborations and choreographies. The main functionality of this service is to ensure that the choreography is well behaving according to the specification policies. The Communication Monitoring Service provides the Communication Monitoring Interface.
- **Runtime Quality Assessment:** The *Runtime Quality Assessment* is responsible for the monitoring of the non-functional data of services. Actually, services contract service level agreements that describe their performances (time, security, etc.). Once deployed, services face the runtime conditions. Their performances may be different from the ones stated in the SLA. The *Runtime Quality Assessment* ensures the service is respecting the SLA contracted.

  Thanks to the connection of the *Business Service Monitoring* to the distributed service bus through dedicated probes, both functional and non-functional monitoring can be operated over a very largely distributed choreography of services.

Interfaces and Functionality

The Business Service Monitoring exposed the following interfaces and functionality:

Interface Provided Name	Operation Name	Input Parameters	Output Parameters	Responsibility
CommunicationMonitoring Interface	monitorCollaboration monitorChoreography	CollaborationIdentifier cIdentifier ChoreographyIdentifier cIdentifier	Event Event	The communication monitoring interface offers functions of Monitoring interactions taking part within a collaboration of services or a choreography.
Runtime Quality Assesment Interface	monitorSLA monitorChoreographySLA	ServiceEndpoint se ChoreographyIdentifier cIdentifier	Event Event	The Runtime Quality Assessment Interface gives functions for monitoring SLA for services and choreographies.

Integration Dependencies

The Business Service Monitoring depends on the following components:

External Interface Name	External Component
Connection	Distributed Service Bus
Policy	Policy Manager
SLA	SLA Manager

9.2. Resource Monitoring

The *Resource Monitoring* component is responsible for two related tasks:

- Actively supplying notifications to interested subsystems about relevant events, such as overloaded systems, out-of-memory conditions, or hardware failures. Active notifications are important for the system to engage in reactive measurements to correct problems as they occur.
- Maintaining an overview of the current and recent status of system resources to be able to respond to queries about them. Query responses are useful to support predictive tasks, such as the creation or destruction of virtual machine instances according to load, services allocation, or services migration.

Notifications will benefit from the *Complex Event Processor (CEP)* mentioned in the Monitoring Subsystem Overview; beyond complex event detection, the *CEP* also implements a notification mechanism following the publish/subscribe paradigm. A specific API will also be implemented to query the status.

Subcomponents

The resource monitoring subsystem will be based on a ganglia-like monitoring system[6], which will be executed on each virtual node of the system. Each node will have a small component responsible for data collection (primarily memory, disk, and CPU usage; others may be added as needed).

Higher-level nodes will aggregate data captured by individual nodes and keep a historical record of it in a round-robin database; Both current and past data will be made available over a specific API. These higher-level nodes will also be responsible for filtering data to collect relevant events, such as high loads or memory usage, and forward them to the CEP for further processing.

[6] http://ganglia.sourceforge.net

Interfaces and Functionality

The subsystem will offer an API similar to the API below:

Interface Name	Operation Name	Input Parameters	Output Parameters	Responsibility
ResourceMonitorInterface	GET	elementId: Id, startDate: Date, endDate: Date	collectedData: List<Date,List<String,String>>, children: List<Id> (The Strings are the attribute name and the corresponding collected value)	Lists collected data for the specified element in the specified period
	GET	elementId: Id, startDate: Date, endDate: Date, attribute: String	attribute: List<Date,String>	Lists collected data for the specified attribute of the element in the specified period

An element may be a specific machine instance or the aggregated data for a cluster, grid, or cloud zone. Attributes are strings such as “availableRam”, “loadAverage”, and “diskUsage”. If the endDate request parameter is ommited, the system responds with data corresponding to the single point in time closest to the startDate parameter. If the startDate parameter is also ommited, the system responds with the current data.

Integration Dependencies

The *Resource Monitoring* subsystem depends on the *CEP* for both complex event processing and for notifications; It may, however, work without it as a query-only system. In order for it to work, it also indirectly depends on the *NodePoolManager*, because each node needs to have the basic components preinstalled on them.

Conclusion

This deliverable has introduced an overall picture of the CHOReOS IDRE (see Figure 10-1), based on the specifications of constituent elements elaborated within Work Packages WP2-3-4 and reported in associated Deliverables D2-3-4.1. This deliverable has in particular defined the interfaces as well as examined dependencies among the various components of the IDRE, although acknowledging that the definitions of interfaces are called to evolve as the components get refined.

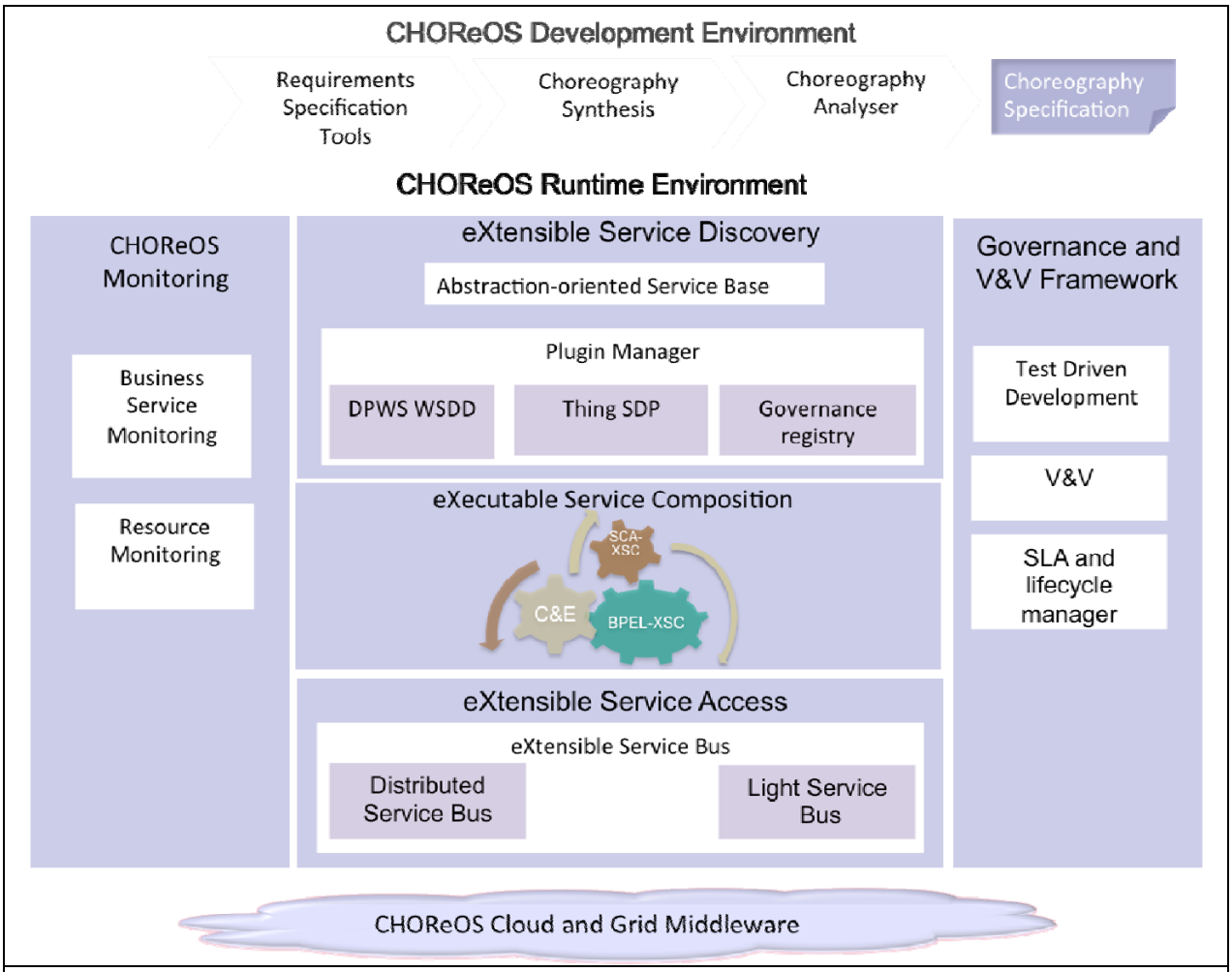


Figure 9-1. CHOReOS IDRE Overview

Following the specification of the IDRE, the related integration plan is defined in companion Deliverable D5.7.1, which describes the integration milestones and the technological tools that will enable integration. Deliverable D5.7.1 also describes the planned test beds for the IDRE assessment.