Deliverable D1.4 (b): Final CHOReOS Architectural Style and its Relation with the CHOReOS Development Process and IDRE

Marco Autili

ICT IP Project

Deliverable D1.4 (b)

Final CHOReOS Architectural Style and its Relation with the CHOReOS Development Process and IDRE

http://www.choreos.eu

LaTeX template v. 1.16

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

Deliverable Number	:	D1.4 (b)
Title of Deliverable	:	Final CHOReOS Architectural Style and its Relation with the CHOReOS Development Process and IDRE
Nature of Deliverable	:	Report
Dissemination level	:	Public
Licence	:	Creative Commons Attribution 3.0 License
Version	:	3.0
Contractual Delivery Date	:	31 March 2013 (Revised)
Actual Delivery Date	:	8 April 2013
Contributing WP	:	WP1
Editor(s)	:	Valérie Issarny (Inria)
Author(s)	:	Benjamin Billet (Inria), Georgios Bouloukakis (Inria), Nikolaos Georgantas (Inria), Sara Hachem (Inria), Valérie Issarny (Inria), Marco Autili (UDA), Davide Di Ruscio (UDA), Paola Inverardi (UDA), Massimo Tivoli (UDA), Amleto Di Salle (UDA), Dionysis Athanasopoulos (UOI), Panos Vasiliadis (UOI), Apostolos Zarras (UOI).
Reviewer(s)	:	Gustavo Ansaldi Oliva (USP), Sébastien Keller (THALES)

Abstract

This is Part b of Deliverable D1.4, which specifies the final CHOReOS architectural style, that is, the types of components, connectors, and configurations that are composed within the Future Internet of services, as enabled by the CHOReOS technologies developed in WP2 to WP4 and integrated in the WP5 IDRE. The definition of the CHOReOS architectural style is especially guided by the objective of meeting the challenges posed by the Future Internet, i.e.:

(i) the *ultra large* base of services and of consumers,

(ii) the high *heterogeneity* of the services that get composed, from the ones offered by tiny things to the ones hosted on powerful cloud computing infrastructures,

(iii) the increasing predominance of *mobile* consumers and services, which take over the original fixed Internet, and

(iv) the required *awareness* of, and related *adaptation* to, the continuous environmental changes.

Another critical challenge posed by the Future Internet is that of security, trust and privacy. However, the study of technologies dedicated to enforcing security, privacy and trust is beyond the scope of the CHOReOS project; instead, state of the art technologies and possibly latest results from projects focused on security solutions are built upon for the development of CHOReOS use cases -if and when needed-.

The CHOReOS architectural style that is presented in this deliverable refines the definition of the early style introduced in Deliverable D1.3. Key features of the CHOReOS architectural elements are as follows:

(1) The *CHOReOS service-based components* are technology agnostic and allow for the abstraction of the large diversity of Future Internet services, and particularly traditional Business services as well as Thing-based services; a key contribution of the component formalization lies in the inference of service abstractions that allows grouping services that are functionally similar in a systematic way, and thereby contributes to facing the ULS of the Future Internet together with dealing with system adaptation through service substitution.

(2) The *CHOReOS middleware-layer connectors* span the variety of interaction paradigms, both discrete and continuous, which are used in today's increasingly complex distributed systems, as opposed to enforcing a single interaction paradigm that is commonly undertaken in traditional SOA; a central contribution of the connector formalization is the introduction of a multi-paradigm connector type, which not solely allows having highly heterogeneous services composed in the Future Internet but also having those heterogeneous services interoperating even if based on distinct interaction paradigms.

(3) The *CHOReOS coordination protocols* introduce the third and last type of architectural elements characterizing the CHOReOS style. They specifically define the structure and behavior of service-oriented systems within the Future Internet as the fully distributed composition of services, i.e., choreographies; the key contribution of the work lies in a systematic model-based solution to choreography realizability, which synthesizes dedicated coordination delegates that govern the coordination of services.

Keyword List

Architectural Style, Components, Connectors, Coordination, Choreography, Future Internet, Interaction paradigms, Scalability, Interoperability.

Document History

Version	Changes	Author(s)
0.1	Outline Draft	Valérie Issarny (Inria)
1.x	First version and revisions of individual chapters	All authors
2.0	Overall integration and edition	Valérie Issarny (Inria)
3.0	Final revision	Valérie Issarny (Inria)

Document Reviews

Review	Date	Ver.	Reviewers	Comments
Outline	21 January 2013	1.0	All authors	Outline agreed by all
Draft	14 March 2013	1.x	Valérie Issarny	Intermediate version released for internal review
QA	29 March 2013	2.0	Gustavo Ansaldi Oliva, Sébastien Keller	Editorial comments
PTC	5 April 2013	A	PTC	-

Glossary, acronyms & abbreviations

Item	Description
AoSBM	Abstraction oriented Service Base Management
ATL	Atlas Transformation Language
API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Notation
BPMN2	Business Process Modeling Notation - ver. 2
CD	Coordination Delegate
CLTS	Choreography LTS - Labeled Transition System
CS	Client Server
DAML-S	DARPA Agent Markup Language for Services
DBMS	Data Base Management System
DSMS	Data Stream Management System
ESB	Enterprise Service Bus
FA	Functional Abstraction
FI	Future Internet
FLTL	Fluent Linear Temporal Logic
FSP	Finite State Processes
GA	Generic Application
HTTP	HyperText Transfer Protocol
IDL	Interface Description Language
IDRE	Integrated Development and Runtime Environment
IOPE	Inputs, Outputs, Preconditions, Effects
IoT	Internet of Things
JBI	Java Business Integration
JSON	JavaScript Object Notation
LTL	Linear Temporal Logic
LTS	Labeled Transition System
LTSA	Labeled Transition System Analyzer
MDE	Model Driven Engineering
MEP	Message Exchange Pattern
NFA	Non Functional Abstraction
OWL	Ontology Web Language
OWL-S	Ontology Web Language for Services
PS	Publish Subscribe
RDF	Resource Description Framework
REST	REpresentational State Transfer
RFID	Radio Frequency Identification
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol

S & A	Sensor and Actuators
SAWSDL	Semantically Annotated Web Service Description Language
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SOM	Service Oriented Middleware
SPARQL	SPARQL Protocol and RDF Query Language
STR	STReaming
STR-CS	STReaming Client/Server
STR-PS	STReaming Pub/sub
STR-TS	STReaming Tuple Space
TS	Tuple Space
ULS	Ultra-Large-Scale
ULS-FI	Ultra-Large-Scale Future Internet
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WADL	Web Applications Description Language
WP	Work Package
WS	Web Service
WSBQL	Web Service Based Query Language
WSCL	Web Service Conversation Language
WSDL	Web Service Definition Language
WSDL-S	Web Service Description Language with Semantics
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actuator Network
WSQM	Web Service Quality Model
XML	eXtensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol
XSB	eXtensible Service Bus
XSB	eXtensible Service Discovery

Table Of Contents

List Of Tables

List Of Figures

1 Introduction

Service-Oriented Computing (SOC) is now largely accepted as a well founded reference paradigm for Internet-based computing [73]. Under SOC, networked devices and their hosted applications are abstracted as autonomous loosely coupled services within a network of interacting service *providers*, *consumers* (aka *clients*) and *registries* according to the *service-oriented interaction pattern* (see Figure 1.1). Acknowledging such an adequacy of SOC and related Service-Oriented Architecture (SOA) to deal with the development of Internet-based software systems, CHOReOS investigates the evolution of SOA and supporting technologies so as to facilitate the development of the future generation of Internet-based software systems, i.e., systems envisioned as part of the Future Internet (r)evolution. Recalling that CHOReOS views the Future Internet as the aggregation of the Internet of Content, of Services and of Things [89], CHOReOS specifically concentrates on supporting the development of service-oriented distributed systems in the Future Internet, while overcoming the specific challenges that are posed by such an aggregation, which we summarize hereafter.

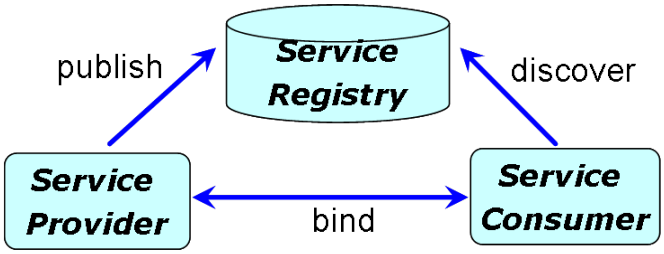


Figure 1.1: Service-oriented interaction pattern

1.1. CHOReOS Challenges for the Future Internet

Following thorough state of the art analysis, we have identified the following key challenges associated with the development of distributed service-oriented software systems to be deployed in the Future Internet network, as previously published by the CHOReOS consortium [89, 48]:

- *ULS – Ultra Large Scale*: The Internets of Content, Services and Things are confronted with scalability issues due to the increasing number, size and quality of their networked entities, which is further exacerbated by the empowerment of users who are now becoming "*prosumers*" (i.e., behave both as producers and consumers) [75, 72, 84]. For instance, simply considering the Internet of Things, the large amount of new information available through things needs to be comprehensively managed and aggregated to provide useful services [72].
- *Heterogeneity*: The Future Internet is heterogeneous in many dimensions, related to physical objects, networks, services and data, which presents a significant challenge for technically sustaining the Future Internet vision [72]. In particular, appropriate semantic technologies, shared standards and mediation are required to ensure interoperability of heterogeneous entities such as things, sensors, and networks [93].

- *Mobility*: Unlike the current Internet, mobility should be natively integrated in the design of the Future Internet. Indeed, an essential challenge for the Future Internet lies in the explicit design of a protocol for a mobile wireless world given that the majority of the connected entities are now mobile.
- *Awareness and adaptability*: Awareness and related adaptability are common requirements for sustaining the Future Internet, be it at the service, content or physical object level. Issues to be addressed include: adapting the Web by and for users, adapting the network to shared media and vice versa, providing personalized content and media to users, providing context-aware and personalized dynamic services [72, 84, 93].
- *Security, Privacy and Trust*: Trust, privacy and security are sensitive cross-domain issues that the current Internet is facing and remain as critical challenges for the Future Internet. With the global-scale communications and exchange of information, users' mobility and the limited resources their devices may have, as well as the Future Internet's "awareness" of users, their data, and their surroundings, it becomes crucial to find appropriate solutions that will protect users. Indeed, current security mechanisms are unfit in such an open, dynamic and aware setting.

Based on the above, the CHOReOS objective is to revisit the Service Oriented Architecture paradigm and supporting technologies so that they allow developing self-aware and adaptive distributed software systems out of an ultra large scale network of component systems that are highly heterogeneous and mobile for most of them. However, while the issue of Security, Privacy and Trust is acknowledged as a key concern for Future Internet systems, this is not tackled by the CHOReOS consortium, and is left as an area for future work, which may in particular build upon results of relevant European initiatives like the FP7 NESSOS "*Network of Excellence on Engineering Secure Future Internet Software Services and Systems*" (see `http://www.nessos-project.eu/` for more).

The core issues that CHOReOS investigates then relate to specifying the relevant service model for abstracting (component) systems of the Future Internet together with associated discovery, interaction and coordination protocols to face the scale, heterogeneity and mobility of the involved systems, and further support the evolution of the systems based on the awareness of the environment. The elicitation of the target service model and protocols is specifically one of the foci of CHOReOS WP1, which focuses on the formalization of the CHOReOS architectural style.

1.2. The CHOReOS Architectural Style for the Future Internet

Software systems may be abstractly described at the architectural level in terms of components and connectors: *components* are meant to encapsulate computation while *connectors* are meant to encapsulate interaction. In other words, control originates in components, and connectors are channels for coordinating the control flow (as well as data flow) between components [81]. A *software architecture style* further captures knowledge about effective design decisions for realizing specified goals within a particular application context [85]. A style is then characterized by the definition of types of: *components*, *connectors* and possibly *configurations* (i.e., system structures) that serve building a given class of systems. Hence, the definition of a software architectural style is central toward eliciting appropriate development and runtime support for any family of systems. Indeed, the style elements altogether specify the abstractions that need to be modeled, from design to implementation, as well as supported by the runtime to enact the target systems.

As CHOReOS adopts the SOA paradigm for the Future Internet systems, the definition of the CHOReOS architectural style revolves around the following types of architectural elements:

1) *Components that abstract services*, which may be refined into consumer, producer or prosumer services

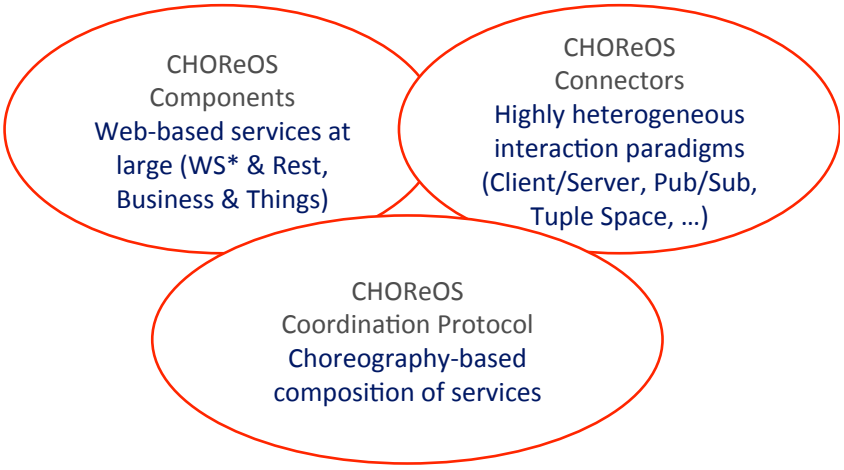


Figure 1.2: The CHOReOS architectural style

2) *Connectors that abstract client-service interaction protocols*, and

3) *Configurations that abstract compositions of services* through (service-oriented) connectors, i.e., *choreography* in the most general form, and orchestration as a specific composition structure that is commonly adopted in today's Internet.

This document then introduces the further refinement of the above architectural elements to support the development of distributed service-oriented systems aimed at the Future Internet. The document specifically finalizes the early definition of the CHOReOS architectural style introduced in Deliverable D1.3 [90]. In particular, the definition of the style reported in this document accounts for the lessons learnt with the implementation of the CHOReOS IDRE, which features the supporting technologies for the development of Future Internet service-oriented systems according to the CHOReOS style. While the next chapters provide the detailed definition of the CHOReOS architectural style, we review below how the Future Internet challenges are faced by the proposed style (see Figure 1.2); however, we recall that the issue of Security, Privacy and Trust is beyond the scope of the CHOReOS project and is thus not addressed in this document:

- *CHOReOS components*: The definition of the CHOReOS component model abstracts the various types of services encountered in the Future Internet, with a special focus on Business and Thing-based services. In addition, the notion of *service abstraction* is introduced to face the challenges of the Future Internet. In a nutshell, service abstractions characterize collections of compatible services. This allows structuring the ULS service base and hence more efficient search, whether performed by developers or using automated service discovery protocols. Further, service abstractions allow dealing with the substitution of service instances at runtime, as instances conforming to a given abstraction may substitute one another from the standpoint of functional and non-functional properties. As a result, CHOReOS components face: ULS (i.e., through the structuring of the ULS service base using service abstractions), heterogeneity (i.e., a CHOReOS component abstract any type of resources networked in the Future Internet), mobility (i.e., a CHOReOS component may be a mobile entity as it may in particular be dynamically

discovered in the environment), and awareness and adaptability (i.e., service abstractions enable the seamless substitution of services).

- *CHOReOS connectors*: The CHOReOS connectors abstract the middleware-layer protocols used by service components for interacting with their environment, taking into account the diversity of components that get connected in the Future Internet. CHOReOS connectors specifically overcome the heterogeneity of the Future Internet by introducing a multi-paradigm connector, which allows services to coordinate although they may exploit different paradigms (i.e., client-server, event-based and shared memory). In addition, considering that interactions in the Future Internet are increasingly continuous as opposed to discrete, in particular due to the networking with the physical world via things, CHOReOS connectors support both discrete and continuous interactions. As a result, CHOReOS connectors face: ULS (i.e., by promoting the usage of weakly coupled interaction paradigms like event-based communication), heterogeneity (i.e., CHOReOS connectors enable highly heterogeneous services to coordinate), mobility (i.e., CHOReOS connectors enable interaction with mobile services, as with any other type of services).
- *CHOReOS configurations*: CHOReOS promotes the fully distributed, choreography-based composition of services, so as to face the ULS of the Future Internet. And, thanks to the choreography of CHOReOS components via CHOReOS connectors, CHOReOS choreographies allow the composition of heterogeneous and mobile services, while being adaptive according to changes in the environment. Still, a critical challenge for the development of choreography-based systems is to ensure the realizability of choreographies, which is addressed in CHOReOS through the synthesis of dedicated coordination delegates.

Another challenge associated with the Future Internet, and especially its Ultra-Large Scale, relates to the required scalability in terms of sustaining a large number of users. This is specifically tackled in CHOReOS by adopting the Cloud computing technology, considering that the implementation of the CHOReOS run-time environment as well as of services, build upon Cloud solutions so that they can face varying load. This is then transparent to the definition of the CHOReOS architectural elements, for which we implicitly consider that they are possibly implemented over the cloud. Obviously, this has a direct impact on the implementation of the CHOReOS middleware, as detailed in the WP3 deliverables.

1.3. Document Outline

The next three chapters detail the definition of the above CHOReOS architectural elements. Specifically:

- Chapter 2 defines CHOReOS service-oriented components in a way that is technology-agnostic, and abstracts the diversity of services that get networked in the Future Internet, spanning business and thing-based services. The specifics of the proposed component model come from the definition of *functional and non-functional abstractions* for services, which enable hierarchically structured and hence scalable *abstraction-oriented service bases*. As further introduced in the chapter, the proposed abstractions also ease service substitution at runtime as service abstractions define pivot services against which behavior and interface of functionally matching service instances may be adapted.
- Chapter 3 defines the various connector types that are encountered in the Future Internet, spanning multiple coordination paradigms, together with discrete as well as continuous interactions. This further leads to the introduction of a multi-paradigm connector that allows interoperability across heterogeneous interaction paradigms while state of the art solutions focus on interoperability across middleware solutions based on the client-server interaction paradigm.

- Chapter 4 focuses on the formalization of the notion of CHOReOS *coordination protocol* that abstracts choreography behavior, together with the synthesis of coordination delegates that enforce the realizability of choreographies.

For the sake of illustration, in the next chapters, we exploit one of the CHOReOS use cases, that is the ”Passenger friendly airport” investigated within WP6, for which the interested reader may find relevant detail in WP6 deliverables available at `http://www.choreos.eu/bin/Download/Deliverables`.

It is important to stress that the definition of the CHOReOS architectural style is a conceptual exercise, for which the main goal is to introduce the key architectural concepts sustaining the development of service-oriented systems in the Future Internet (FI for short). In particular, our main objective in the definition of the CHOReOS style is to highlight the impact of the FI challenges upon the SOA paradigm, i.e.: the specifics of the resulting service-oriented components, connectors, and configuration. In addition, we provide a formal definition of the proposed architectural components so that their semantics are non ambiguous and can be easily leveraged for the implementation of supporting technologies.

A significant part of the CHOReOS project is particularly dedicated to the implementation of technologies supporting the development of service-oriented systems according to the CHOReOS architectural style. Further, acknowledging the fact that the definition of the style needs to be assessed against the lessons learned from the implementation, this document reports on the definition of the style after it has been informed by the development of the technologies that are integrated within the CHOReOS IDRE, while the early definition of the architectural style introduced in Deliverable [90] informed the development of the CHOReOS technologies. To make the link between the CHOReOS architectural style and CHOReOS IDRE explicit, Chapter 5 concludes the document by sketching the key features of the CHOReOS development process and associated IDRE in light of the definition of the CHOReOS architectural style.

2 CHOReOS Components: Abstracting Services

The definition of **CHOReOS components** constitute one of the three main parts of the CHOReOS architectural style that we propose towards addressing the main FI challenges, *i.e., scalability, heterogeneity, mobility,* and *awareness & adaptability*.

Concerning, *heterogeneity* and *mobility* we provide *a generic, unified formal model of services*. The purpose of this service model is:

- To capture the service-related information that is needed to represent both Business and Thing-based services, while abstracting the specificities of particular service paradigms, standards and technologies, so that services can in particular be dynamically bound to according to their semantics rather than their underlaying technology.
- To become the common ground for the overall CHOReOS development process and the methods, tools and middleware that support this process.

Regarding *scalability,* and *awareness & adaptability*, we formally define the concept of service abstractions, which represent groups of alternative services that provide similar functional/non-functional properties through different interfaces. These formal definitions capture the information of a corresponding *abstractions model*, whose purpose is:

- To enable, together with the generic service model, searching and browsing the plentitude of services that become available in the FI, thereby addressing scalability.
- To enable the dynamic substitution of services with other services that provide similar functional/non-functional properties, thereby addressing adaptability.

Hereafter, we use the term *CHOReOS (service-oriented) component model* to refer to both the *CHOReOS service model* and the *CHOReOS abstractions model*. As anticipated, the initial version of the proposed component model that we proposed in D1.3 did not fully cover the needs raised by the overall CHOReOS development process and the methods, tools & middleware that support this process. In particular, two main issues came up from the development of the CHOReOS IDRE:

- To refine the CHOReOS component model with definitions that concern Thing-based service types and instances.
- To refine the CHOReOS component model with definitions of components that are needed for the consistent, non-disruptive and scalable service adaptation.

The issues revealed from the development of the CHOReOS IDRE became, thus, the driving force for the revision of the proposed component model. These issues and the corresponding refined concepts of the model are further detailed in the rest of this chapter. More specifically, in Section 2.1, we begin with the revised definitions of the service model that have been introduced in D1.3 [90] and we proceed with the refinements of these definitions for Thing-based services. In Section 2.2, we start with the definitions of functional/non-functional abstractions and we discuss their role in service discovery, as introduced in D1.3 [90]. Following, we introduce the refinements of the abstractions model that concern service replacement.

$$Service\ type \quad : \quad s = (n, p, l, c, i, i_r, \mathcal{C}) \tag{2.1}$$
$$Interface \quad : \quad i = (n, p, O) \tag{2.2}$$
$$Operation \quad : \quad op = (n, p, In, Out, pre, post) \tag{2.3}$$
$$Service\ instance \quad : \quad si = (n, i, uri, d, nf) \tag{2.4}$$
$$\tag{2.5}$$

Table 2.1: Definitions & notations related to CHOReOS components

2.1. CHOReOS Service Model

In this section, we provide the formal definitions of the CHOReOS service model. The model accounts for the heterogeneity of services to be aggregated in the FI, while acknowledging that services are essentially (if not uniquely) Web-based at the (Future) Internet level although Web-based services are called to evolve to face the FI challenges (e.g., see next chapter on the need to deal with diverse interaction paradigms). The proposed service model was derived based on a detailed survey of the two major service paradigms, namely the WS* paradigm and the RESTful paradigm. The interested reader may refer to D1.3 [90] for further details regarding this survey. In the rest of this section, we provide the formal definitions of the CHOReOS service model that have been initially proposed in D1.3, and then we give their refinements for the case of Thing-based services.

2.1.1. Core Service Model

In this section, we recall the general paradigm-independent definitions (Table 2.1) of the concepts that constitute the CHOReOS service model. The model revises the original definition introduced in [90] according to the related definition of well-known semantic services technologies like OWL-S, SA-WSDL, WSMO, etc (see D1.3 for further details [90]). The model definition also accounts for the definition of the networked system model that is introduced in [19] to enable emergent middleware that supports on-the-fly interoperability in complex distributed systems. The rationale for the proposed definition is that the purpose of the CHOReOS service model is to provide a more abstract, unified view of relevant service technologies, rather than providing yet another specific technology.

Definition 1 (Service type) *Let Ω denote an infinitely countable set of domains and Σ denote an infinitely countable set of names. Then, a service type $s = (n, p, l, c, i, i_r, \mathcal{C})$ is defined as a tuple that consists of:*

- *A name, $s.n \in \Sigma$, that characterizes the service type.*
- *An optional service profile, $s.p$, such that $dom(s.p) \in \Omega$, that corresponds to a user-intuitive explanation of s; depending on the standard used, this can vary between a simple textual description, a list of keywords, a more advanced description (e.g., DAML/OWL-S/SAWSDL) or any other description.*
- *A style, $s.l$ such that $dom(s.l) = \{WS^*, RESTful\} \in \Omega$, which refers to whether s conforms to the WS* or to the RESTful paradigm.*
- *A port type name $s.c$ such that $dom(s.c) \in \Sigma$, which specifies the type of middleware-layer connector via which the component interacts (see Chapter 3 on CHOReOS connectors for detail).*
- *A provided interface, $s.i$, $dom(s.i) \in \Omega$, that specifies the functionalities provided by s.*

- *A required interface, $s.i_r$, $dom(s.i_r) \in \Omega$, that specifies the functionalities required by s.*
- *A set of capabilities $s.\mathcal{C}$, , such that $dom(s.\mathcal{C}) \in \Omega$, where each element of $s.\mathcal{C}$ is a pair (f, L) that defines a valid conversation L (expressed as an LTS over $s.i$ and $s.i_r$) with the service s, to provide a given high-level functionality $f \in \Sigma$.*

Definition 2 (Service interface) *A service interface $i = (n, p, O)$ is defined as a tuple that comprises:*

- *A name, $i.n \in \Sigma$, which characterizes the interface.*
- *An optional profile, $i.p$, such that $dom(i.p) \in \Omega$, that corresponds to a user-intuitive explanation of the interface.*
- *A set of operations, $i.O = \{op_1, \ldots, op_{|i.O|}\}$, that correspond to different functionalities provided through the interface. Note that this set may contain overloaded operations, i.e., operations with the same name and different input and output parameters.*

Definition 3 (Interface operation) *An interface operation $op = (n, p, In, Out, pre, post)$ is a tuple that consists of:*

- *A name, $op.n \in \Sigma$, for the operation.*
- *An optional profile, $op.p$, such that $dom(op.p) \in \Omega$ that contains a user-intuitive explanation of the operation.*
- *A set of input parameters, $op.In = \{p_1, \ldots, p_{|op.In|}\}$ and a set of output parameters $op.Out = \{p_1, \ldots, p_{|op.Out|}\}$. The set of output parameters may consist of a subset, $op.N \subseteq op.Out$, that comprises output parameters produced during the normal execution of the operation and a subset, $op.Ex \subseteq op.Out$, that comprises output parameters produced in exceptional situations. A parameter $p_i \in op.In \cup op.Out$ is generally defined as a tuple, $p_i = (n, p, t)$, that consists of a name, $p_i.n \in \Sigma$, an optional profile, $p_i.p$ such that $dom(p_i.p) \in \Omega$, and a type/domain $p_i.t \in \Omega$.*
- *An optional pre-condition, $op.pre$, that must hold for the correct execution of the operation. The pre-condition is generally defined as a predicate over the operation's constituent elements.*
- *An optional post-condition, $op.post$, that shall hold after the correct execution of the operation. As in the case of the pre-condition, the post-condition is generally defined as a predicate over the operation's constituent elements.*

Definition 4 (Service/Component instance) *Let $s = (n, p, l, c, i, i_r, \mathcal{C})$ be a type of services that follows the previous definitions. Then, a particular service instance (hereafter we also use the term service to refer to an instance of a service type) $si = (n, s, uri, d, nf)$ is defined as a tuple that consists of:*

- *A service name, $si.n \in \Sigma$.*
- *The service type $si.s$ that is implemented by si.*
- *An endpoint address, $si.uri$ such that $dom(si.uri) \in \Omega$.*
- *An optional description of implementation/middleware related details that defines the specific connector instance for $si.s.c$, which is used by the service for its interactions with the environment (e.g., the protocol used for accessing the service functionalities, the message format, etc.).*

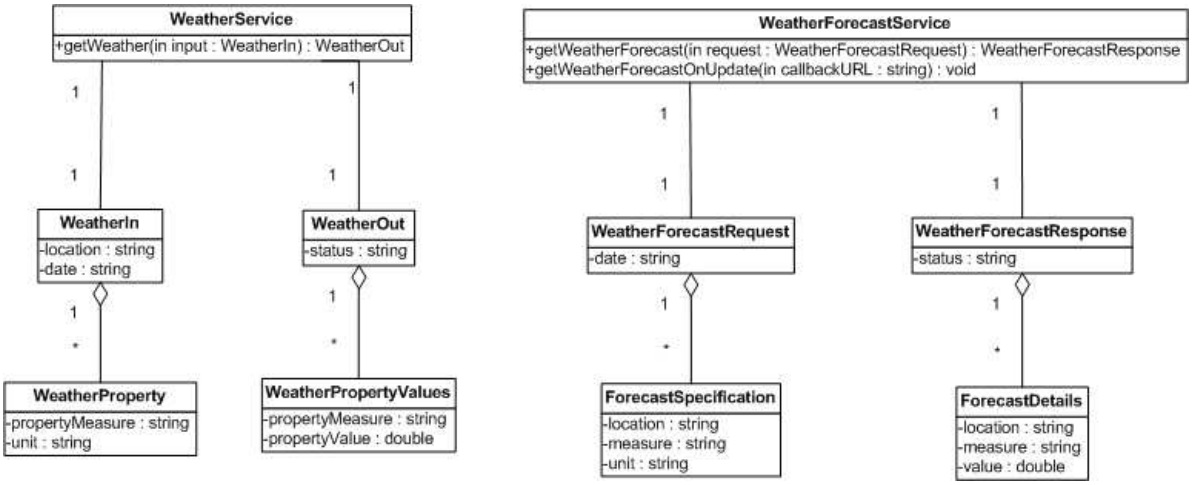


Figure 2.1: Weather services for the passenger friendly airport use case.

- *An optional description of non-functional properties that characterize si. Specifically, the non-functional description $si.nf = [qp_1, \ldots, qp_{|nf|}]$ is a vector of quality properties, where each property $si.nf[qp_i] = (q_i, value)$ is a tuple that consists of a quality indicator $q_i \in Q$ that belongs in a domain of quality indicators $Q \in \Omega$ and a $value \in dom(q_i)$ that belongs to the corresponding quality indicator domain. In general, we do not restrict the non-functional characterization of services to a standard domain of quality indicators; on the contrary, we employ this formulation to allow flexibility on this kind of descriptions. Nevertheless, we can distinguish between runtime quality indicators such as reliability, availability, reputation, price, performance [4], design quality indicators such as different kinds of cohesion for service interfaces discussed in [11], and finally physical properties that typically characterize Thing-based services.*

Taking a concrete example, in the passenger friendly airport use case of WP6, we may have several services that can play different roles specified in the choreographies involved (e.g., Airport, Weather forecast, ATC, Luggage handling company, Security company, etc.). Figure 2.1, for instance, shows two similar services, i.e., `WeatherService` and `WeatherForecastService`, which serve for reporting the weather. These services could be used to play the weather service role of the passenger friendly airport choreography. Then, the interface specifications of these services that conform to the proposed service model are given in Figure 2.2(a) and (b). In particular, `WeatherService` provides the `getWeather` operation, which reports, for a given date and location, the values of a given set of weather properties. Specifically, the input parameters of this operation comprise a location (e.g., Athens, Paris) and a date. The input parameters further comprise a list of `WeatherProperty` elements. Each `WeatherProperty` element is characterized by a `propertyMeasure` (e.g., temperature, humidity) and a `unit` for this `propertyMeasure` (e.g., Fahrenheit, Celsius). The output parameters of the operation comprise an overall weather `status` (e.g., sunny, cloudy) and a list of WeatherPropertyValue elements. Each such element is characterized by a `propertyMeasure` (e.g., temperature) and the value of this property (e.g., 19 C). The interfaces do not include profile specifications since there is no such information available.

On the other hand, the `WeatherForecast` service provides two operations. The `getWeatherForecast` operation reports for a given date the values of a given set of weather properties, at a given set of locations. Specifically, the input parameters of this operation comprise a date and a list of `ForecastSpecification` elements. Each `ForecastSpecification` element is characterized by a `location` (e.g., Athens, Paris), a `measure` (e.g., temperature, humidity) and a `unit` for this measure (e.g., Fahrenheit, Celsius). The output parameters of the operation consists of an overall weather `status` (e.g., sunny, cloudy) and a list of `ForecastDetails` elements. Each such element is

Interface WeatherService			
WeatherService.**O**			
getWeather			
In	**p.n**	**p.t**	**p.p**
	location	string	empty
	date	string	empty
	properties[*]	WeatherProperty	empty
Out	**p.n**	**p.t**	**p.p**
	status	string	empty
	propertyValues[*]	WeatherPropertyValues	empty
p	empty		
pre	empty		
post	empty		

(a) `WeatherService`

Interface WeatherForecastService							
WeatherForecastService.**O**							
getWeatherForecast					getWeatherForecastOnDemand		
In	**p.n**	**p.t**	**p.p**		**In**	**p.n**	**p.t**	**p.p**
	date	string	empty			callBackURL	string	empty
	specification[*]	Forecast Specification	empty					
Out	**p.n**	**p.t**	**p.p**		**Out**	**p.n**	**p.t**	**p.p**
	status	string	empty			empty	empty	empty
	forecast details[*]	Forecast Details	empty					
p	empty				**p**	empty		
pre	empty				**pre**	empty		
post	empty				**post**	empty		

(b) `WeatherForecastService`

Figure 2.2: Examples of CHOReOS component interface specifications for weather services.

characterized by a location, a `measure` (e.g., temperature) and the value of this measure (e.g., 19 C). The `getWeatherForecastOnDemand` operation, reports weather changes by sending information to a given URL. Hence, its input parameters comprise only the given URL, while its output parameters are empty. The interface does not include a profile specification.

Figure 2.3 gives a more technical view that shows how the service model specifications are represented in the CHOReOS service base [92, 88], which is developed in WP2. In particular, Figure 2.3(a) shows information concerning the `AirportService` type, which provides the `Airport` interface. The provided information comprises the operations of the `Airport` interface, the inputs/outputs for each operation and the URL of the capabilities specification of the `AirportService`. Figure 2.3(b) shows information concerning a service instance of the `AirportService` type. The provided information comprises the endpoint address and the non-functional properties that characterize the service instance.

2.1.2. Service Model Refinements for Thing-Based Services

An important aspect of the CHOReOS service model is to provide means for the specification of Thing-based services. The Internet of Things view [96], is a core challenge of the FI, which is expected to comprise an ultra large number of devices that will provide computing and communication capabilities, allowing them to interact with their surrounding environment (including the physical world) and the opposite. Given the large variety and the heterogeneity of Things, the service oriented paradigm is ideal towards seamlessly integrating them in the environment.

(a) `AirportService` type

(b) `AirportService` instance

Figure 2.3: Examples of CHOReOS component specifications in the AoSBM.

In CHOReOS, services may be traditional Business services as well as Thing-based services. In the initial CHOReOS middleware prototype, the Thing-based services that are supported are RESTful services; however, they may as well be WS∗. Still, Thing-based services are characterized by a number of specific properties, which refer to the relation with the physical world that they enable, as implemented by the CHOReOS Thing-based service-oriented middleware [88]:

- *ServiceType*: specifies the type of a particular Thing-based service.
- *PhysicalConcept*: defines the particular physical concept that is measured or acted upon by the Thing-based service.
- *ThingType* : specifies the type of the sensor/actuator that is abstracted by the Thing-based service.
- *DataType*: specifies the type of the data that is measured by/provided to/ the Thing-based service.
- *Units*: defines the specific units that are used for the measurements/actuations, according to a particular metrics system.
- *ServiceAddress*: defines the unique address of the Thing-based service.
- *Location*: defines the location of the Thing-based service.
- *Accuracy*: specifies the accuracy of the measurements/actuations produced by the Thing-based service.
- *Range*: defines the range of the sensing/actuation.

$$Thing\ Service\ Type \quad : \quad t = (n, p, l, c, i, \epsilon, \epsilon) \tag{2.6}$$

$$\begin{aligned} s.t. \quad & t.n \in dom(ServiceType) \wedge \\ & t.p = (phc, snt) \in dom(PhysicalConcept) \times dom(ThingType) \wedge \\ & t.l \in WS*, RESTful \end{aligned} \tag{2.7}$$

Table 2.2: Thing-based Service Type definition.

- *DeviceID*: specifies the unique identifier of the hosting device.

Given the aforementioned Thing-related properties, we refine below the CHOReOS service model so as to reflect these properties. On the one hand, the first five properties relate to the notion of service type (Definition 1). On the other hand, the last five properties relate to the notion of service instance (Definition 4). Consequently, we employ the first five properties to refine the definition of the notion of service type, towards the definition of the notion of Things-based service type. Moreover, we employ the last five properties to refine the definition of the notion of service instance, towards the definition of the notion of Thing-based service instance.

Definition 5 (Thing service type) *Let Ω denote an infinitely countable set of domains and Σ denote an infinitely countable set of names. Then, a Thing-based service type is a service type $t = (n, p, l, c, i, i_r, \mathcal{C})$ that satisfies the following constraints (Table 2.2):*

- *The name of the service type, $t.n \in \Sigma$, belongs to the domain of the ServiceType property, $dom(ServiceType)$.*
- *The profile of the service type is a tuple $t.p = (phc, snt)$ that consists of:*
  - *The physical concept, $t.p[phc]$, that is measured or acted upon. Hence, the physical concept $t.p[phc]$ belongs to the domain of the PhysicalConcept property, $dom(PhysicalConcept)$.*
  - *The thing type, $t.p[snt]$, that is abstracted by t. Therefore, the sensor/actuator type $t.p[snt]$ belongs to the domain of the ThingType property, $dom(ThingType)$.*
- *The style of the service type $t.l \in \{WS^*, RESTful\}$ (although the first CHOReOS prototype focuses on RESTful Thing-based services).*
- *The port type $s.c$ specifies the type of middleware-layer connector through which the Thing is accessed.*
- *The service type defines the interface $t.i$ used to access the Thing.*
- *The service type does not include any required interface, i.e., $t.i_r = \epsilon$.*
- *The service type does not expose any high-level capability, i.e., $t.\mathcal{C} = \epsilon$.*

Definition 6 (Thing service instance) *Let $t = (n, p, l, c, i, \epsilon, \epsilon)$ be a Thing-based service type that follows Definition 5. A Thing-based service instance of t is defined as a a tuple $ti = (n, t, uri, d, nf)$ that conforms with the following constraints (Table 2.3):*

- *The name of the service instance, $ti.n \in \Sigma$, belongs to the domain of the ServiceName property, $dom(ServiceName)$.*

$$
\begin{aligned}
Thing\ Service\ Instance \quad : \quad & ti = (n, t, uri, d, nf, R) && (2.8)\\
s.t. \quad & ti.n \in dom(ServiceName) \wedge \\
& ti.uri \in dom(ServiceAddress) \wedge \\
& ti.d \in dom(DeviceID) \wedge \\
& ti.nf = [loc, accu, range] \wedge \\
& ti.nf[loc] \in dom(Location) \wedge \\
& ti.nf[accu] \in dom(Accuracy) \wedge \\
& ti.nf[range] \in dom(Range) && (2.9)
\end{aligned}
$$

Table 2.3: Thing-based Service Instance definition.

- *The interface $t.i$ of the service type t is implemented by ti.*
- *The endpoint address, $ti.uri$, of the service instance belongs to the domain of the ServiceAddress property, $dom(ServiceAddress)$.*
- *The implementation related details, $ti.d$, of the service instance comprise the identifier of the hosting device. Hence, $ti.d$ belongs to the domain of the DeviceID property, $dom(DeviceID)$.*
- *The description of non-functional properties that characterize ti is a vector $ti.nf = [loc, accu, range]$ that comprises three properties:*
  - *The first property, $ti.nf[loc]$, is the location of ti. Hence, $ti.nf[loc]$ belongs to the domain of the Location property, $dom(Location)$.*
  - *The second property, $ti.nf[accu]$, is the accuracy of ti. Therefore, $ti.nf[accu]$ belongs to the domain of the Accuracy property, $dom(Accuracy)$.*
  - *The third property, $ti.nf[range]$, is the sensing/actuation range of ti. Thus, $ti.nf[range]$ belongs to the domain of the Range property, $dom(Range)$.*

2.2. CHOReOS Abstractions Model

The concepts that constitute the CHOReOS abstractions model are employed to facilitate service discovery and service adaptation. In this section, we first recall the formal definitions of the core concepts of the CHOReOS abstractions model that were introduced in D1.3 [90] and discuss their role in service discovery. Following, we refine the proposed abstractions model by introducing the formal definition of *functional abstraction services*, whose purpose is to facilitate service adaptation. Specifically, a functional abstraction service is derived from the specification of a corresponding functional abstraction. It realizes the abstract interface of the functional abstraction. Through this interface, the functional abstraction service provides unified access to the services that are represented by the functional abstraction. The unified access relies on the mappings between the abstract interface and the interfaces of the represented services, which are also part of the functional abstraction specification. The mappings are (re)configurable, allowing thus, to change dynamically the represented services that are accessed via the functional abstraction service.

$$Functional\ abstraction \ : \ fa = (i, R, M, anc, desc) \quad (2.10)$$
$$Mapping\ between\ fa.I\ and\ r_i \in fa.R \ : \ m_{ri_i} = (m_{op}, M_{In}, M_{Out}) \quad (2.11)$$
$$Operation\ mapping \ : \ m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O \quad (2.12)$$
$$Inputs\ mapping\ between\ op \in fa.I.O\ and\ m_{op}(op) \ : \ m_{in} : op.In \rightarrow m_{ri_i}.m_{op}(op).In \quad (2.13)$$
$$Outputs\ mapping\ between\ op \in fa.I.O\ and\ m_{op}(op) \ : \ m_{out} : op.Out \rightarrow m_{ri_i}.m_{op}(op).Out \quad (2.14)$$
$$Non-functional\ abstraction \ : \ nfa = (nf, R, anc, desc) \quad (2.15)$$

Table 2.4: Definitions related to the CHOReOS abstractions.

2.2.1. Core Abstractions Model

In the CHOReOS architectural style, we assume that information about services is organized in the CHOReOS Abstraction-oriented Service Base (AoSB). The realization of the abstraction-oriented service base is the CHOReOS AoSBM (AoSB Management) component that is developed within WP2 and that is provided as part of the CHOReOS XSD middleware service of WP3 (see [92, 88] for further details). More formally, the abstraction-oriented service base is defined below.

Definition 7 (Abstraction-oriented service base) *The CHOReOS abstraction-oriented service base* $sb = (C, FA, NFA)$ *is defined as a tuple that consists of:*

- *A set of service collections* $sb.C = \{c_1, \ldots, c_{|sb.C|}\}$. *Each collection* $c_i \in sb.C$ *is a set of service descriptions provided by a specific source. A source may be a service provider, a service registry/portal, a distributed service discovery protocol, etc.*
- *A set of functional abstractions hierarchies* $sb.FA = \{fa_1, \ldots, fa_{|sb.FA|}\}$. *Specifically, each element* $fa \in sb.FA$ *is the root functional abstraction of a corresponding hierarchy; the hierarchy is mined from a particular collection of services* $c \in sb.C$. *Note that for a collection* $c \in sb.C$ *there may be more than one functional hierarchy, mined based on different realizations of the mining process defined in Definition 9. Hence, we may have* $|sb.FA| \neq |sb.C|$
- *A set of non-functional abstractions hierarchies* $sb.NFA = \{nfa_1, \ldots, nfa_{|sb.NFA|}\}$. *Specifically, each element* $nfa \in sb.NFA$ *is the root of a corresponding hierarchy; the hierarchy is mined from a particular collection of services* $c \in sb.C$. *Again, for a collection* $c \in sb.C$ *there may be more than one non-functional hierarchy, mined based on different realizations of the mining process defined in Definition 12. Hence, we may have* $|sb.NFA| \neq |sb.C|$

Functional abstractions

As discussed in D1.3 [90], a functional abstraction represents a set of services. The notion of functional abstraction is employed to enable scalable service discovery; a query for a required service can be matched against a functional abstraction, instead of being matched against each one of the represented services. The notion of functional abstraction is further employed to enable service substitution; a choreography that is developed based on a functional abstraction can be easily reconfigured to use any of the services that are represented by the abstraction. Following, we recall the formal definition of functional abstraction (Table 2.4).

Definition 8 (Functional abstraction) *Given a service base* sb, *we define a functional abstraction* $fa = (i, R, M, anc, desc) \in sb.FA$ *that is reverse engineered from a collection of services* $c \in sb.C$ *as a tuple that comprises the following elements:*

- *A set of services $fa.R = \{si_1, \ldots, si_{|fa.R|}\}$, which are represented by fa.*
- *An abstract interface, $fa.i$, whose operations correspond to common/similar operations offered by the interfaces of the represented services $fa.R$.*
- *A set of mappings, $fa.M = \{m_{s_1}, \ldots, m_{s_{|fa.R|}}\}$, between the abstract interface $fa.i$ and the (provided) interfaces of the represented services. Specifically a mapping $m_{s_i} = (m_{op}, M_{In}, M_{Out}) \in fa.M$ between $fa.i$ and the (provided) interface ri_i of a represented service s_i is defined as a tuple that consists of:*
  - *A function $m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O$ between the operations $fa.i.O$ of the abstract interface and the operations $ri_i.O$ of the represented interface.*
    We assume that the mapping $m_{ri_i}.m_{op}$ is well-formed if the following conditions hold for each pair of mapped operations $op \in fa.i.O$ and $m_{ri_i}.m_{op}(op) \in ri_i.O$ [60]:
    1) $op.pre \Rightarrow m_{ri_i}.m_{op}(op).pre$
    2) $m_{ri_i}.m_{op}(op).post \Rightarrow op.post$
  - *A set of mappings $m_{ri_i}.M_{In}$ between the input parameters of mapped operations and a set of mappings $m_{ri_i}.M_{Out}$ between the output parameters of mapped operations. In particular, given the mapping $m_{ri_i}.m_{op} : fa.i.O \rightarrow ri_i.O$ between the operations $fa.i.O$ of the abstract interface and the operations $ri_i.O$ of the represented interface ri_i, for each pair of mapped operations $op \in fa.i.O$ and $m_{ri_i}.m_{op}(op) \in ri_i.O$ we have:*
    * *$m_{ri_i}.M_{In}$ contains a function $m_{in} : op.In \rightarrow m_{ri_i}.m_{op}(op).In$ for the inputs of the mapped operations.*
      We assume that m_{in} is well-formed if for every pair of mapped input parameters $p \in op.In$ and $m_{in}(p)$ the type of p is a sub-type of the type of $m_{in}(p)$ [60].
    * *$m_{ri_i}.M_{Out}$ contains a function $m_{out} : op.Out \rightarrow m_{ri_i}.m_{op}(op).Out$ for the outputs of the mapped operations.*
      With respect to the Liskov & Wing co-variance rule [60], we assume that m_{out} is well-formed if for every pair of mapped output parameters $p \in op.Out$ and $m_{out}(p)$ the type of $m_{out}(p)$ is a subtype of the type of p.
  - *In general, certain subsets of the services that are represented by fa may be further organized with respect to lower-level functional abstractions. Hence, fa is further characterized by a set of such lower-level abstractions, $fa.desc$. Moreover, the services that are represented by fa may be a subset of services organized with respect to a higher-level functional abstraction $fa.anc$.*

As already discussed in D1.3 [90], the well-formedness rules that we assume in the definition of functional abstractions do not guarantee strict behavioral compatibility between the represented services. The issue of behavioral compatibility is dealt in detail for each choreography that is synthesized, as part of the choreography synthesis process (see D2.2 [92] for further details). In addition, advanced behavioral compatibility relations based on mediation may be considered for greater flexibility in the definition of abstractions. This also includes accounting for possible knowledge about the ontology-based semantics of service operations so as to infer n-m mappings between operations, instead of simple 1-1 mappings, as investigated within the FP7 ICT Future and Emerging technology project CONNECT (see `https://www.connect-forever.eu/`) [8]. However, the elicitation of semantic-aware and mediation-based functional abstractions is area for future development of CHOReOS solutions, while current CHOReOS technologies support functional abstractions based on 1-1 mappings for operations and strong behavioral simulation.

Returning to the example of the services that can play the role of the weather service in the passenger friendly airport use case (Figure 2.1), we observe that the interfaces of the services offer a pair of very similar operations. In particular, the `getWeather` operation of `WeatherService` and the

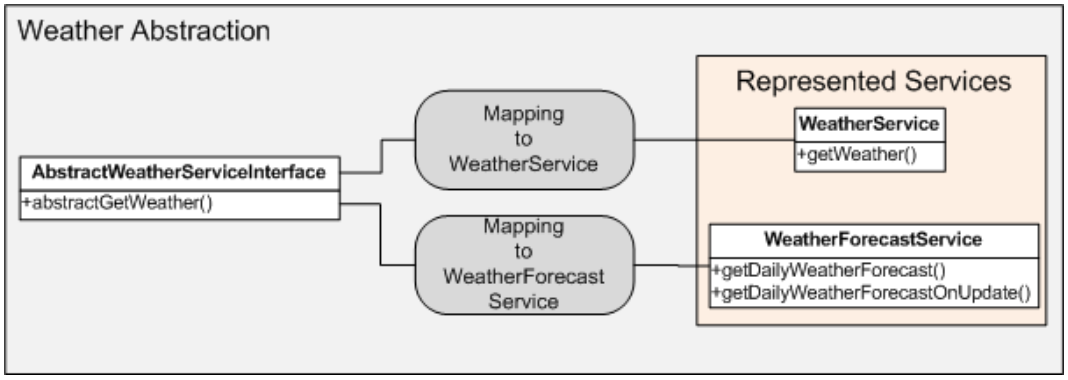


Figure 2.4: A functional abstraction for weather services.

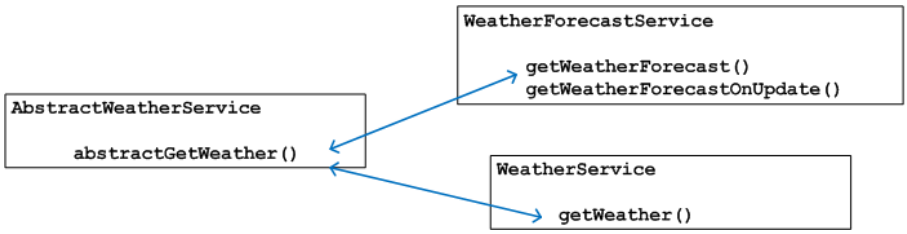


Figure 2.5: Operation mappings for the weather service abstraction.

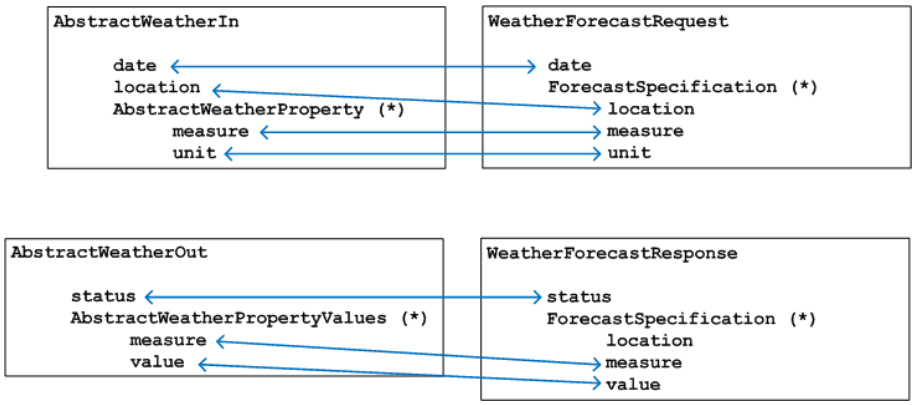


Figure 2.6: Parameter mappings for the weather service abstraction.

`getWeatherForecast` operation of `WeatherForecastService` report the values of a required set of weather properties for a given date. The only difference between these operations is that the former can report the values of the required weather properties for a single location that is given as input to the operation, while the latter can do the same task for multiple locations. Hence, there is an opportunity for defining a functional abstraction that represents these two services. A possible interface for this abstraction is given in Figure 2.4. In particular, the abstract service interface that represents the interfaces of the two services comprises a single operation named `abstractGetWeather`. The mapping of the abstract interface to the interfaces of the two services is rather straightforward and is given in Figure 2.5. Finally, Figure 2.6 gives an example of a mapping between the input/output parameters of the abstract operation `abstractGetWeather` and the input/output parameters of the `getWeatherForecast` operation of `WeatherForecastService`.

Figure 2.7 gives a more technical view of the representation of functional abstraction specifications in the CHOReOS abstraction-oriented service base [92, 88]. In particular, the figure shows a func-

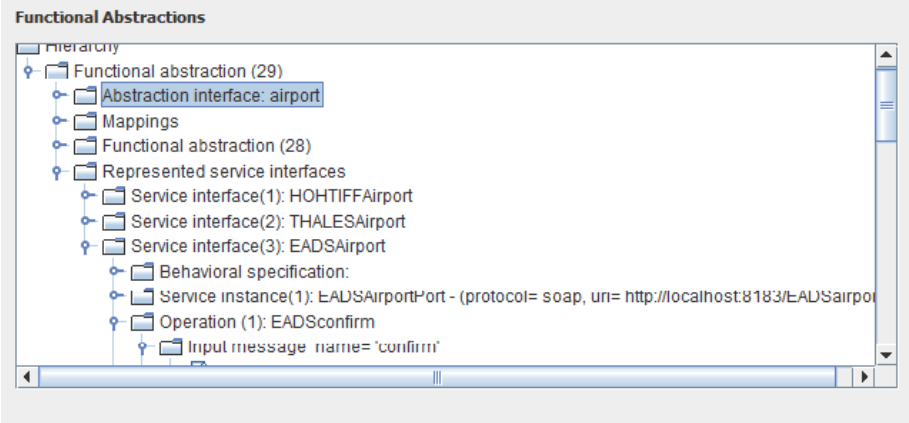


Figure 2.7: Functional abstraction representation in the AoSBM.

tional abstraction that represents a set of services, which can play the `Airport` role in the passenger friendly airport use case. The information that is provided includes the abstract interface, the represented services and the mappings between the abstract interface and the interfaces of the represented services.

Although the definition of a functional abstraction seems easy for the two services of our example, in the general case, this task is not straightforward. Consequently, in CHOReOS, as part of WP2 we provide methods for mining functional abstractions out of a given set of available services. Further details concerning these methods are given in D2.1 [87]. However, we recall below the general definition of the functional abstractions mining process that was introduced in D1.3 [90].

Definition 9 (Functional abstraction recovery) *The recovery of functional abstractions is realized by an operation* $ProduceFH$ *that accepts as input a collection of services* $c \in sb.C$ *that belong to the service base* sb *and computes a hierarchy of functional abstractions* fa*, which is then included in* $sb.FA$*, i.e.,* $ProduceFH : sb.C \rightarrow sb.FA$*.*

Non-functional abstractions

As discussed in D1.3 [90], a non-functional abstraction is a group of services that is characterized by a vector of quality properties, as it is done in the case of individual services. In general, the value of each non-functional property may be a combination of statistics like Mean/StdDevRange/Median that characterize the distribution of the quality property values that characterize the grouped services. The non-functional abstractions can be used to organize the services of a particular collection $c \in sb.C$ that belongs to the abstraction-oriented service base sb. Moreover, they can be employed to organize the services $fa.R$ that are represented by a functional abstraction $fa \in sb.FA$.

Taking the example of the weather services, suppose our quality properties are reputation, price and availability. Moreover, assume that `WeatherService`, `WeatherForecastService` and a third service `MeteoService` are characterized by corresponding quality properties, the values of which are given in Figure 2.8. Then, a non-functional abstraction that represents these three services could be characterized by the average value of reputation, the maximum price, and the minimum availability (i.e., $[reputation = -0.1, price = 55EURO, availability = 0,8]$). Nevertheless, often the designers or the domain experts that lookup for services do not really care about concrete quality property values. Instead, they may be interested in higher level, end-user friendly quality characterizations (e.g., $[reputation = neutral, price = cheap, availability = acceptable]$) [12]. For that reason, in the case of non-functional abstractions in [90], we introduced the concept of *domain hierarchies* that characterize quality indicators, which is given below.

	reputation	*price*	*availability*
WeatherService	-0.7	50 EURO	0.8
WeatherForecastService	-0.1	60 EURO	0.95
MeteoService	0	55 EURO	0.85

Figure 2.8: Non-functional properties for the weather services

Definition 10 (Domain hierarchy) *Let* $\mathbf{Q}$ *denote a countable set of quality indicators of interest and* $\mathbf{\Omega}$ *denote an infinitely countable set of domains. Each quality indicator* $q \in Q$ *is characterized by a concrete domain* $dom(q) \in \Omega$*. The domain can be a discrete or dense set of acceptable values for the quality indicator (e.g., the quality indicator reliability can have values in the dense domain of [0..1]). Then, a domain hierarchy* $\mathbf{H} = \{\delta_1, \ldots, \delta_{|H|}\}$ *for* q *is a finite* list *of domains* $\{\delta_1, \ldots, \delta_n\}$ *such that* $\delta_1 = dom(q)$*. The following constraints are further required for these domains:*

- *For every pair of domains* δ_{low}, δ_{high} *such that* $low < high$ *(i.e.,* δ_{low} *is found lower in the hierarchy than* δ_{high}*), a total, onto* ancestor *function is defined for their members* $\alpha_{\delta_{low}}^{\delta_{high}} : \delta_{low} \rightarrow \delta_{high}$*.*
- $\forall i < j < k, x \in \delta_i, y \in \delta_j, z \in \delta_k | \alpha_{\delta_i}^{\delta_j}(x) = y \wedge \alpha_{\delta_j}^{\delta_k}(y) = z \Rightarrow \alpha_{\delta_i}^{\delta_k}(x) = z$
- *For every* $\delta_{low}, \delta_{high} \in H$*, if* δ_{low} *is ordered then,* $\forall x, y \in \delta_{low} | x < y \Rightarrow \alpha_{\delta_{low}}^{\delta_{high}}(x) \leq \alpha_{\delta_{low}}^{\delta_{high}}(y)$*.*

We say that a domain δ_{high} *is* more general*, or* subsumes*, or* dominates *another domain* δ_{low} *in the same hierarchy whenever* $low < high$ *and the function* $\alpha_{\delta_{low}}^{\delta_{high}}$ *is defined.*

Intuitively, a domain hierarchy is a list of abstraction levels, each providing more abstract characterizations of the values of the domains found lower in the hierarchy. The lower a position in the hierarchy's list, the more detailed the domain is. The values of the domains in the hierarchy are mapped to each other via an ancestor function α. We require that α is (a) a function, (b) total and (c) onto. In other words, all the values of a lower domain are mapped to a higher level and the values of higher levels in the hierarchy always have descendants. Also, we require that there is a consistency constraint between the members of the domains. Specifically, since one can ascend from level say δ_2 to level δ_5 either directly, via the function $\alpha_{\delta_2}^{\delta_5}$ or via the composition of different functions over the intermediate levels, we require that the result of all these paths over the values of the involved domains result in the same value in δ_5. By definition, a more general domain abstraction has at the most the same cardinality with a detailed one. Note also that the inverse of α is not a function. Finally, well-formed hierarchies map continuous ranges of detailed values to abstract values. Equivalently, a well-formed abstraction imposes an equivalence relation that partitions the detailed domain in continuous ranges of values.

In our example that involves the weather services that can be used in the passenger friendly airport use case, we can employ the domain hierarchies that are given in Figure 2.9. The dotted edges signify the ranges of the ancestor functions between two levels. Observe that the lowest level is dense whereas the others are discrete, without any theoretical problems. Also, observe that the ancestor function involves ranges which is a typically expected case for domains where an ordering can be defined with an intuitive manner, but not obligatory.

Based on the previous concepts, we recall below the formal definition of non-functional abstractions that was introduced in [90].

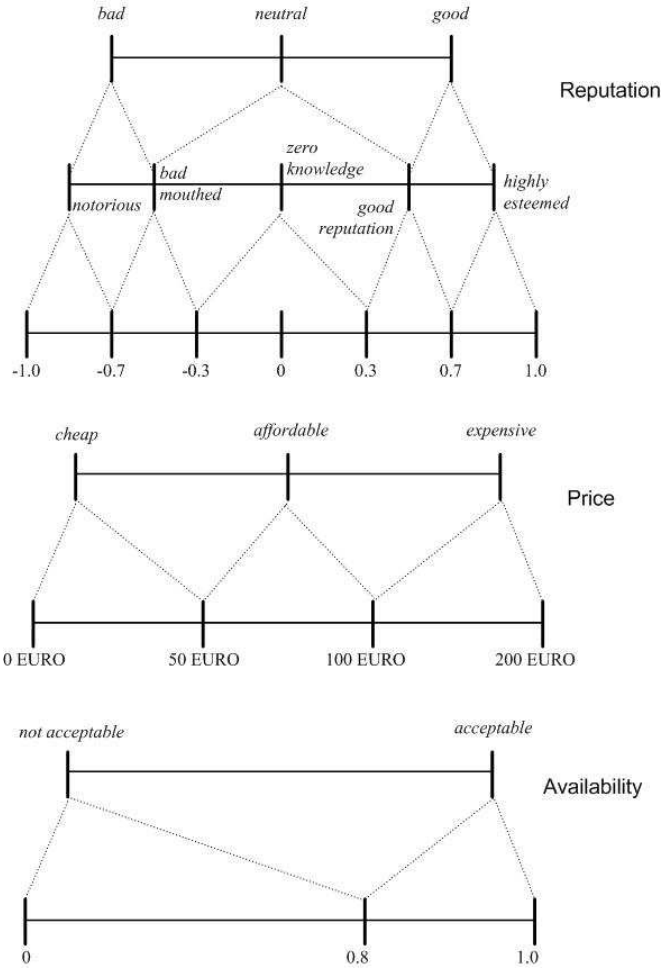


Figure 2.9: Domain hierarchies for reputation, price and availability

Definition 11 (Non-functional abstraction) *Given a set of quality indicators $Q = \{q_1, \ldots, q_{|Q|}\}$ (each quality indicator $q_i \in Q$ is characterized by a concrete domain $dom(q_i) \in \Omega$) and a set of domain hierarchies $S_H = \{H_1, \ldots H_{|S_H|}\}$ defined for the set of quality indicators Q, a non-functional abstraction $nfa \in sb.NFA$ is defined as a tuple $nfa = (nf, R, anc, desc)$ where:*

- *$nfa.R$ is the set of the services represented by nfa.*
- *$nfa.nf = [qp_1, \ldots, qp_{|nf|}]$ is a vector of quality properties, where each property $nfa.nf[qp_i] = (q_i, value)$ is a tuple that consists of a quality indicator $q_i \in Q$ and a $value \in \delta_{q_i}$, the domain of which belongs to the domain hierarchy $H_i \in S_H$ that corresponds to the quality indicator q_i.*
- *For each service $s \in nfa.R$ and for each quality indicator $q_i \in Q$, $nfa.nf[qp_i].value = \alpha^{\delta_{q_i}}_{dom(q_i)}(s.nf[qp_i].value)$. In other words, all the services of the non-functional abstraction are characterized by the same abstract values for all their quality indicators.*
- *Certain subsets of the services that are represented by nfa may be further organized with respect to lower-level non-functional abstractions. Hence, nfa is further characterized by a set of such lower-level abstractions, $nfa.desc$. Moreover, the services that are represented by nfa may be a subset of services organized with respect to a higher-level functional abstraction $nfa.anc$.*

Going back to our example, based on the non-functional properties of the weather services of Figure 2.8 and the domain hierarchies of Figure 2.9, it is possible to define a non-functional abstraction that groups the three services with the characterization $[reputation = neutral, price =$

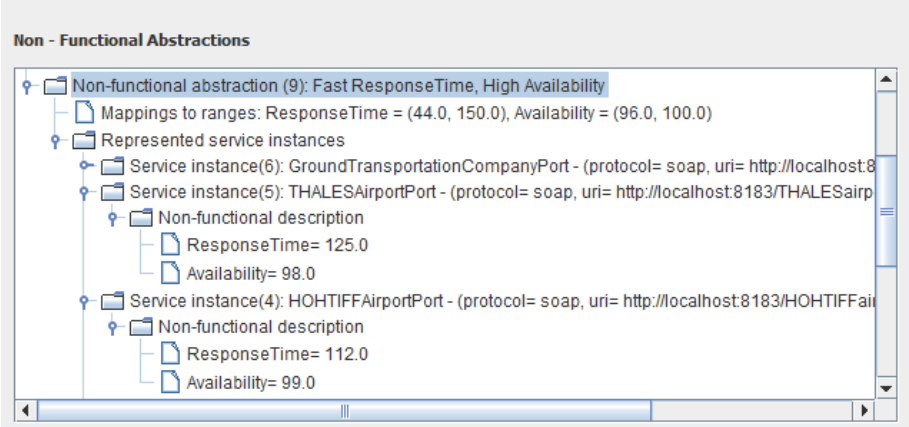


Figure 2.10: Non-functional abstraction representation in the AoSBM.

$affordable, availability = acceptable]$. Moreover, it is possible to define two different abstractions, one for representing `WeatherForecaseService` and `MeteoService` with the characterization $[reputation = zero - knowledge, price = affordable, availability = acceptable]$ and the second one that represents only `WeatherService` with the characterization $[reputation = bad - mouthed, price = affordable, availability = acceptable]$.

A more technical view of the representation of non-functional abstraction specifications in the CHOReOS abstraction-oriented service base [92, 88] is given in Figure 2.10. Specifically, the figure shows a non-functional abstraction that represents service instances with the characterization $[responce - time = fast, availability = high]$.

As in the case of functional abstractions, D2.1 [87] comprises methods for mining non-functional abstractions out of existing services. Below, we recall the general definition of this mining process, which was introduced in D1.3 [90].

Definition 12 (Non-functional abstraction recovery) *The recovery of non-functional abstractions is realized by an operation $ProduceNFH$ that accepts as input either a collection of services $c \in sb.C$ that belong to the abstraction-oriented service base sb or a set of services that are represented by a functional abstraction $nfa \in sb.FA$ and computes a hierarchy of non-functional abstractions nfa, which is included in $sb.NFA$, i.e., $ProduceNFH : sb.C \bigcup_{\forall fa \in sb.FA}(\{fa.R\}) \rightarrow sb.NFA$.*

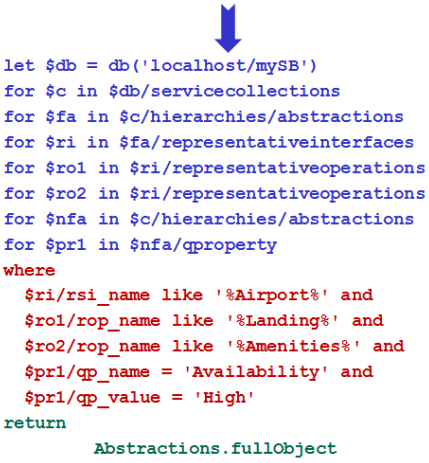
2.2.2. Abstraction-Driven Discovery of Services

Following, we recall the definitions of the main functionalities that concern the management of the abstraction-oriented service base that were introduced in D1.3 [90]. The details concerning the realization of these functionalities are discussed in D2.1 [87] and D2.2 [92].

Definition 13 (Service registration) *The registration of services information in the abstraction-oriented service base $sb = (C, FA, NFA)$ is realized by an operation $Register : Src \rightarrow sb.C$ that may accept as input a source selected from a set of known sources Src that provide information about services. Then, the operation constructs a new collection of services by retrieving information about available services from the source. Following service registration, the new collection can be organized with respect to the mining processes, defined in Definitions 9 and 12.*

Definition 14 (Refreshment) *The refreshment of a collection that already exists in the abstraction-oriented service base $sb = (C, FA, NFA)$ is realized by an operation $Refresh : Src \times sb.C \rightarrow sb.C$ that accepts as input the collection and the source used for producing the collection. Then, the operation upgrades the contents of the existing collection. This task may further trigger the upgrade of related*

A query for the Airport role



```
let $db = db('localhost/mySB')
for $c in $db/servicecollections
for $fa in $c/hierarchies/abstractions
for $ri in $fa/representativeinterfaces
for $ro1 in $ri/representativeoperations
for $ro2 in $ri/representativeoperations
for $nfa in $c/hierarchies/abstractions
for $pr1 in $nfa/qproperty
where
  $ri/rsi_name like '%Airport%' and
  $ro1/rop_name like '%Landing%' and
  $ro2/rop_name like '%Amenities%' and
  $pr1/qp_name = 'Availability' and
  $pr1/qp_value = 'High'
return
        Abstractions.fullObject
```

Figure 2.11: A WSBQL query for airport services.

abstractions hierarchies. If the abstractions hierarchies radically change (i.e., new abstractions are introduced in the hierarchies) the previous versions may be kept either for a limited time or for as long as they are needed (i.e., there exist choreographies that rely on these hierarchies).

Definition 15 (Service lookup) *Executing a query over the abstraction-oriented service base* $sb = (C, FA, NFA)$ *involves an operation* $executeLookupQuery : dom_q \rightarrow P(sb.FA) \times P(sb.NFA)$ *that accepts as input a query expression* $q \in dom_q$*, such that* $dom_q \in \Omega$*. The lookup returns a set of functional abstractions and a set of non-functional abstractions that meet the (functional/non-functional) constraints, specified in* q*. A query expression to the abstraction-oriented service base is a tuple* q=(ϕ_{ST},ϕ_S)*, where:*

- ϕ_{ST} *is an expression that specifies constraints over the properties of the service types, stored in the service base.*
- ϕ_S *is an expression over the properties of the service instances, stored in the service base.*

The syntax and semantics of the query are further detailed in D2.2 [92], where we propose the WSBQL language that realizes the proposed approach. To give an idea of how the querying is done, Figure 2.11 gives a WSBQL query expression for services that can play the Airport role of the passenger friendly airport use case. In this query, we have an expression on the properties of the interface that is searched; the name of the interface should be similar to `Airport`. Moreover, we have two expressions over the operations of the interface to be provided by the service being sought; the interface should expose an operation, whose name is similar to `Landing` and another one, whose name is similar to `Amenities`[1]. Finally, we have an expression over the required non-functional properties; the `Availability` of the services should be `High`. The results of the query should be grouped with respect to the abstractions that represent the retrieved services. A more technical view that shows the execution of the query of Figure 2.11 in the AoSBM is given in Figure 2.12.

2.2.3. Abstraction-Driven Service Adaptation

The issue of dynamic reconfiguration became a hot topic back in the 90's. The main objective of dynamic reconfiguration is to perform changes to a running system, so as to deal with dynamically evolving aspects that relate to the system. Such aspects may be the functional/non-functional requirements that

[1] The semantics of the "like" operator in WSBQL are the same with the semantics of the SQL "like" operator [92].

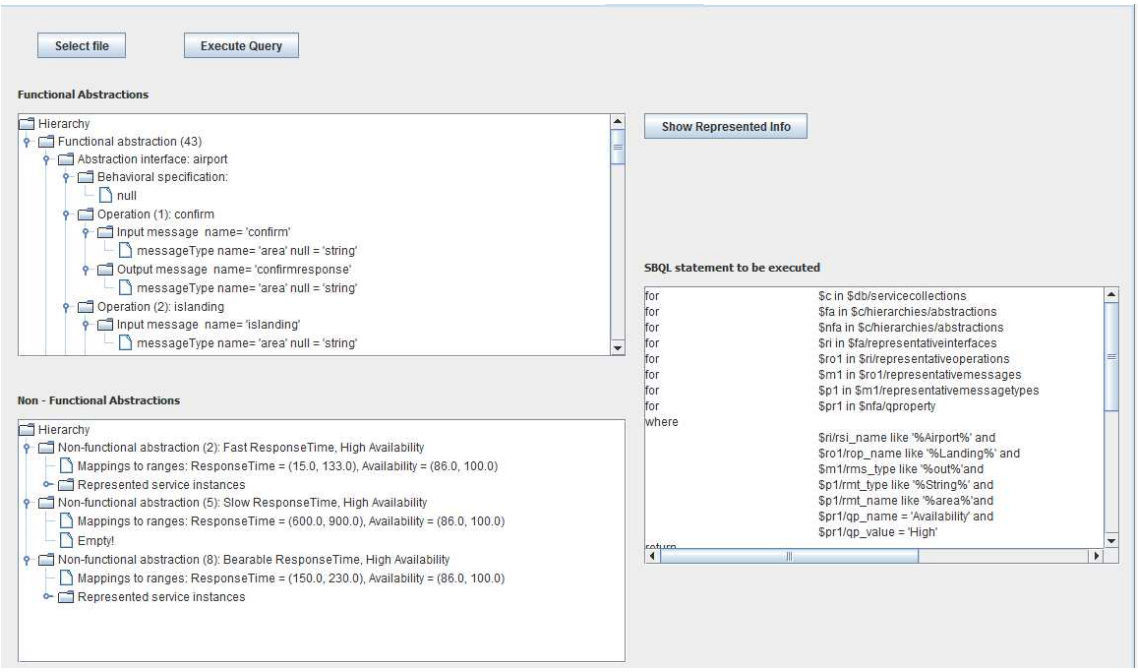


Figure 2.12: WSBQL execution in AoSBM.

should be satisfied by the system, the functional/non-functional properties of the components/connectors that constitute the system, properties of the execution environment where the system operates, etc. The changes that may be performed typically involve adding, removing, or substituting components/-connectors of the running system [58].

In CHOReOS, we consider service choreographies as the main paradigm for developing systems in the Future Internet. In this context, we investigate the role of abstractions in choreography adaptation; we more specifically investigate the *adaptation of choreographies through the replacement of services with other substitute services based on functional abstractions*.

The rest of this section is structured as follows. First, we revisit the seminal work of Kramer & Magee, who introduced a fundamental set of properties that should characterize a dynamic reconfiguration process, so as to soundly manage changes in the configuration of a running system [51]. Then, we extend this fundamental set of properties to deal with the specificities of the FI. Following, we discuss the role of abstractions in choreography adaptation and more specifically related service substitution, in relation to the previously mentioned extended set of properties.

Adaptation Properties

In [51], Kramer & Magee formally define the following set of fundamental properties that should characterize a reconfiguration process, which soundly manages changes to the configuration of a running system:

- *Consistency*: As said in [51]: *"changes should leave the system in a consistent state"*. A consistent state is one from which the system can continue providing correct service rather than progressing towards an error state. The definition of consistent states for a particular system involves specifying certain invariants (e.g., safety and liveness properties) that must hold during the system's execution. Moreover, defining consistent states relates to the specification of a fault model regarding elements that constitute the system.
- *Minimal disruption*: As said in [51]: *"changes should minimize the disruption to the system"*. In other words, changing some system elements should not cause the suspension of the whole

system's execution.

In CHOReOS, we adapt and extend this fundamental set of properties to deal with choreography adaptation based on service substitution, in the context of the FI:

- *Consistency*: The substitution of a service should leave the services that depend on the substituted service in a consistent state. A consistent state for a service is a state from which the service can continue operating correctly.
- *Minimal disruption*: Substituting a service should not cause the suspension of the whole choreography.
- *Scalability*: The architectural elements that are in charge of the adaptation should be decentralized. Moreover, the architectural elements that are in charge of the adaptation should not assume global knowledge regarding the choreography structure. Finally, the complexity of the different tasks of the adaptation should scale reasonably with respect to the entities involved in these tasks.

The Role of Abstractions in Service Adaptation

Section 2.2.1 defines the notion of functional abstraction that represents a group of services, which provide similar functionalities. The functional abstraction is characterized by an abstract interface and mappings between this interface and the interfaces of the represented services. From the specification of a functional abstraction, we derive a corresponding implementation, called *functional abstraction service*, which facilitates the substitution of the services that are represented by the functional abstraction.

Specifically, substituting a service with another one implies changes to the services that depend on the one that is substituted. This impact may be extremely large in the context of the FI. The notion of functional abstraction service allows reducing this impact as required by *the minimal disruption property*. The functional abstraction service *hides* from the dependent services the service that they depend on. The dependent services use the service that they need, by invoking the functional abstraction service that hides it, instead of invoking directly the service. In this setting, the hidden service can be substituted with another one, without affecting the dependent services.

In a sense, a functional abstraction is a meta-service type, while a functional abstraction service is a service type that is derived from it. More formally, we define the notion of *functional abstraction service* as follows.

Definition 16 (Functional abstraction service) *Given a functional abstraction* $fa = (i, R, M, anc, desc)$*, the corresponding functional abstraction service* $fast = (n, p, l, c, i, i_r, \mathcal{C})$ *is a service type for which the following properties hold:*

- $fast$ *uses the connector type* $fast.c$ *that allows interacting with any service of the functional abstraction, in the most general case* $fast.c$ *is the GA multi-paradigm connector type that is introduced in the next chapter.*
- $fast$ *provides the interface* $fast.i$ *that is a superset of the interface that is specified by* fa*, i.e.,* $fa.i \subset fast.i$*.*
- *Besides the operations of* $fa.i$*, the interface* $fast.i$ *comprises the following operations:*
  - *The* $SetServiceInterface$ *operation that (re)configures the interface* hi *of the service* $hsi \in fa.R$ *that is hidden behind* $fast$*.*
  - *The* $SetServiceInstance$ *operation that (re)configures the URI,* $hsi.uri$*, of the service* $hsi \in fa.R$ *that is hidden behind* $fast$*.*

- *The $SetCurrentMapping$ operation that (re)configures the mapping $m_{hi} \in fa.M$ that maps the operations of $fa.i$ to the operations of hi.*

- *Based on the mapping $m_{hi} \in fa.M$, the realization of each operation $op \in fast.i \cap fi.i$ transforms the invocations made to op to invocations of the $m_{hi}.m_{op}(op)$ operation of the hidden interface hi.*
- *$fast$ does not specify any required interfaces and hence $fast.i_r = \epsilon$*
- *$fast$ does not specify any behavior and hence $fast.\mathcal{C} = \epsilon$*

The notion of functional abstraction service allows substituting services without disrupting much the services that use them. Nevertheless, *consistency still remains an issue that should be dealt with*. A consistency problem that should be handled when substituting a service with another one concerns the behavioral compatibility of the services involved. As already mentioned, the behavioral compatibility of the services that synthesize a choreography is handled at the level of the choreography synthesis process and therefore is not further discussed here (see D2.2 [92]).

Another consistency problem that should be handled, has to do with interactions that took place with the service before the substitution. Several services may have used the substituted service to obtain certain data, or perform certain tasks. The data that have been obtained, or the tasks that have been performed may become invalid after the substitution. This situation can be avoided if it is possible to transfer the state of the service to its substitute [34, 35]. In practice, assuming that services expose state information is quite reasonable in the context of a particular organization or federation. In the broader context of FI, we make a weaker assumption. More specifically, we assume that the services are notified about the substitution of the services that they use. Given such a notification, each service should maintain its local consistency. To realize this approach, we introduce the notion of *impact analyzers*. In particular, each functional abstraction service is associated with an impact analyzer that is responsible for changing the service that is hidden behind the functional abstraction service and notifying the services that used the hidden service via the functional abstraction service.

Definition 17 (Impact Analyzer) *The impact analyzer is a service type $im = (n, p, l, c, i, \epsilon, \epsilon)$ that is responsible for adapting the services hidden behind functional abstraction services. Specifically, each impact analyzer instance is associated with a particular functional abstraction service instance, $fasti$. It maintains a list of services, D, which depend on $fasti$. To this end, im provides an interface $im.i$, which offers the following operations:*

- *The $Register$ operation allows services to register themselves to D.*
- *The $Unregister$ operation allows services to remove themselves from D.*
- *The $Adapt$ operation substitutes the service that is hidden behind $fasti$ with a given (behaviorally compatible) substitute service. As part of this process, each registered service is notified about the beginning and the end of the substitution. Following a notification, each registered service is in charge of maintaining its local consistency.*

Back to our example, Figure 2.13 gives a functional abstraction service that corresponds to the functional abstraction of Figure 2.4. The interface of the `AbstractWeatherService` provides the `getWeather` operation, along with the additional operations specified above. The service hides the `WeatherService` and the `WeatherForecastService`. Invocations to the `getWeather` operation are transformed to corresponding invocations of the operations that are offered by the hidden services according to the mappings that are given in Figures 2.5 and 2.6. Moreover, Figure 2.13 shows the impact analyzer that is associated with the `AbstractWeatherService`.

The last issue that should be discussed is *scalability*. In particular, the proposed approach does not involve centralized architectural elements; services are hidden behind dedicated functional abstraction

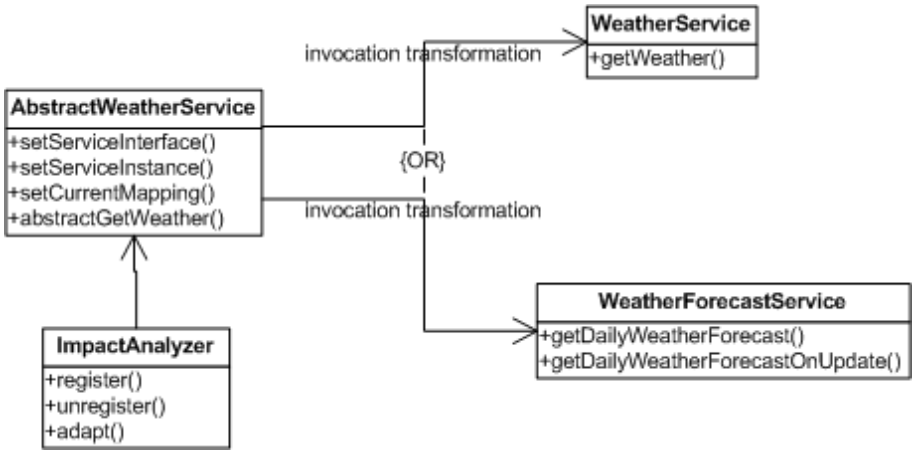


Figure 2.13: A functional abstraction service for the weather services.

services, which are in turn controlled by dedicated impact analyzer services. The impact analyzers do not assume global knowledge of the services involved in a choreography; instead they are responsible for notifying only the registered services. The three main adaptation tasks of the service adaptation process are: the registration of services to the impact analyzers, the reconfiguration of the mappings that are used by the functional abstraction services and the notification of the registered services. The complexity of the first two tasks is constant, while the complexity of the last task is linear to the number of the registered services.

To conclude, the main concepts of the service adaptation discussed in this section are realized as part of the CHOReOS middleware. Further details concerning the technical aspects of the service adaptation concepts are given in D3.2.2 [91].

3 CHOReOS Connectors: Interoperability across Interaction Paradigms

Following the definition of the CHOReOS component model and related service abstractions that enable the assembly of heterogeneous services within service-oriented systems, this chapter focuses on the definition of core **CHOReOS connectors**, that is, the abstraction of the middleware-layer interaction protocols that enable Future Internet services to interact. As outlined in the previous chapter and further reported in former WP1 deliverables [86, 89, 90], the level of heterogeneity in distributed systems has increased dramatically in the recent years and is expected to further increase in the Future Internet context [19]. Indeed, complex distributed applications in the Future Internet are based on the open integration of extremely heterogeneous systems, such as lightweight embedded systems (e.g., sensors, actuators and networks of them), mobile systems (e.g., smartphone applications), and resource-rich IT systems (e.g., systems hosted on enterprise servers and Cloud infrastructures). Such heterogeneity impacts significantly the definition of middleware-layer connectors to be used for the assembly of services within complex distributed systems. The composed services differ in terms of interaction paradigms, communication protocols, and data representation models, which are most often provided by supporting middleware platforms. In particular, with regard to middleware-supported interaction, the client/server (CS), publish/subscribe (PS), and tuple space (TS) paradigms are among the most widely employed ones today, with numerous related middleware platforms. Those paradigms are further evolving towards enabling interaction in the Internet of Things, which bring any Thing in the network, and in particular Sensors and Actuators (S & A) able to interpret and act upon the physical world. Such evolution primarily builds upon the core interaction paradigms CS, PS and TS, while also requiring support for both discrete and continuous interactions over any of those paradigms.

As introduced in Deliverable D1.3 [90], the connector types associated with the CHOReOS architectural style:

(i) Leverage the diversity of interaction paradigms associated with complex distributed systems, and

(ii) Enable cross-paradigm interaction to sustain interoperability in the highly heterogeneous Future Internet.

The former aspect is addressed through the definition of the corresponding connector types. The latter is concerned with solving architectural mismatches arising between connected components, for which we introduce an automated solution so as to enable cross-domain interoperability in truly open and dynamic systems. As surveyed in [47, 86, 46], existing cross-domain interoperability efforts are based on either bridging communication protocols on a pairwise basis or making systems interact through an intermediary reference protocol. The latter is in particular well illustrated by the ESB paradigm that has proven successful for SOA. More precisely, services get connected to the ESB *via* a middleware adapter, which adapts the middleware platform employed by the service to the common bus protocol, and exposes on the bus a SOA interface for the system. However, ESB-based solutions are primarily based on the CS paradigm. In general, state of the art interoperability solutions do not or only poorly address interaction paradigm interoperability, although extensions such as event-driven SOA or ESB supporting the PS paradigm partially tackle the issue. This is why, within CHOReOS, we introduce a new connector type, called *GA connector* (GA stands for "*Generic Application (connector)*), which over-

comes the limitation of today's ESB-based connectors for cross-domain interoperability in the Future Internet.

Based on the above, this chapter details our systematic abstraction approach to interaction protocol interoperability across paradigms. It considerably revises and extends the corresponding same chapter of Deliverable D1.3 [90], while it is written to be self-contained. New contributions of the chapter include the formalization of the base and GA connector types, the enhancement of these connector types with continuous interactions through streaming, the introduction of a new method for actually constructing the GA connector, and the formal verification of GA semantics with respect to base connector semantics. This chapter relies on the following definitions:

1) In Section 3.2, we introduce base CS, PS and TS connector types, which formally characterize today's core interaction paradigms. The proposed types comprehensively cover the essential semantics of the considered paradigms, based on a thorough survey of the related literature and representative middleware instances.

2) In Section 3.3, we abstract further the three core connector types into a single Generic Application (GA) connector type, which unifies the three types while paying particular attention to the preservation of their interaction semantics. As a result, GA connectors support interactions among highly heterogeneous services of the FI, and especially across domains. In particular, we show how the GA connector implements cross-paradigm interoperability according to the semantics of the various paradigms that get composed, by relying on and further extending the method of *protocol conversion via projections* [54]. This paves the way towards automated reasoning about, and fostering of, interoperability between components/services at the middleware layer, with respect to their respective interaction paradigms. In a complementary way, the next chapter deals with interoperability at the choreography level (i.e., application layer) by accounting for the components' respective behavioral specification.our goal

3) Up to this point, we concentrate on abstracting discrete interactions supported by state of the art middleware protocols. However, continuous interactions, denoted by the generic term *streaming*, are increasingly important in the Internet-connected world due to the exchange of content via richer media and further interaction with and within the Internet of Things. Streaming protocols, besides sharing the common characteristic of enabling continuous end-to-end data flows, may have diverse control and data transfer semantics, which they actually borrow from the core interaction paradigms. Hence, we study continuous interactions on top of the base connector types in Section 3.4.

Prior to the specification of the CHOReOS connector types, the next section briefly recalls the main notions underlying the definition of connector types in an architectural style (mostly borrowed from [5, 85, 47]), which was already introduced in Deliverable D1.3 [90], but is briefly repeated here so that the required background is known to the reader and the document is self-contained. The section furthermore introduces the viewpoints from which we are going to specify and analyze connectors in the following sections.

3.1. Background on Connector Formalization

Formally and according to [5], the behavioral semantics of a connector is defined by a set of *role* processes and a *glue* process where:

- The *role* processes (See Figure 3.1, ❶) specify the expected local behavior of each of the interacting parties.
- The *glue* process (See Figure 3.1, ❸) specifies how the behaviors of these parties are coordinated.

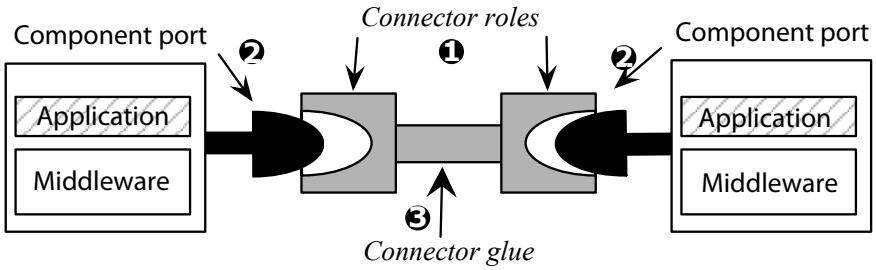


Figure 3.1: Components & Connector

In addition, the way that components use connectors to interact among them are specified by *port* processes (See Figure 3.1, ❷, and Definition 1 in Chapter 2). Then, by specifying the behavior of roles, glues, and ports using some process algebra (e.g., the authors of [83] use FSP processes [63]), architectural matching and thus interoperability may be reasoned upon. Specifically, a component can be attached to a connector only if its port is *behaviorally compatible* with the connector role it is bound to. Allen and Garlan [5] define behavioral compatibility between a component port and a connector role based on the notion of refinement. Informally, a component port is behaviorally compatible with a connector role if the process specifying the behavior of the former refines the process characterizing the latter. In other words, it should be possible to substitute the role process by the port process.

Note that interaction protocols inside distributed systems span both the application and middleware layers, where we are setting our focus above the lower network layers and assume IP-based networking environments. Both the application and middleware layers are sources of heterogeneity. The behavior of a connector may then be defined as a hierarchical protocol that specifies the behavior of the application-layer interaction protocol in terms of middleware-specific protocols [47]. In this chapter, we concentrate on interoperability at the middleware layer, across heterogeneous middleware, and related paradigms. Interaction interoperability at the application layer is addressed at the level of the overall choreography, which is the focus of Chapter 4. Furthermore, specification of port processes for application components is tackled in Chapter 2. Putting everything in one picture, components of Chapter 2 interact via application-layer connectors that coordinate among themselves via choreographies of Chapter 4. This application-layer interaction relies on the underlying middleware-layer connectors of the present Chapter. In particular, the connectors that we elicit in this chapter are *abstract* connectors or *connector types*, since they represent middleware interaction paradigms. Connector types can be refined into concrete connectors modeling existing middleware platforms.

Figure 3.2 depicts our overall approach to interoperability across interaction paradigms, which is a significance over state of the art in the area of middleware interoperability, as existing solutions are limited to interoperability across protocols adhering to the same paradigm. While networked applications (*aka* networked services) *app A*, *app B*, and *app C* rely on legacy interaction protocols (CS, PS, TS connectors) and hence use their associated API to interact with their environment, these legacy protocols are mapped onto associated primitives of the end-to-end GA connector toward sustaining interoperability in the FI without sacrificing the protocols' semantics. Then, internally to the GA connector, CS, PS, TS protocol mismatches are solved based on the *connector converter* element. The next section introduces the base connector types associated with the definition of the core interaction paradigms encountered in today's distributed systems. Then, as already indicated, Section 3.3 introduces the CHOReOS-specific GA connector type, which allows for cross-paradigm interoperability based on the mapping of base connector types to/from GA.

In the next sections, we specify each connector type by addressing several of its aspects from different viewpoints:

1) We begin by *discussing informally the connector type semantics*. Semantics of interest include *space coupling*, *time coupling* and *concurrency*:

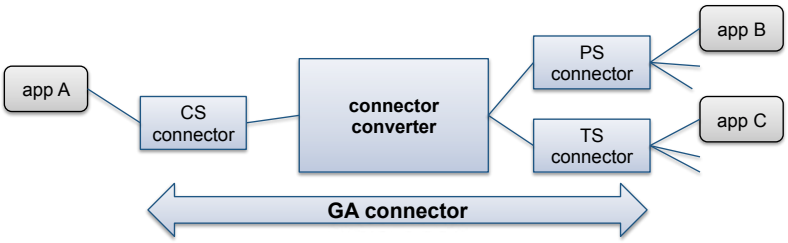


Figure 3.2: GA-based connector interoperability

- *Space coupling semantics* determines how peers interconnected via the connector identify each other and, consequently, how interaction elements (such as messages for a CS connector) are routed from one peer to the other.
- *Time coupling semantics* essentially determines if peers need to be present and available at the same time for an interaction or if, alternatively, the interaction can take place in phases occurring at different times.
- *Concurrency semantics* characterizes the exclusive or shared access semantics of the virtual channel established between interacting peers.

These three categories of semantics are of primary importance, because these are end-to-end connector semantics: when interconnecting different connectors, we seek to map and preserve these semantics across the connectors. In addition to these categories, we discuss *reception semantics*, which has to do with the way a peer receives interaction elements sent by another peer, such as by synchronously polling for these elements or by setting a listening mechanism that will asynchronously notify the arrival of such elements. In general, this last semantics is not end-to-end, i.e., the receiving peer may choose its reception semantics independently of the sending peer and transparently for it.

2) We then *introduce the connector abstract API* (Application Programming Interface). This API presents the programming model supported by the connector and offered to the application components using the connector for their interaction. The objective for the API is to be able to represent a wide-range of middleware platforms that apply the interaction paradigm modeled by the connector. The API is a set of *primitives* expressed as operations or functions supported by the middleware. Certain of these primitives, when executed, provoke the emission of homonymous end-to-end *protocol primitives*, which thus implement the distributed interaction. This abstract API can be refined to a specific middleware platform by mapping primitives and incorporating the data structures and types of the middleware platform. For specifying connector APIs, we use a pseudo-C syntax with the following conventions: (i) functions have no return value; they only have I, O or I/O arguments; (ii) we identify only argument semantic names but not their types; (iii) argument means I, while *argument means O or I/O; (iii) *function() is a pointer to a function.

3) Based on the connector semantics and related API, we *introduce an abstract interface description language* (IDL) for specifying the public interfaces of systems relying on middleware represented by the specific connector. Our IDLs are largely inspired by WSDL, the XML-based IDL for Web services. The objective for an IDL (similar to the API) is to be able to describe a wide-range of systems that are based on the connector. IDLs are specified conceptually, while we have also implemented each one of them as an XML schema document. Based on the flexibility of XML schema, an IDL can be easily refined in order to enable the description of a concrete system that is based on the connector, e.g., we can refine the abstract XML elements into the precise data structures and types of the specific middleware and system. It is further worth highlighting that the IDLs specified for each of the connector types map onto the interface definition for components introduced in Chapter 2 (see Definition 2).

4) Based on the informal identification of semantics in (1), we proceed to *formally specifying the connector behavioral semantics* in terms of role processes and glue process. We specify the behavior of the connector in the FSP process algebra, whose underlying semantics are defined in terms of LTS (Labeled Transition Systems). LTS provide almost self-explanatory graphical representation of processes, hence we will use this representation in the following sections. For specifying connector behavioral semantics in FSP, we use the LTSA verification tool for concurrent systems[1]. From the FSP specification of a process, LTSA produces an LTS graphical representation, which can be used for facilitated inspection of processes that have a small number of states. In our specification of base connectors, we have opted for modeling, in a first step, their essential behavior; this allows keeping the number of states low for the produced LTSs and, as much as possible, grasping their properties by inspection. As part of the connectors' essential behavior, we model time coupling and concurrency semantics, which are of high priority as we argued under above item 1). While we precisely represent and map space coupling semantics in the connectors' APIs and IDLs, we simplify this semantics in the connector behavioral modeling: the elicited LTSs represent already space-coupled interacting peers. As for reception semantics, which are less important, we model only synchronous indefinitely blocking reception. We note as a last point that the actions in the LTSs are labeled according to the primitives of the connector API that they abstract. In particular, the connector role processes specify the right – or accepted by the connector – use of the API. As pointed out above, certain API primitives, when executed, provoke the emission of homonymous end-to-end protocol primitives; this is modeled implicitly or explicitly inside the connector glue. The LTS of a glue process represents precisely the interactions between the API and protocol primitives. This relies on the *parallel composition* of the roles and the glue, which implies that LTS actions labeled the same across the roles and the glue are/can only be executed synchronized in a single transition.

5) Our final analysis consists in*formally verifying the connector behavioral semantics* as specified under item 4). This allows stating the correctness of our connector models with respect to the semantics that they are supposed to have. This further enables identifying the semantics of the GA connector derived from the interconnection of base connectors. Hence, we need to verify the correctness of our connector models with respect to time coupling and concurrency semantics. We formally express and verify time coupling and concurrency semantics in LTL temporal logic. These semantics are expressed as *safety* and *liveness* properties. In general and informally, safety properties express that something bad should never happen in the execution of a system, while liveness properties express that something good should eventually happen. We use, in particular, a version of LTL supported by the LTSA verification tool, called Fluent Linear Temporal Logic (FLTL). We verify our elicited FLTL expressions on LTSA. A quick reference to FLTL is depicted in Figure 3.3.

3.2. Base Connector Types Abstracting Core Interaction Paradigms

This section identifies the three main interaction paradigms used in distributed systems (i.e., CS, PS and TS), and defines the corresponding connector types. The proposed connector types are the outcome of an extensive survey of these paradigms as well as of related middleware platforms in the literature. Our objective is to be able to abstract in each corresponding connector a large number of interaction protocols, as implemented by today's middleware solutions, by a comprehensive set of semantics. While Deliverable D1.3 introduced the connector types mostly informally, this section provides the detailed formal specification for the connectors, which further allows rigorous reasoning about architectural matching, thereby informing the elicitation of the multi-paradigm connector type, GA. In a later step, we extend the proposed connector types to cope with continuous interactions, which are crucial in the context of the Internet of Things. This is presented in Section 3.4.

[1] http://http://ltsa.wikidot.com

Fluent **fluent**	**fluent** FL = <{s1,…sn}, {e1..en}> FL becomes true immediately if any of the initiating actions {s1,…sn} occur and false immediately if any of the terminating actions {e1..en} occur.
Assertion **assert**	**assert** PF = FLTL_Expression defines an FLTL property.
&&	conjunction (*and*)
||	disjunction (*or*)
!	negation (*not*)
->	implication ((A->B)≡(!A || B))
<->	equivalence ((A<->B) ≡(A->B)&&(B->A))
next time **X** *F*	iff *F* holds in the next instant.
always **[]***F*	iff *F* holds now and always in the future.
eventually **<>***F*	iff *F* holds at some point in the future.
until *P* **U** *Q*	iff *Q* holds at some point in the future and *P* holds until then.
weak until *P* **W** *Q*	iff *P* holds indefinitely or *P* **U** *Q*
forall	**forall** [i:R] FL(i) conjunction of FL(i)
exists	**exists** [i:R] FL(i) disjunction of FL(i)

Figure 3.3: A quick reference to FLTL (quoted from [63])

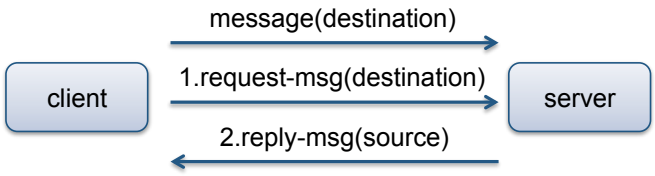


Figure 3.4: CS space and time coupling semantics

3.2.1. CS: Client-Server Connector Type

The *CS connector type* integrates a wide range of semantics, covering both the *direct (i.e., non queue-based) messaging* and *remote procedure call* (RPC) paradigms. In the first case, a single message is sent from the sending entity (client) to the receiving entity (server), while, in the second case, an exchange takes place between the two entities with a request message followed by a reply message; both cases are depicted in Figure 3.4. The CS interaction paradigm is the most widely employed one, e.g., in middleware platforms like Web Services, Java RMI and CORBA.

In terms of *space coupling* semantics between the two interacting entities, CS requires that the sending entity must know the receiving entity and hold a reference of it. With respect to *time coupling* semantics, both entities must be connected at the time of the interaction. Both semantics are represented in Figure 3.4. CS also enables all different kinds of reception semantics: The receiving entity may choose to block its execution, synchronously waiting for the message (as long as it takes or with a timeout), or set up a callback function that will be triggered asynchronously by the middleware when the message arrives. Finally, with respect to *concurrency* semantics, a dedicated virtual channel is used between a sender and a receiver: as long as servers do not have an excessive load of messages to process, all messages sent by different clients will be received by the designated servers (see

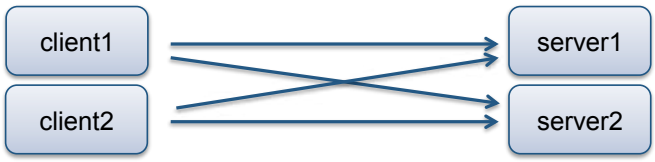


Figure 3.5: CS concurrency semantics

send (destination, operation, input)
*receive_sync (*source, *operation, *input, timeout)*
*receive_async (source, operation, *callback(source, operation, input), *handle)*
end_receive_async (handle)
*invoke_sync (destination, operation, input, *output, timeout)*
*invoke_async (destination, operation, input, *callback(output), *handle)*

Figure 3.6: CS connector API

Figure 3.5).

CS connector API. The above semantics are supported by the CS primitives and their arguments listed in Figure 3.6. These primitives constitute the CS connector API. Functionality of CS primitives is as follows:

- *send* executes the synchronous emission of a message, embedding *operation* name and related *input* parameter, to the *destination*. We note here that both client and server identify an operation served by the server in the same way as *operation(input, output)*.
- *receive_sync* executes the synchronous reception of a single message, which may be waited for until *timeout* expires. *source* and *operation* may be given a value, in which case received messages are filtered, or none; their final values are then returned by the middleware. If operation is two-way, the receiver should reply with *send(source, operation, output)*.
- *receive_async* sets the asynchronous reception of multiple messages and specifies the associated *callback*. *handle* returned by the middleware is a reference that can be used to terminate the reception at will.
- *end_receive_async* closes the channel for asynchronous receipts.
- *invoke_sync* executes a two-way synchronous operation on the client side. Reception of the reply message from *destination* carrying *output* is waited for until *timeout* expires.
- *invoke_aync* executes a two-way asynchronous operation on the client side. Reception of the reply message is done via the callback. The client may use *end_receive_async* to terminate the asynchronous reception before the reply is received.

We note here that the proposed interface considers a single recipient for the message (i.e., *1-1* interaction), while it can be easily generalized to multiple recipients (i.e., *1-n* interaction). Although client-server systems traditionally implement 1-1 interaction schemes, *1-n* interaction schemes become relevant for group communication that is particularly suitable for enforcing non-functional dependability properties.

CS IDL. Based on the CS connector semantics and API, we introduce an abstract IDL for specifying the public interfaces of systems relying on CS middleware. CS-IDL is conceptually presented in Figure 3.7. The *(input/output) message* is considered as the essential interaction element in CS-IDL. The

CS-based service interface		
element	sub-element	attribute
(input/output) message		semantics
		name
		type
main scope of message	service system identity	name
		address type
		address value
sub-scope of message	operation	semantics
		name
		type
		value
interaction semantics of message		{one-way, notification, request-response, solicit-response}

Figure 3.7: CS IDL

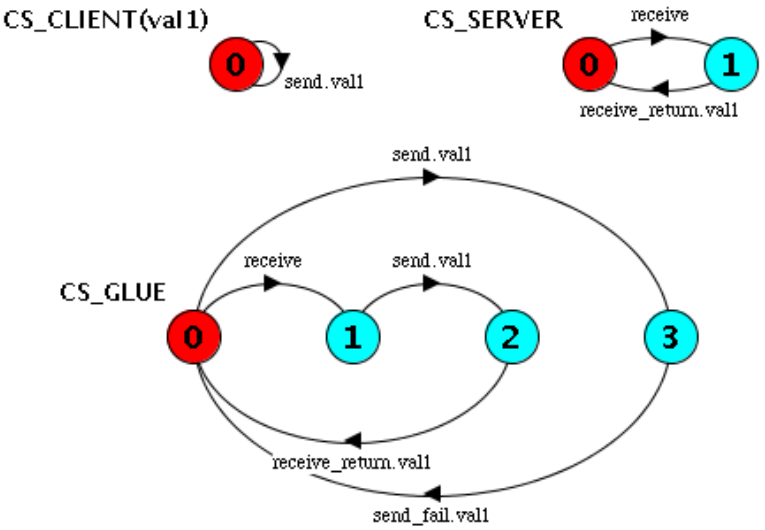


Figure 3.8: CS behavioral semantics for one-way interaction

main new concept here is that a message is assigned two qualifiers, *main scope* and *sub-scope*, which are actually, in inverse order, the operation served by the message and the URL of the service providing the operation. These qualifiers delimit the set of peer entities that will receive the message – actually only one service in the CS system and, more finely, its specific operation.

CS behavioral semantics. The behavioral semantics of the CS connector are depicted in Figures 3.8 and 3.9 in the form of LTS processes for the connector roles and corresponding glue. The former figure specifies one-way interactions, while the latter specifies two-way interactions; both are between one client and one server. The FSP descriptions of the depicted LTSs are parametrizable, so they can easily be tuned to represent more clients and servers with crossing interactions. We model, for the moment, only synchronous indefinitely blocking message reception. We intend to enhance soon the behavioral specification of the CS connector with additional features such as synchronous reception with a timeout or asynchronous via a callback.

Verification of CS semantics. Our goal is to verify the correctness of our elicited CS connector with respect to the space coupling, time coupling, and concurrency semantics that it is supposed to rep-

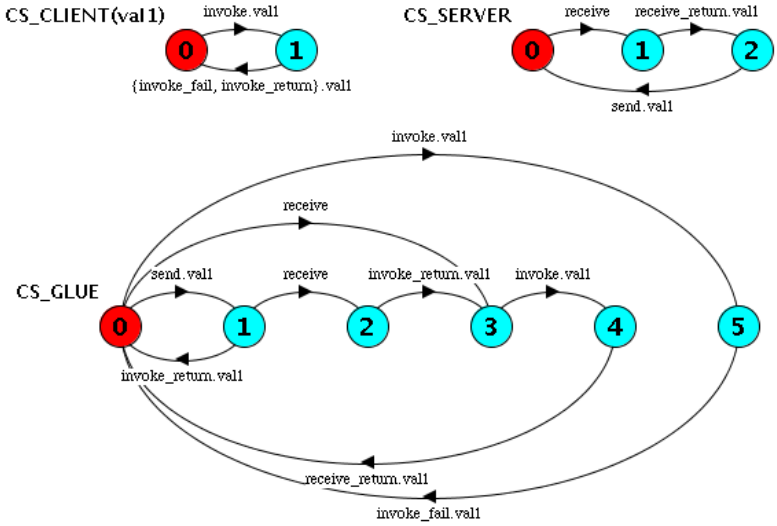


Figure 3.9: CS behavioral semantics for two-way interaction

resent. In our modeling, space coupling semantics can be directly verified, since they are taken for granted: our connectors represent already coupled interacting peers, e.g., in Figure 3.8, we suppose that the client's send has already been correctly routed to the designated server. So, we only need to verify time coupling and concurrency semantics. We formally express CS time coupling and concurrency semantics for one-way interaction as FLTL assertions (see Figure 3.3) below and verify them on LTSA.

Hence, Assertion (3.1) expresses the property that all messages sent by the client are received by the server. This property is violated, since the server may be not be connected (online) at some point. Then, by using Fluent (3.2), which is true when the server is online, we enhance Assertion (3.1) into Assertion (3.3), which additionally requires that the server is online at the time of the interaction. Assertion (3.3) is verified as always true.

$$assert\ SENT\_DATA\_RECEIVED\_ALWAYS = forall[d:DATA]\,[\,]\,(send[d] -> (!send\_fail[d]\ U\ receive\_return[d])) \quad (3.1)$$

$$fluent\ SERVER\_ONLINE = <\{receive\},\{receive\_return[d:DATA]\}> \quad (3.2)$$

$$assert\ SENT\_DATA\_RECEIVED\_IF\_SERVER\_ONLINE = forall[d:DATA]\,[\,]\,((SERVER\_ONLINE\ \&\&\ send[d]) -> (!send\_fail[d]\ U\ receive\_return[d])) \quad (3.3)$$

3.2.2. PS: Publish-Subscribe Connector Type

The *PS connector type* abstracts in a comprehensive way the different types of publish/subscribe systems, such as *queue-*, *topic-* and *content-based systems* [32]. In the PS interaction paradigm, multiple peer entities interact *via* an intermediate *broker* entity. Publishers produce events, which are received by peers that have previously subscribed for receiving the specific events. In *topic-based PS* [64], events are characterized with a topic, and subscribers subscribe to specific topics (see Figure 3.10). In *content-based PS* [25], subscribers provide content filters (conditions on specific attributes of events), and receive only the events that satisfy these conditions. Following the common practice for related middleware platforms [64], we also integrate in our PS model *queue-based messaging*. In this case, an event is sent from a publisher to the queue of a specific subscriber, which may be considered as a constrained case of topic-based PS.

Figure 3.10: PS space coupling semantics

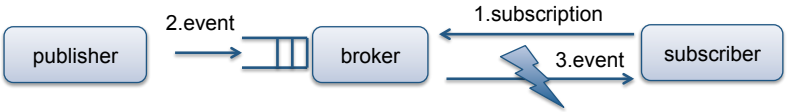


Figure 3.11: PS time coupling semantics

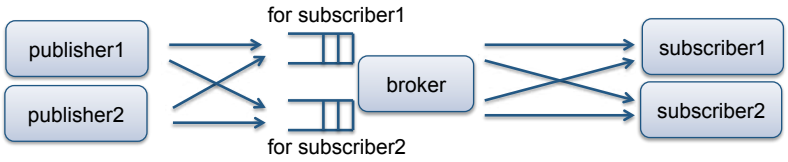


Figure 3.12: PS concurrency semantics

In terms of *space coupling* semantics between interacting peers, in the PS paradigm, peers do not need to know each other or how many they are; e.g., in the case of topic-based systems, events are diffused to subscribers only based on the topic (see Figure 3.10). With the exception of queue-based PS, where a queue belongs to a specific subscriber, and hence the publisher should hold a reference of this queue. With respect to *time coupling* semantics, peers do not need to be present at the same time: subscribers maybe disconnected at the time that events are published; they can receive the pending events when reconnected (see Figure 3.11). Nevertheless, the broker maintains an event until all related subscribers have received it or until the event expires. PS further enables rich reception semantics: Subscribers may choose to check for pending events synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the broker when an event arrives. Finally, with respect to *concurrency* semantics, the broker maintains a dedicated buffer for each subscriber. Hence, unless an event expires, all events sent by different publishers will be eventually received by interested subscribers (see Figure 3.12).

PS connector API. The primitives associated with the above interaction semantics are listed in Figure 3.13, where we represent the notions of queue, topic, and content with the generic *filter* parameter. *filter* can be a value or an expression. In addition, the *lease* parameter stands for the lifetime of the event. More precisely, the PS connector type implements the following primitives:

- *publish* publishes an event that is semantically qualified by *filter* and will be stored by the broker for max *lease* time.
- *subscribe* subscribes for receiving events that are qualified by *filter*. Alternatively, *filter* may be generated by the broker – possibly after negotiation with the subscriber – to qualify the specific subscription, e.g., in the case of a queue allocated for the subscriber. *handle* returned by the broker can be used to uniquely reference the subscription.
- *listen* enables asynchronous reception of multiple events related to the subscription identified by *handle* via the *callback*.
- *get_next* executes synchronous reception of a single event within *timeout*.
- *end_listen* closes a channel of asynchronous event reception.
- *unsubscribe* ends a subscription.

publish (broker, filter, event, lease)
*subscribe (broker, *filter, *handle)*
*listen (handle, *callback(event))*
*get_next (handle, *event, timeout)*
end_listen (handle)
unsubscribe (handle)

Figure 3.13: PS connector API

PS-based service interface		
element	sub-element	attribute
event		semantics
		name
		type
main scope of event	publish-subscribe system identity	name
		address type
		address value
sub-scope of event	filter	semantics; part of semantics is {queue, topic, content}
		name
		type
		value
interaction semantics of event	produce/consume	{publish, subscribe}
	lease	type
		value

Figure 3.14: PS IDL

PS IDL. Similarly to CS-IDL, we introduce an abstract IDL for specifying the public interfaces of systems relying on PS middleware. The difference here is that introduction of open interfaces for PS systems (in the same way as SOA has done for CS systems) is far less developed. We rely on the PS connector semantics and related API, which themselves have been extracted from a wide-range of PS-based systems. PS-IDL is presented in Figure 3.14, while its XML-based implementation allows its refinement for concrete PS-based system, in the same way as for CS-IDL. The essential interaction element in PS-IDL is *event*, and its *main scope* and *sub-scope* are the PS system URL and the *filter*, respectively, used for qualifying the event. In a way similar to CS, these qualifiers delimit the set of peer entities that will receive the event.

PS behavioral semantics. The behavioral semantics of the PS connector are depicted in Figure 3.15. The LTS processes for the PS roles and glue represent the interaction of one publisher publishing same-value events and one subscriber. Actually, they correspond to a single subscription and model the way that this subscription is supported in terms of buffer resources by the broker. The FSP descriptions of the LTS processes are parameterizable, so they can easily be tuned to represent more publishers – publishing different-value events – and subscribers with crossing interactions. We model for the moment, only synchronous indefinitely blocking event reception. Additionally in our modeling, publications are either buffered (buffer size = 1) or not enabled, which is a simple way of emulating an infinite buffer space. We intend to enhance soon the behavioral specification of the PS connector with additional features such as asynchronous event reception.

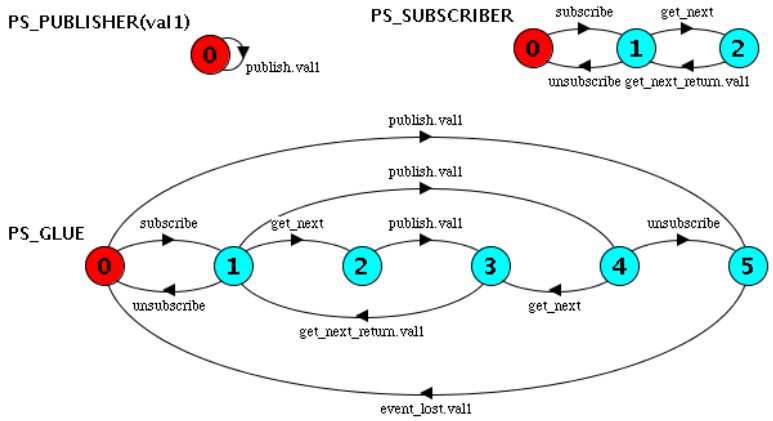


Figure 3.15: PS behavioral semantics

Verification of PS semantics. Similarly to the CS connector, we verify, in the following, time coupling and concurrency semantics for the PS connector. In particular, Assertion (3.4) expresses the property that all events sent by the publisher are received by the subscriber. This property is violated, since the subscriber may not be subscribed before an event is published. Then, by using Fluent (3.5), which is true when the subscriber is subscribed, we enhance Assertion (3.4) into Assertion (3.6), which additionally requires that the subscriber is subscribed at the time of the publication. However, this is still not sufficient, since the subscriber may unsubscribe after the publication and before the reception of the event. Finally, Assertion (3.7), which additionally requires that the subscription is maintained until the reception of the event, is verified as always true.

$$assert\ SENT\_DATA\_RECEIVED\_ALWAYS = forall[d:DATA]\,[\,]\,(publish[d] -> (!event\_lost[d]\ U\ get\_next\_return[d])) \quad (3.4)$$

$$fluent\ SUBSCRIBED = <\{subscribe\},\{unsubscribe\}> \quad (3.5)$$

$$assert\ SENT\_DATA\_RECEIVED\_IF\_SUBSCRIBED\_BEFORE\_PUBLISH = forall[d:DATA]\,[\,]\,((SUBSCRIBED\ \&\&\ publish[d]) -> (!event\_lost[d]\ U\ get\_next\_return[d])) \quad (3.6)$$

$$assert\ SENT\_DATA\_RECEIVED\_IF\_SUBSCRIBED\_BEFORE\_PUBLISH\_AND\_UNTIL\_RECEPTION = forall[d:DATA]\,[\,] (((SUBSCRIBED\ W\ get\_next\_return[d])\ \&\&\ publish[d]) -> (!event\_lost[d]\ U\ get\_next\_return[d])) \quad (3.7)$$

3.2.3. TS: Tuple Space Connector Type

Regarding the base connector type abstraction for data-oriented interactions, we build on the tuple space paradigm, although more specialized than basic shared memory. This is to model rich interaction semantics that is now associated with shared data spaces in distributed systems. The definition of the *TS connector type* is based on the classic tuple space semantics as introduced by the Linda coordination language [37], while it further incorporates a number of advanced features that have been proposed in the literature, such as asynchronous notifications, explicit scoping, and bulk data retrieval primitives.

In the TS interaction paradigm, multiple peer entities interact via an intermediate *shared data space* (see Figure 3.16). Peers can post data into the space and can also synchronously retrieve data from it, either by taking a copy or removing the data. Data take the form of tuples; a tuple is an ordered list of

Figure 3.16: TS space coupling semantics

Figure 3.17: TS time coupling semantics

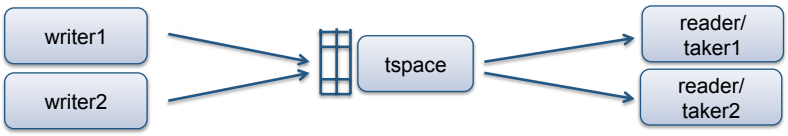


Figure 3.18: TS concurrency semantics

typed elements. Data are retrieved by matching based on a tuple template, which may define values or expressions for some of the elements.

Regarding *space coupling* semantics in the TS paradigm, interacting peers may independently and with no knowledge of each other write and read/take data from the space (see Figure 3.16). As for *time coupling* semantics, TS peers can act without any synchronization. In comparison with PS, peers do not need to subscribe for data, they can retrieve data spontaneously and at any time (see Figure 3.17). Nevertheless, the data space maintains data until they are removed by some peer or until the data expire. With respect to *concurrency*, TS has a number of specificities. In particular, peers have access to a single, commonly shared copy of the data. Additionally, concurrent access semantics of the data space are non-deterministic: among a number of peers trying to access the data concurrently, the order is determined arbitrarily. Hence, if a peer that intends to take specific data is given access to the space before other peers that are interested in the same data, the latter will never access this data. This means that not all data added to the space by different writers eventually reach all interested readers (see Figure 3.18).

In addition to the above semantics, our TS model integrates the following extensions. First, besides synchronous retrieval of tuples, a number of approaches have enabled asynchronous notifications in tuple spaces [21, 36]. More specifically, peers may choose to check for matching tuples synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the data space when matching data appear. This call does not carry the data, possible action of taking or reading the data should be executed by the peer. Second, in distributed realizations of tuple spaces and especially those enabled in mobile environments [67] and/or location-aware environments (e.g., wireless sensor networks) [16], it is important to be able to identify and access only a part of the shared space; this is commonly denoted by the concept of scoping in tuple spaces. Third, several authors have pointed out the *multiple read* problem [78]: if there are multiple tuples matching a read request issued to the tuple space, the tuple retrieved is selected arbitrarily; thus, a sequence of read requests does not guarantee that all the matching tuples will be retrieved. This adds to the uncertainty in data reception due to concurrency as discussed above. Some approaches propose *bulk read* primitives [67, 77] that retrieve all the matching tuples.

TS connector API. API primitives of the TS connector are listed in Figure 3.19 and relate to:

- *out* inserts a *tuple* semantically qualified by *template* into the data space for a max *lease* period. The parameter *extent* is a value or an expression that resolves to a specific part of the data space.
- *take* executes synchronous reception and removal of a single or all tuples matching to *template*,

out (tspace, extent, template, tuple, lease)
*take (tspace, extent, template, policy, *tuple, timeout)*
*read (tspace, extent, template, policy, *tuple, timeout)*
*register (tspace, extent, template, *callback(), *handle)*
unregister (handle)

Figure 3.19: TS connector API

TS-based service interface		
element	sub-element	attribute
tuple		semantics
		name
		type
main scope of tuple	tuple space system identity	name
		address type
		address value
sub-scope of tuple	extent	semantics
		name
		type
		value
	template	semantics
		name
		type
		value
interaction semantics of tuple	produce/consume	{out, take, read}
	consume policy	{one, all}
	lease	type
		value

Figure 3.20: TS IDL

depending on the *policy* specification (which may be *one* or *all*), until *timeout* expires.

- *read* has similar semantics to *take*, but does not remove the tuple(s).
- *register* sets up asynchronous notification for new tuples matching to *template* via the *callback*. The *callback* does not deliver a new tuple, possible action of taking or reading the tuple should be executed by the peer. *handle* returned by the data space is a reference that can be used to terminate the asynchronous notification.
- *unregister* closes an asynchronous notification channel.

TS IDL. The abstract IDL for TS-based systems is depicted in Figure 3.20. Same as for PS-based systems, there are no standard open interfaces for TS systems, hence we rely on the generality of our TS connector semantics and API. We also provide a refinable XML-based implementation of TS-IDL. The essential interaction element in TS-IDL is *tuple*, while its *main scope* and *sub-scope* are the TS system URL and the pair $\{$*extent*, *template*$\}$, respectively, used for qualifying the tuple. Similarly to CS and PS, these qualifiers delimit the set of peer entities that will potentially receive the tuple.

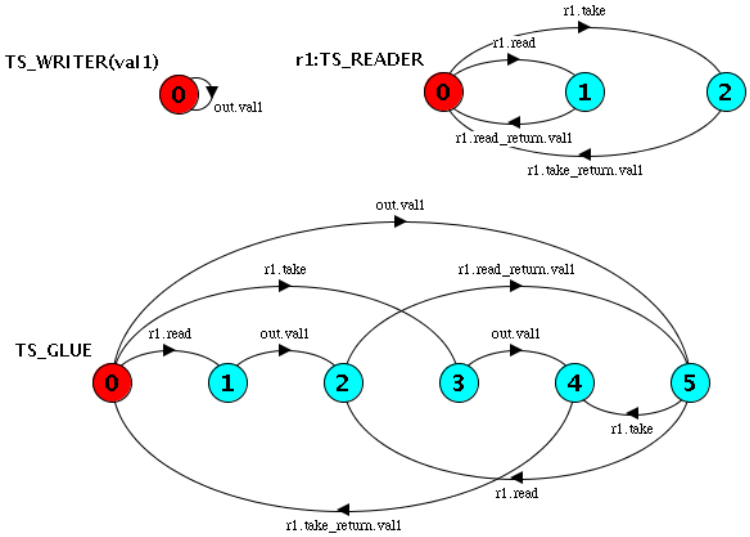


Figure 3.21: TS behavioral semantics

TS behavioral semantics. The behavioral semantics of the TS connector are depicted in Figure 3.21. The LTS processes for the TS roles and glue represent the interaction of one writer and one reader/-taker. Actually, they correspond to a same-value tuple array and model the way that this tuple array is supported in terms of storage resources by the broker. The FSP descriptions of the LTSs are parameterizable, so they can easily be tuned to represent more writers – writing different-value tuples – and readers/takers with crossing interactions. We model, for the moment, only synchronous indefinitely blocking tuple reception. Additionally in our modeling, we use the same simple technique as in the PS connector for emulating an infinite data space: writes are either buffered (buffer size = 1) or not enabled. We intend to enhance soon the behavioral specification of the TS connector with additional features such as asynchronous tuple reception.

Verification of TS semantics. Same as for the two previous connectors, we verify, in the following, TS time coupling and concurrency semantics on a TS connector connecting one writer and two readers/takers. Hence, Assertion (3.8) expresses the property that all data written by the writers are received by all readers. This property is violated, since one of the readers may take the data before some of the other readers read them. We use Fluent (3.9), which is true when there is a pending request for take by some reader. Then, Assertion (3.10) requires that there is no pending take request that precedes a write, and that, once a write is executed, no take request is executed before all readers first read the data. This ensures that all written data are always received by all readers.

$$assert\ SENT\_DATA\_RECEIVED\_BY\_ALL\_ALWAYS \\ = forall[d:DATA]\ [\,]\ (out[d] -> forall[r:READERS] \\ (![READERS].take\_return[d]\ U\ ([r].read\_return[d]\ ||\ [r].take\_return[d]))) \tag{3.8}$$

$$fluent\ PENDING\_TAKE[r:READERS] = <\{[r].take\}, \{[r].take\_return[d:DATA]\}> \tag{3.9}$$

$$assert\ SENT\_DATA\_RECEIVED\_BY\_ALL\_IF\_ALL\_READ\_BEFORE\_TAKE \\ = forall[d:DATA]\ [\,]\ (((forall[r:READERS]\ !PENDING\_TAKE[r])\ \&\&\ out[d]\ \&\& \\ (forall[r:READERS]\ (![READERS].take\ U\ [r].read\_return[d]))) -> \\ forall[r:READERS]\ (![READERS].take\_return[d]\ U\ ([r].read\_return[d]\ ||\ [r].take\_return[d]))) \tag{3.10}$$

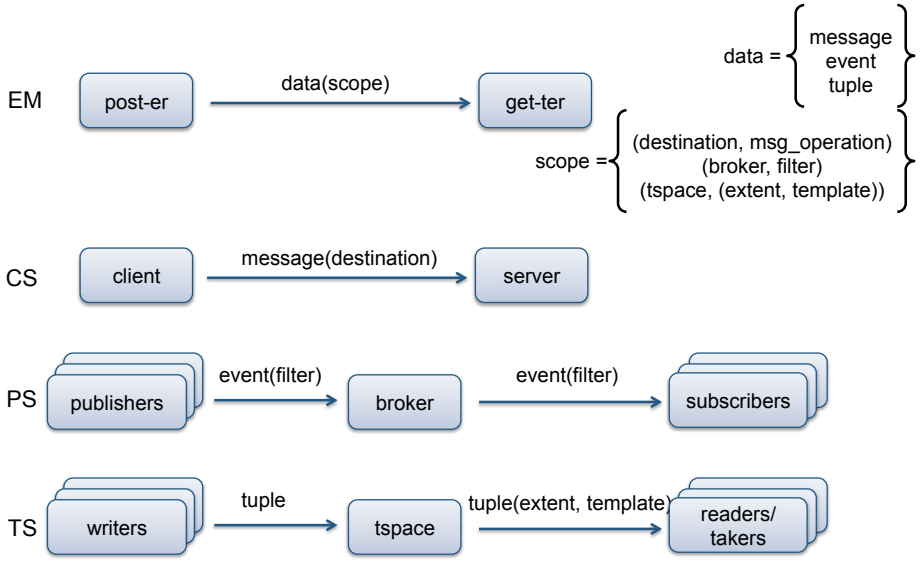


Figure 3.22: GA space coupling semantics wrt. CS, PS, TS space coupling semantics

3.3. GA: Generic Application Connector Type

Given the three base middleware-layer connector types defined in the previous section, we now introduce the CHOReOS Generic Application (GA) connector type. Our objective is to devise a single generic connector that comprehensively represents the end-to-end cross-paradigm/domain interaction semantics of application entities that employ different base (middleware) connectors.

With respect to the service bus paradigm, which has already proved successful toward sustaining interoperability in SOA, our goal is to introduce an intermediary reference protocol (e.g., an ESB's common bus protocol) that leverages the richness of today's interaction paradigms, as opposed to constraining interaction to a single (principally CS) paradigm. Further, as already stressed, it is central that the proposed reference protocol allows for cross-paradigm interoperability.

We identify two main high-level API primitives for the GA connector:

1) A *post()* primitive employed by a peer for sending data to one or more other peers (i.e., production of information), and

2) A *get()* primitive employed by a peer for receiving data (i.e., consumption of information).

For example, a PS *publish()* primitive can be abstracted by a *post()*.

Based on these high-level primitives and on the detailed analysis of the three base connector types in the previous section, we can already identify *space coupling* semantics for the GA connector by appropriately mapping among the space coupling semantics of the base connectors. This is depicted in Figure 3.22. There are two important elements in the GA space coupling semantics:

1) We define the essential interaction element for GA to be *data*. *data* can represent any one of CS' *message*, PS' *event* or TS' *tuple*.

2) We introduce the explicit scoping parameter *scope* to generalize addressing for the different interaction paradigms. We analyze *scope* as {*main scope, sub-scope*}. Then, for the CS connector, this maps to {*destination/source, operation*}, for PS, it maps to {*broker, filter*}, and for TS, to {*tspace,* {*extent, template*}}. In practice, *scope* enables restricting the entities that have access to the data conveyed in a sender/post-er – receiver/get-ter interaction by qualifying the data. For example, we map the *destination* parameter of CS *send()* to *main scope*; thus, the *input* data

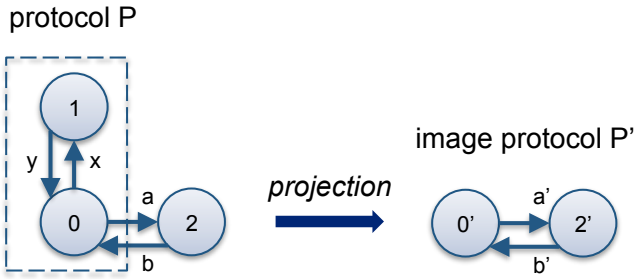


Figure 3.23: Protocol projection

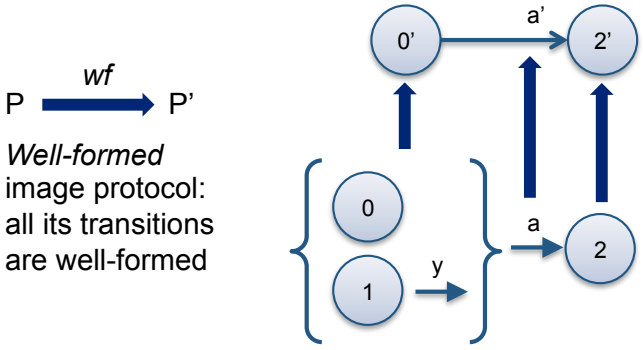


Figure 3.24: Well-formed image protocol

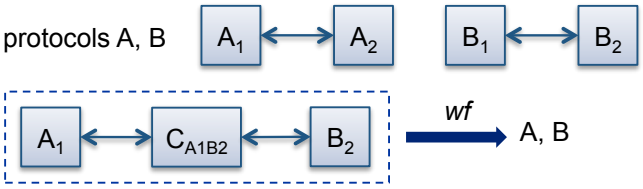


Figure 3.25: Protocol conversion properties

carried by *send()* will be accessed only by the single CS destination peer (in another example, the *filter* parameter of PS mapped to *sub-scope* both specifies addressing and qualifies the event).

In order to identify the complete semantics of GA (in particular, identify time coupling and concurrency semantics in addition to the introduced space coupling semantics) and construct a conversion mechanism among the heterogeneous base connectors (see Figure 3.2), we need first to briefly introduce in the next section the method of *protocol conversion via projections* [54], on which we build.

3.3.1. Protocol Conversion via Projections

A protocol P is projected to an *image protocol* P' by abstracting away certain protocol states (fusing with others) and transitions (fusing or eliminating). A simple example of protocol projection for a protocol P is depicted in Figure 3.23. We additionally require that the image protocol P' is *well-formed*, that is, all its transitions should be well-formed. This is exemplified in Figure 3.24 for the transition a' of protocol P'. If P' is a well-formed image protocol, *safety* and *liveness properties* of P' apply, *properly inversely projected*, also to P. We already saw in the previous sections that time coupling and concurrency semantics of our connectors can be expressed as such properties.

Let A and B be two protocols where $A = A_1A_2$ and $B = B_1B_2$, i.e., each one is produced as the parallel composition of two peer protocol entities (see Figure 3.25). If a converter C_{A1B2} can be constructed such that A and B are well-formed images of $A_1C_{A1B2}B_2$, properties of A and properties of

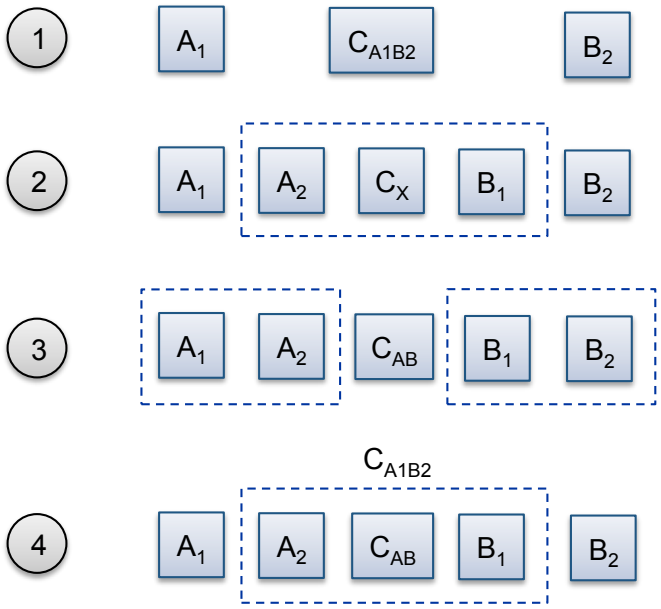


Figure 3.26: Extension of the conversion method

B apply, properly inversely projected, also to $A_1 C_{A1B2} B_2$. This can be used to verify properties of the conversion.

Extension of the conversion method. Construction of a converter, as required by the projection method, is not straightforward. Additionally, applying projection and checking well-formedness is tedious for protocols with many states. We introduce an extension to the method, where we guide the method based on our knowledge of the end-to-end protocols A and B. Our extension to the method introduces the following steps, as depicted in Figure 3.26:

1) We are looking for the converter C_{A1B2}.
2) A good guess for the interfaces of the converter towards A_1 and B_2 are the corresponding peer protocol entities A_2 and B_1.
3) In this way, the initial problem is transformed into finding C_{AB}. C_{AB} essentially does data flow mapping between A and B, and is quite straightforward to build. Additionally, verifying properties of C_{AB} by projection is simpler.
4) We can show that $C_{A1B2} = A_2 C_{AB} B_1$ is a solution according to the method of protocol conversion via projections.

Application of the conversion method to GA. We construct $C_{CS-PS-TS}$, which does data flow mapping between any two of CS, PS, TS, as shown in Figure 3.27. Then, according to the conversion method via projections, for the interconnection of, e.g., the applications *app A* and *app B*, the end-to-end interaction protocol is the common image protocol of CS and PS. This applies equally to the other cases of interconnection, namely, between *app A* and *app C* as well as between *app B* and *app C*. Based on this, the resulting GA is the union of the common image protocols of all pair combinations of CS, PS and TS.

We show in Figures 3.28 and 3.29 the elicited partial behaviors of $C_{CS-PS-TS}$ for the interactions CS-to-PS and PS-to-TS, respectively. Then, the partial behaviors of the GA connector are produced by the parallel composition of the interconnected base connectors with the appropriate converter, hence:

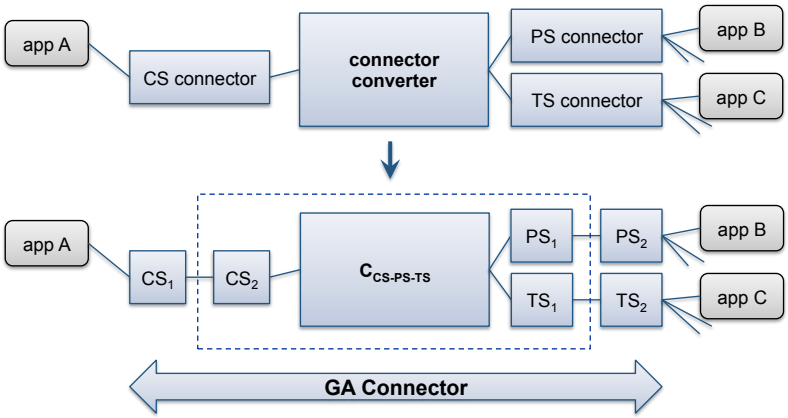


Figure 3.27: Application of the conversion method to GA

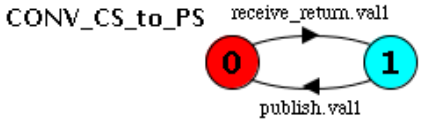


Figure 3.28: Partial converter for the interaction CS-to-PS

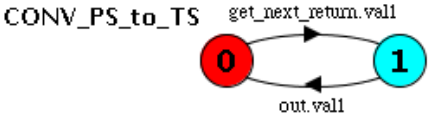


Figure 3.29: Partial converter for the interaction PS-to-TS

$$CS \,||\, C_{CS-PS} \,||\, PS, \quad PS \,||\, C_{PS-CS} \,||\, CS$$
$$CS \,||\, C_{CS-TS} \,||\, TS, \quad TS \,||\, C_{TS-CS} \,||\, CS$$
$$PS \,||\, C_{PS-TS} \,||\, TS, \quad TS \,||\, C_{TS-PS} \,||\, PS$$

The resulting GA connector for the interaction CS-to-PS has 48 states and the GA PS-to-TS connector has 72 states, which makes their LTS diagrams too hard to display and read, hence we don't show them here.

Tuning of the conversion method for GA. According to the projection method, properties of CS, PS and TS, inversely projected, apply to GA. This means that they apply *to the corresponding half* of the end-to-end interconnection. Hence, (only) common properties of CS, PS, TS apply end-to-end. As shown in the previous sections where the base connectors were introduced, time coupling and concurrency semantics of CS, PS and TS are not directly compatible. In particular, we saw that for successful interaction, for CS, CS server must be online (Property 3.3), while for PS, an ongoing subscription is necessary (Property 3.7), and for TS, all interested peers must be allowed to read the common target data before one of the peers takes them (Property 3.10). This means that, e.g., *app A* and *app B*, if interconnected, may perceive different semantics, which can be problematic for the composed application.

The solution is to constrain the semantics of the heterogeneous connectors to a compatible subset, *by application-side enforcement* via parallel composition with a *component port* (see Section 3.1). Thus, to verify whether the conversion preserves end-to-end the semantics required by the two applications, we should include their enforcing behaviors (the related component ports) in the GA connector

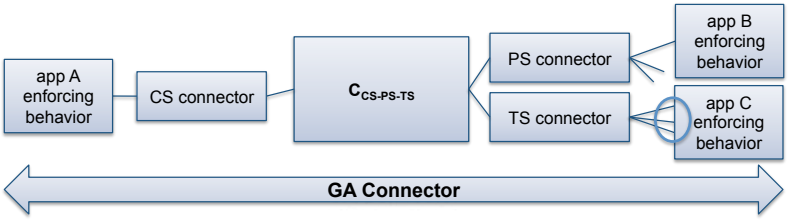


Figure 3.30: Tuning of the conversion method for GA

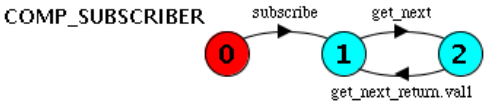


Figure 3.31: PS component port refining the subscriber role

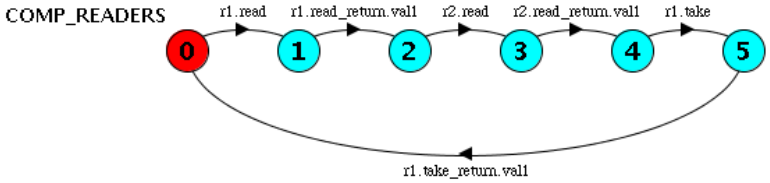


Figure 3.32: TS component port refining two reader roles

(see Figure 3.30). Particularly for TS applications, the behaviors of all the users of the tuple space are potentially relevant (see Property 3.10), due to the TS concurrency semantics. According to the projection theory, it is then sufficient to verify that, with the application-side enforcement, the new individual semantics for CS, PS and TS are now compatible; in this case, they apply to the *whole* end-to-end GA interconnection.

Regarding CS, no enforcing behavior can really be applied by a CS application, other than timely going online and minimizing the time between reception of a message and being available again, e.g., by having sufficient message processing resources running in threads separate from the receiving thread. However, this cannot be expressed in terms of our CS connector model. As for PS, we show in Figure 3.31, a component port for a subscriber role that subscribes and then never unsubscribes. This somehow enhances the *receive-always* semantics of the PS connector. The resulting PS connector is slightly different from the original one; it is depicted in Figure 3.33. Regarding TS, Figure 3.32 depicts a component port that coordinates two reader roles so that both readers read data one after the other before one of them takes the data. This ensures *receive-always* semantics of the TS connector. The original TS connector for one writer and two readers has 18 states (too big to be usefully depicted here). When composing with the port, the resulting TS connector is reduced to 8 states and depicted in Figure 3.34.

We now verify the new time coupling and concurrency semantics for the constrained PS and TS connectors. Property (3.6) now holds for the PS connector, while property (3.8) holds for the TS connector. Property (3.3) still characterizes the CS connector. Hence, if the CS, constrained PS and constrained TS connectors are interconnected, the end-to-end *receive-always* semantics of the resulting GA connector is now improved with respect to the unconstrained connectors. More specifically, sent data on the GA connector are always received if the CS server is online and the PS subscriber timely subscribes. We note that the resulting – slightly constrained – GA connector for the interaction CS-to-PS still has 48 states, while the constrained GA PS-to-TS connector (for two readers) is considerably reduced from 216 to 68 states.

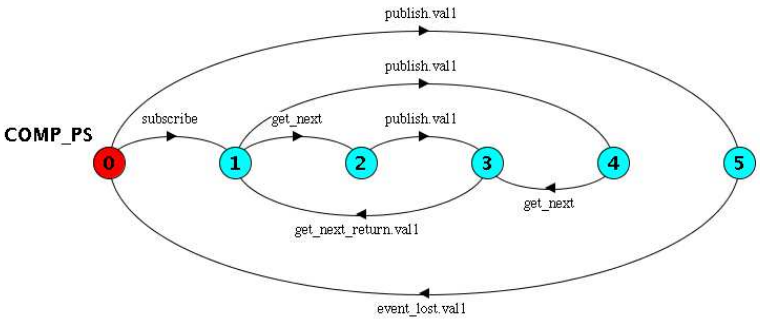


Figure 3.33: PS connector constrained by component port

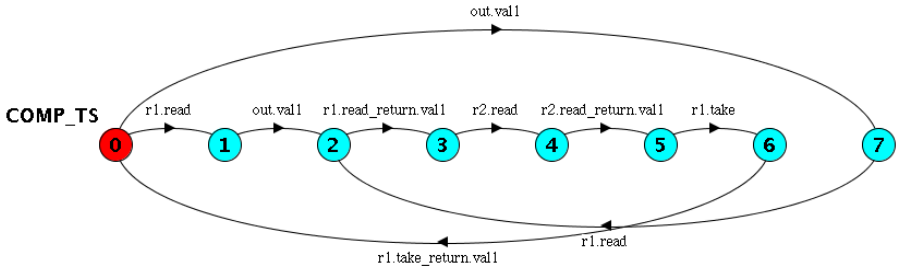


Figure 3.34: TS connector constrained by component port

3.3.2. Completing the Specification of the GA Connector

Based on the previous analysis, end-to-end time coupling and concurrency semantics of GA result from the common corresponding semantics of CS, PS and TS. CS is the more restrictive of the three paradigms, while PS and TS allow more flexibility to the application. Hence in the case of PS and/or TS connectors interconnected with a CS connector, the PS and/or TS applications should take care of enforcing the additional behavioral semantics, if these are required by the CS application. While each case should be treated individually, we can state in general that in a CS-PS-TS interconnection, the resulting end-to-end GA semantics are the ones of CS.

In more detail, for successful GA one-way interaction (i.e., CS-PS-TS one-way interaction with appropriate application ports), with regard to time coupling, a GA peer should in general be 'ready' (online for the CS case, subscribed for the PS case), and with regard to concurrency semantics, an exclusive virtual channel is put in place end-to-end between two GA peers. Reception semantics can be any of the ones already supported by CS, PS, and TS. Finally, the CS two-way interaction – as employed by CS and if properly supported by the PS and TS applications – should also make part of the GA semantics.

GA connector API. Based on the above elicited GA semantics and by mapping among the common API semantics of the CS, PS and TS connectors, we can elicit a generic API for the GA connector, as depicted in Figure 3.35.

GA IDL. Same as for the GA API, based on mapping among the IDLs of the base connectors, we elicit the IDL for the GA connector as shown in Figure 3.36.

GA behavioral semantics. We already identified informally in the previous the behavioral semantics of the GA connector. We show the resulting GA connector roles in Figures 3.37 and 3.38. As already discussed in Section 3.3.1, the GA connector glue is formally produced by the parallel composition of the interconnected base connectors with the appropriate converter; the resulting GA processes specifying behavioral semantics are too big to display and read.

post (scope, data)
*get_sync (*scope, *data, timeout)*
*get_async (scope, *callback(scope, data), *handle)*
end_get_async (handle)
*post_get_sync (scope, post_data, *get_data, timeout)*
*post_get_async (scope, post_data, *callback(get_data), *handle)*

Figure 3.35: GA connector API

GA-based service interface		
element	sub-element	attribute
data		semantics
		name
		type
main scope of data	system identity	name
		address type
		address value
sub-scope of data	data qualifier(s)	semantics
		name
		type
		value
interaction semantics of data		{post, get, post-get, get-post}

Figure 3.36: GA IDL

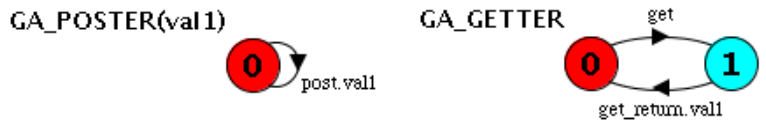


Figure 3.37: GA connector roles for one-way interaction

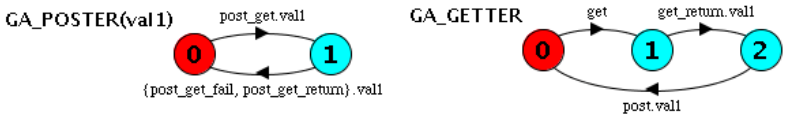


Figure 3.38: GA connector roles for two-way interaction

Verification of GA semantics. As already discussed in Section 3.3.1, and according to the projection theory, to verify the connector GA semantics, it is sufficient to verify them on its projections to each one of the interconnected base connectors. We indeed reported there on the verification of Properties (3.1-3.3), (3.4-3.7), (3.8-3.10) on CS, PS, TS, correspondingly. In this section, we confirm this by verifying GA semantics on the actual end-to-end GA connector for the interconnections CS-to-PS and PS-to-TS, where we consider both the original and constrained CS, PS and TS connectors .

In particular, Assertions (3.11-3.13) express *receive-always* semantics for the end-to-end interaction CS-to-PS, by adding together conditions regarding CS behavior and PS behavior. We verify that Assertion (3.12) holds if PS constrained and that only Assertion (3.13) holds if PS not constrained (CS is not constrained anyway). This is in conformance with the results of Section 3.3.1.

Furthermore, Assertions (3.14-3.16) express *receive-always* semantics for the end-to-end interaction

PS-to-TS, by adding together conditions regarding PS behavior and TS behavior. We verify that Assertion (3.15) holds if PS and TS are constrained and that only Assertion (3.16) holds if PS and TS are not constrained. This again is in conformance with the results of Section 3.3.1.

$$assert\ SENT\_DATA\_RECEIVED\_ALWAYS\ =\ forall[d:DATA]\ [\ ]\ (send[d]\ ->\ ((!send\_fail[d]\ U\ receive\_return[d])\ \&\&\ (!event\_lost[d]\ U\ get\_next\_return[d])))\tag{3.11}$$

$$assert\ SENT\_DATA\_RECEIVED\_IF\_SERVER\_ONLINE\_AND\_SUBSCRIBED\_BEFORE\_PUBLISH\ =\ forall[d:DATA]\ [\ ]\ ((SERVER\_ONLINE\ \&\&\ SUBSCRIBED\ \&\&\ send[d])\ ->\ ((!send\_fail[d]\ U\ receive\_return[d])\ \&\&\ (!event\_lost[d]\ U\ get\_next\_return[d])))\tag{3.12}$$

$$assert\ SENT\_DATA\_RECEIVED\_IF\_SERVER\_ONLINE\_AND\_SUBSCRIBED\_BEFORE\_PUBLISH\_AND\_UNTIL\_RECEPTION\ =\ forall[d:DATA]\ [\ ]\ ((SERVER\_ONLINE\ \&\&\ (SUBSCRIBED\ W\ get\_next\_return[d])\ \&\&\ send[d])\ ->\ ((!send\_fail[d]\ U\ receive\_return[d])\ \&\&\ (!event\_lost[d]\ U\ get\_next\_return[d])))\tag{3.13}$$

$$assert\ SENT\_DATA\_RECEIVED\_BY\_ALL\_ALWAYS\ =\ forall[d:DATA]\ [\ ]\ ((publish[d]\ ->\ (!event\_lost[d]\ U\ get\_next\_return[d]))\ \&\&\ (get\_next\_return[d]\ ->\ <>\ out.[d])\ \&\&\ (out[d]\ ->\ forall[r:READERS]\ (![READERS].take\_return[d]\ U\ ([r].read\_return[d]\ ||\ [r].take\_return[d]))))\tag{3.14}$$

$$assert\ SENT\_DATA\_RECEIVED\_BY\_ALL\_IF\_SUBSCRIBED\_BEFORE\_PUBLISH\ =\ forall[d:DATA]\ [\ ]\ (((SUBSCRIBED\ \&\&\ publish[d])\ ->\ (!event\_lost[d]\ U\ get\_next\_return[d]))\ \&\&\ (get\_next\_return[d]\ ->\ <>\ out.[d])\ \&\&\ (out[d]\ ->\ forall[r:READERS]\ (![READERS].take\_return[d]\ U\ ([r].read\_return[d]\ ||\ [r].take\_return[d]))))\tag{3.15}$$

$$assert\ SENT\_DATA\_RECEIVED\_BY\_ALL\_IF\_SUBSCRIBED\_BEFORE\_PUBLISH\_AND\_UNTIL\_RECEPTION\_AND\_IF\_ALL\_READ\_BEFORE\_TAKE\ =\ forall[d:DATA]\ [\ ]\ ((((SUBSCRIBED\ W\ get\_next\_return[d])\ \&\&\ publish[d])\ ->\ (!event\_lost[d]\ U\ get\_next\_return[d]))\ \&\&\ (get\_next\_return[d]\ ->\ <>\ out.[d])\ \&\&\ (((forall[r:READERS]\ !PENDING\_TAKE[r])\ \&\&\ out[d]\ \&\&\ (forall[r:READERS]\ (![READERS].take\ U\ [r].read\_return[d])))\ ->\ forall[r:READERS]\ (![READERS].take\_return[d]\ U\ ([r].read\_return[d]\ ||\ [r].take\_return[d]))))\tag{3.16}$$

3.4. Streaming Connector Types

The previous sections have introduced the thorough definition of the base connector types associated with interaction paradigms relied upon in today's distributed systems together with that of a new multi-paradigm connector to sustain interoperability in the Future Internet. However, we have so far abstracted the fact that interactions may be discrete as well as continuous. Indeed, while discrete interactions have for long been predominant within distributed systems, the increasing connectivity of digital systems together with the richness of the content made available over the Internet call for continuous interactions. This is getting even more important in the Internet of Things context where applications handle data coming from the physical world. A key characteristic of these data is that they keep changing over time and hence require continuous handling, as for instance exemplified by applications in the area of traffic management or of warehouse logistics [17]. Those applications typically handle data

streams as opposed to discrete data sets commonly encountered in the classical Internet. Those data streams, which are sequences of structured data, are continuously consumed by the applications without being able to anticipate when the sequence ends. Henceforth, those applications can not store the complete data streams and must therefore apply real time computation on the data as they are received [39]. As a result, standard data models and associated data management algorithms are not adequate to handle stream-based data and need to be revisited. Indeed, on the one hand, under standard data models, finite data sets are considered to be persistent while applied computations are considered to be volatile as they are no longer pertinent once the data set has been manipulated. On the other hand, data streams being data sets whose size is theoretically infinite, data lose its persistent nature (data is meaningful only at the time it is produced) while data requests are persistent due to their continuous execution. Just like Data Base Management Systems (DBMS) are the systems used to store and handle finite data sets, Data Stream Management Systems (DSMS) are the systems allowing to manage sets of data streams and apply computation on them [9]. The following section provides a brief overview of the state of the art DSMS, so that the readers get familiar with operations associated with data stream management. Section 3.4.2 then revisits the connector types introduced in the previous sections so that both discrete and continuous interactions are captured.

3.4.1. Background on Data Stream Management

We identify quite a few DSMSs in the literature; they were originally introduced in the context of Wireless Sensors and Actuators Networks (WSAN) and may be classified into three broad families, which are respectively based on: (i) the relational model, (ii) macroprogramming, and (iii) the SOC paradigm. Within CHOReOS, we are more specifically interested in the third category of DSMSs. However, we provide an overview of the 3 categories in what follows as they all introduce important notions toward offering streaming protocols that are well suited for the FI, and especially its Internet of Things subset.

The *relational DSMS*s that extend the relational model typically add concepts necessary to handle data streams and persistent queries, together with the stream-oriented version of the relational operators (selection, projection, union, etc.). The sensor network is then handled as a large database, distributed or centralized depending on the specific approach [43], on which queries are executed. Queries typically comply with the SQL formalism with some stream-specific operations. From a practical perspective, queries are translated into query plans that are distributed in the network. State of the art DSMSs primarily differ with respect to: the expressiveness of the query language, the associated algebra and assumptions made about the underlying networking architecture. TinyDB [62] exposes the measures sensed by the network as a relation (i.e., table) on which it is possible to apply queries over the sensed values as well as the metadata associated with the sensors. During the handling of queries, all the nodes execute the queries that are distributed in the network and the results of each query get aggregated as they traverse the routing tree maintained by the system. In the same vein, Cougar [98] acts as a database of sensors, which gathers the sensed values as temporal data series, as well as the metadata about the sensors themselves (position, feature, etc.). The query plans are provided to proxies that take care of activating the relevant sensors and applying the operations on the collected data. MaD-WiSe [6, 7] offers a runtime system for queries that is fully distributed, and each sensor may directly execute part of a query plan and then deal with sensor-specific tasks. Instead of relying on traditional requests, Aurora [2] uses data streams diagrams, which express the combination of relational operators over the streams received by the system. Aurora was originally centralized and was later revised into a distributed system named Borealis [1], which introduces proxies that receive queries and execute them with respect to a given set of sensors. From a theoretical perspective, various systems propose custom extension to the relational model as well as custom implementations of the relational operators. For instance, STREAM [9] distinguishes streams from relations, where the latter can be handled by classical relational operators. New operators then allow dealing with translation from stream to relations (typically using windows), and vice versa (using streamers). Other proposals [20, 29, 10, 49, 27, 26, 17, 52, 57] deal with issues as diverse as blocking and non-blocking

operators, windows, stream approximation, and optimizations.

Macroprogramming-based DSMS are oriented toward the development of applications over WSAN, as opposed to the expression of data queries over the network. The macroprogrammes are typically specified using a domain-specific language, and are compiled into microprogrammes to be run on the networked nodes, hence easing the developer's tasks who has no longer to bother with the decomposition and further distribution of the macroprogrammes. Macroprogramming-based DSMSs are overall similar to classical macropogramming approaches aimed at WSAN. Still, they feature additional primitives and mechanisms oriented toward stream management. For instance, Regiment [69, 68] introduces a functional language that enable programming the WSAN and manipulating the streams that flow in the network. As for Semantic Streams [97], it defines a declarative language based on Prolog, which features data structures to handle streams, together with mechanisms to reason about the semantics of sensors. For instance, the system is able to compose or adapt data according to the available sensors and the provided data request.

Finally, *service-oriented DSMSs* aim at integrating with classical service-oriented architectures, thereby allowing to exploit the functionalities of the infrastructure (interaction and discovery protocols, registries, service composition based on orchestration or choreography, security infrastructure, etc.). Similarly to database-oriented relational DSMSs, the simplest service-oriented DSMSs are centralized with a unique point of data collection [38, 41, 28, 94], or semi-distributed based on a set of data collection points [23, 71, 56]. Practically, most approaches adopt RESTful services and expose the various sensors as resources identified using URIs that are more or less complex and whose parameters define the transformations to be applied on the produced data. For instance, WebPlug [71] introduces extensions to URIs so that they allow querying about the resources metadata (history, connected consumers, last access, etc.). At the architectural level, the integration of data streams with the Web is achieved using well-proven technologies. In particular, we identify work on the implementation of streaming aimed at Web services, both for (i) the RESTful architecture using HTTP-specific mechanisms like Web hooks or long polling [95], and (ii) the WS$*$ architecture through the addition of new Message Exchange Patterns (MEP) customized for stream-based communication like, e.g., receiving multiple requests and producing multiple responses in parallel as part of a single invocation [53]. Regarding the paradigms used to broadcast streams, they vary from one solution to another. Stream Feeds [28] uses pull requests to gather historical data and push requests to received new data issued by the sensors. RMS [94] goes a step further by building upon a topic-based pub/sub infrastructure, while WebPlug [71] uses an infrastructure based on pollers that periodically check on the state change of resources. Some approaches consider using existing streaming protocols like RTP [3] and RTSP (for multimedia streaming), messaging protocols XMPP [45], or languages that allow expressing queries over XML streams [70]. Other work focuses on exploiting semantic Web technologies, in particular extending the SPARQL language to process continuous queries over RDF data [17], or syndication [95]. For instance, LSM [56] brings semantics to streams by using an ontology for WSN, which integrates the types of sensed data as well as sensor features (mobility, power, calibration, etc). However, the complexity of the proposed technologies requires significant adaptation so that they can be hosted by tiny, wireless sensors. Significant research effort is ongoing [59, 79, 31, 33, 30, 82, 66, 61, 14] to bring advanced connectivity to those resource-constrained devices by providing lightweight implementations of the technologies composing the Web (HTTP client and server, TCP/IP, Web services, etc.) but this is often at the expense of flexibility (static pages, only a subset of functionalities is provided, offloading of computation, etc.) [31].

In general, the work focused on WSAN and the one oriented toward the Web of Things remain largely distinct, from both a theoretical and a technical perspective. From the technology standpoint, there are, on the one hand, highly constrained devices (RFID chips [13], low capacity embedded sensors, various routing and communication protocols [65], etc.) and, on the other hand, things (or smart things) that are potentially able to handle more complex protocols [13] and to interface with more flexible but also more resource-consuming infrastructures. From a theoretical standpoint, the rich WSAN-related work on DSMSs has given rise to formal algebra and advanced data model, featuring various non-

blocking operators as well as probabilistic operators [27, 26], which allow developers to overcome the uncertainty associated with the data streams originating from sensors (transient errors) or to deal with approximation over streams. The theoretical work on the Web of Things is much weaker, as the effort is mostly focused on leveraging available, standardized Web technologies and to adapt them to the requirements posed by streaming. One of our goals within CHOReOS is to reconcile these two lines of work on DSMS, i.e., WSAN-oriented and Thing-oriented, to offer a powerful DSMS for the Internet of Things, as part of the CHOReOS middleware; this issue is more specifically investigated within WP3. From a conceptual perspective, the CHOReOS connectors must leverage discrete as well as continuous protocols. This leads us to enrich the definition of the CHOReOS connector types introduced in the previous section with their streaming counterpart.

3.4.2. STR$*$: Streaming Connector Types

In the following, we elicit the abstract streaming connector type STR supporting continuous interactions and then propose three different realizations of such continuous interactions on top of the discrete interactions supported by the base connector types CS, PS, and TS. In this way, we produce three streaming connector types STR_CS, STR_PS and STR_TS, correspondingly. We specify the API of each new connector type.

STR connector. The STR connector abstracts common semantics widely found in data streaming protocols and related middleware platforms. A stream is a continuous flow of data from a producer to a consumer employing a logical channel between the two entities. This channel needs to be established before the flow of data can start. Typically, it is the consumer that establishes this channel with a related request sent to the producer. Accordingly, it is again the consumer that decides when to close the channel with another related request to the producer. A stream is identified by the pair $<$*producer, stream_id*$>$, i.e., the name or address of the producer and an id of the stream unique for the specific producer. We note here that we have opted for representing with the STR connector only data communication semantics of streaming protocols and middleware platforms. Other features found in data streaming, such as continuous queries, compression and windowing mechanisms, can be added on top of the stream communication semantics of the STR connector.

The primitives of the abstract streaming connector type STR are listed in Figure 3.39. These primitives constitute the STR connector API. Functionality of STR primitives is as follows:

- *open_stream* is executed by the consumer in order to establish a streaming channel between the *producer* and itself. Stream *stream_id* is requested to be streamed.
- *send_item* is executed by the producer to send an *item* of stream *stream_id* to the *consumer*. An item is the smallest piece of information that has a meaning for the consumer, e.g., a picture, a sensor reading, etc.
- *receive_item_sync* and *receive_item_async* are used by the consumer to receive one and several *item*s of stream *stream_id* synchronously and asynchronously, correspondingly. Then, *end_receive_item_async* ends an asynchronous reception.
- *close_stream* is executed by the consumer in order to close a previously established streaming channel.

We show in the following how the STR API primitives can be implemented on top of the CS, PS, and TS primitives, as part of the derived streaming connector types STR_CS, STR_PS, and STR_TS, correspondingly.

```
open_stream(producer, stream_id)
send_item(consumer, stream_id, item)
receive_item_sync(producer, stream_id, *item, timeout)
receive_item_async(producer, stream_id, *callback(producer, stream_id, item), *handle)
end_receive_item_async(handle)
close_stream(producer, stream_id)
```

Figure 3.39: STR connector API

```
open_stream(producer, stream_id)
        = send(producer, open_operation, stream_id)
send_item(consumer, stream_id, item)
        = send(consumer, stream_id, item)
receive_item_sync(producer, stream_id, *item, timeout)
        = receive_sync(producer, stream_id, *item, timeout)
receive_item_async(producer, stream_id, *callback(producer, stream_id, item), *handle)
        = receive_async(producer, stream_id, *callback(producer, stream_id, item), *handle)
end_receive_item_async(handle)
        = end_receive_async(handle)
close_stream(producer, stream_id)
        = send(producer, close_operation, stream_id)
```

Figure 3.40: STR_CS connector API realization

STR_CS connector. The STR_CS connector employs the one-way primitives and semantics of the CS connector to implement the STR streaming semantics. The STR_CS connector API realization is depicted in Figure 3.40. Then, STR primitives are implemented as follows:

- *open_stream* and *close_stream* are implemented as one-way CS messages conveying the operations *open_operation* and *close_operation* with parameter *stream_id*.
- The rest of the STR primitives are directly implemented with the corresponding CS primitives.

STR_PS connector. The STR_PS connector enables stream producers - publishers and stream consumers - subscribers to interact through the PS broker. PS connector semantics apply to this interaction, hence, a consumer can request a stream produced by a specific producer only indirectly, i.e., the $<$*producer, stream_id*$>$ identification of a stream is mapped to a PS topic. Likewise, a producer sends out (publishes) a stream without having been previously requested to by a specific consumer: eventually, the interested subscribed consumers will receive this stream. The STR_PS connector API realization is depicted in Figure 3.41. Then, STR primitives are implemented as follows:

- *open_stream* and *close_stream* are implemented with the *subscribe* and *unsubscribe* PS primitives, correspondingly.
- *send_item* is implemented with the *publish* PS primitive, where no *consumer* is identified any more, and the producer inserts its name into the topic along with the *stream_id*.
- *receive_item_sync*, *receive_item_async* and *end_receive_item_async* are implemented with the *get_next*, *listen* and *end_listen* PS primitives, correspondingly.

open_stream(producer, stream_id)
*= subscribe(broker, <producer, stream_id>, *handle)*
send_item(consumer, stream_id, item)
= publish(broker, <producer, stream_id>, item, lease)
*receive_item_sync(producer, stream_id, *item, timeout)*
*= get_next(handle, *item, timeout)*
*receive_item_async(producer, stream_id, *callback(producer, stream_id, item), *handle)*
*= listen(handle, *callback(item))*
end_receive_item_async(handle)
= end_listen(handle)
close_stream(producer, stream_id)
= unsubscribe(handle)

Figure 3.41: STR_PS connector API realization

open_stream(producer, stream_id)
= void
send_item(consumer, stream_id, item)
= out(space, producer, stream_id, item, lease)
*receive_item_sync(producer, stream_id, *item, timeout)*
*= read/take(space, producer, <stream_id, new_tuples>, all, *item, timeout)*
- new_tuples = forall tuple in space such as timestamp(tuple) >timestamp(last_reading)
*receive_item_async(producer, stream_id, *callback(producer, stream_id, item), *handle)*
*= register(space, producer, stream_id, *callback(), *handle)*
*- upon callback(): read/take(space, producer, <stream_id, new_tuples>, all, *item, timeout)*
end_receive_item_async(handle)
= unregister(handle)
close_stream(producer, stream_id)
= void

Figure 3.42: STR_TS connector API realization

STR_TS connector. The STR_TS connector enables stream producers - writers and stream consumers - readers to interact through the TS space. TS connector semantics apply to this interaction, hence, a reader can access a stream produced by a specific producer only indirectly, i.e., the *<producer, stream_id>* identification of a stream is mapped to the TS *<extent, template>* pair. Likewise, a producer sends out (writes) a stream without having been previously requested to by a specific consumer: eventually and depending on TS read/take coordination semantics, the interested readers - consumers will receive this stream. The STR_TS connector API realization is depicted in Figure 3.42. Then, STR primitives are implemented as follows:

- *open_stream* and *close_stream* are void, since access to a stream via the space is open to all space users. Alternatively, some application-specific streaming setup protocol can easily be implemented between a producer and a consumer via the space by writing and reading some agreed signal.
- *send_item* is implemented with the *out* TS primitive, where no *consumer* is identified any more, and the producer inserts its name into the *<extent, template>* pair along with the *stream_id*.
- *receive_item_sync* is implemented with a *read* or *take* TS primitive, depending on the specific streaming application semantics. The *template* argument of *read* or *take* takes the composite

value <*stream_id, new_tuples*>, which enables retrieving only the tuples of the stream that arrived after the last reading. This is further accomplished by setting (when supported by the space) the *policy* argument of *read* or *take* to *all*, which enables retrieving all the relevant tuples.

- *receive_item_async* is implemented with the *register* TS primitive. When *callback()* is triggered – since it does not deliver any data – the consumer should execute a *read* or *take* in the way described in the previous bullet.
- Finally, the *end_receive_item_async* is implemented with the *unregister* TS primitive.

As we can see from the above, the STR_CS, STR_PS and STR_TS connectors all implement the STR streaming semantics, while each one of them adds to this semantics the specific semantics of the underlying base connector, i.e., CS, PS and TS, correspondingly. In the same way, we can implement the STR connector on top of the GA connector, which can represent any of the CS, PS, and TS connectors; this is pretty direct, so we do not present this here. Hence, interoperability between the STR_CS, STR_PS, and STR_TS connectors can be studied in the same way as for the base connectors. We intend to complete this study in the last months of the CHOReOS project.

Concluding this chapter, we note that our base middleware connector types CS, PS, TS, as well as our middleware interoperability method based on the GA connector type, have already been applied to the *eXtensible Service Bus (XSB)*, which is one of the main components of the CHOReOS middleware architecture developed in WP3, as can be found in Deliverable D3.1 and the subsequent WP3 deliverables. In particular, we have implemented our interoperability solution on top of the EasyESB enterprise service bus, by enhancing the binding components and common protocol of the bus, which are typically Web services-oriented, with our connectors, which enable semantics-preserving interaction between services employing heterogeneous middleware platforms. Furthermore, we have applied the XSB solution to the WP6 *Passenger Friendly Airport* use case choreography, and in particular to the *Unexpected arrival warning* and *Unexpected arrival handling* sub-choreographies, which can be found in Deliverable D6.2, in order to support services that run on top of publish-subscribe and tuple space middleware for the M24 related demo. We are now concentrating on extending the CHOReOS XSB with streaming, as well as on developing a custom DSMS for the FI, both based on the findings outlined in Section 3.4.2 and leveraging relevant state of the art solutions.

4 CHOReOS Coordination Protocols: Abstracting Choreography Behavior

In this chapter, accounting for the definitions of CHOReOS *components* and *connectors* of the previous chapters, we complete the definition of the final version of the CHOReOS architectural style by adding the notion of **CHOReOS coordination protocol** that abstracts choreography behavior. Specifically, the CHOReOS *coordination protocol* introduces a higher, application-layer connector that defines system-wide behavior, based on the connection of CHOReOS components through middleware-layer connectors introduced in the previous chapters.

This version of the CHOReOS coordination protocol extends and finalizes the previous version presented in Deliverable D1.3 [90]. In particular, the notion of coordination protocol has been extended by adding new coordination primitives to be able to handle the much more powerful coordination logics "implied by" complex BPMN2 choreography specifications. Moreover, the specific architectural style constraints (to be imposed on the choreography-based system to suitably coordinate the discovered services) have been refined to enable the full automation of the synthesis process, worked out in WP2 during the second year of the project.

Keeping the main structure of the homonymous chapter in Deliverable D1.3 [90], this chapter is structured as follows: Section 4.1 recalls the problem of choreography-based coordination in the FI that we solve in CHOReOS, and relates it with the *realizability check* and *conformance check* problems that are usually considered in the literature. Then, building upon our previous results sketched in Deliverable D1.3 [90], Section 4.2 provides the formal abstractions that CHOReOS uses for choreography-based coordination. Finally, Section 4.3 concludes the chapter by illustrating the application of the synthesis process to the CHOReOS airport use case that is investigated in WP6.

4.1. Choreography-based Coordination in the FI

When considering choreography-based service-oriented systems, the following two problems are usually considered: (i) *realizability check* - checks whether the choreography can be realized by implementing each participant service so that it conforms to the played role; and (ii) *conformance check* - checks whether the set of services satisfies the choreography specification or not. In the literature many approaches have been proposed to address these problems (e.g., see [18, 22, 44, 74, 80, 76, 40, 42]). However, by moving a step forward with respect to the state of the art, a further problem worth considering when actually realizing service choreographies by possibly reusing (third-party) services concerns *automatic realizability enforcement*. That is, given a choreography specification and a set of existing services discovered as suitable participants (by exploiting the service abstractions associated with CHOReOS components, as defined in Chapter 2, and the discovery process defined in Deliverable D2.2 [92]), restrict the interaction among them so as to fulfill the collaboration prescribed by the choreography specification. This requires to extract from the choreography specification the global coordination logic to be then distributed and enforced among the participants.

In this sense, it is worth to note that realizability for BPMN2 choreographies, as presented in the

Object Management Group (OMG)[1] standard, is not an issue per se. In fact, the standard specification gives the modeling conditions that must be respected by the choreography designer in order to ensure realizability "by construction". Such well-formedness rules were also advocated in other works, e.g., [24]. In other words, assuming the adoption of a *generative approach*, the notions of realizability and its enforcement do not represent an issue if, starting from a choreography specification, the goal is to implement a set of services to realize the choreography. That is, if all the modeling conditions prescribed by OMG are fulfilled, the composed services can be in principle implemented so that realizability is ensured by construction.

However, as described in Deliverable D2.2 [92], CHOReOS considers a broader scenario where the services to be used to realize the choreography are either dynamically discovered within the Future Internet, or implemented by adopting a generative test driven development approach (if suitable services cannot be found into the service registry). This means that, when the services are being discovered, we consider the local behavior of each participant, as extracted from the choreography by an end-point projection technique we have implemented in CHOReOS. Then, exploiting the CHOReOS simulation technique (implemented as part of the synthesis processor), the end-point projection of each participant is used to find services whose behavior simulates the roles to be played by the participant. The simulation technique is based on the notion of strong simulation [15] thoroughly extended to deal with the extended Labeled Transition Systems (LTSs) we use in CHOReOS. However, as detailed in Deliverable D1.3 [90], although services may have been discovered as suitable end-points to realize all the participant roles of the specified choreography, their composite interaction may prevent the choreography realizability if left uncontrolled (or coordinated in a wrong way).

In this direction, CHOReOS proposes a solution for realizability enforcement by leveraging model-based methodologies and relevant SOA standards, while making *choreography* development a systematic process based on the reuse and the assembly of services discovered within the Future Internet. In fact, CHOReOS revisits the concept of choreography-based service-oriented systems and introduces a model-based development process and associated methods, tools, and middleware for *coordinating* services in the Future Internet. Since a choreography is a network of collaborating services, the notion of coordination protocol becomes crucial. In fact, it might be the case that the collaborating services, although potentially suitable in isolation, when interacting together can lead to *undesired interactions*. These are interactions that do not belong to the set of interactions modeled by the choreography specification and can happen when the services collaborate in an uncontrolled way. To prevent undesired interactions, we automatically synthesize additional software entities, called *Coordination Delegates* (CDs), and interpose them among the participant services. CDs coordinate the services' interaction in a way that the resulting collaboration realizes the specified choreography. As explained in the next section, this is done by exchanging suitable *coordination information* that is automatically generated out of the choreography specification.

4.2. Formal Abstractions for FI Choreography-based Coordination

This deliverable thoroughly refines and extends preliminary results introduced in Deliverable D1.3 [90] by describing how to extract from a BPMN2 choreography diagram an extended automata-based specification of the coordination logic "implied" by the choreography. Specifically, the extension of LTSs (called *Choreography LTS - CLTS*), preliminarily introduced in deliverable D1.3 [90] and thoroughly refined in this deliverable, provides an explicit description of the coordination logic that must be applied to enforce the choreography. The refined extended CLTSs represent an intuitive, yet powerful, means to precisely describe the complex coordination logics implied by BPMN2 choreography specifications. In particular, the initial version of CLTS presented in [90] has been extended with additional constructs to handle more complex constructs of BPMN2 Choreography Diagrams, e.g., conditional exclusive gateways (decision, alternative paths), inclusive gateways (inclusive decision, alternative but also parallel

[1] http://www.omg.org/spec/BPMN/2.0/

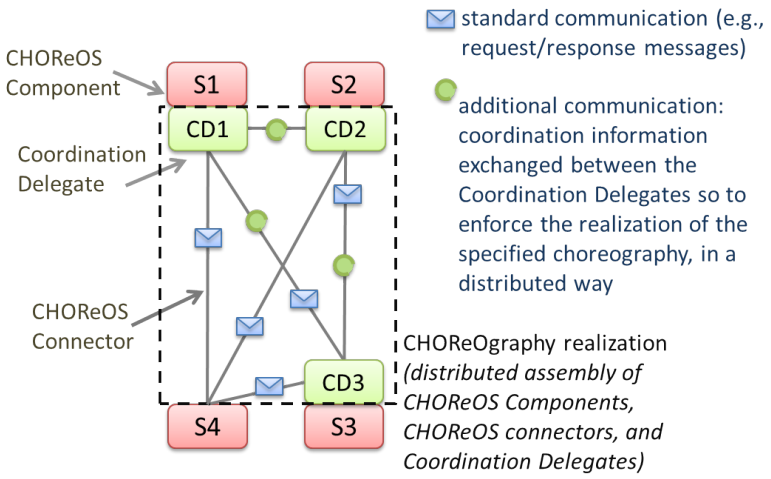


Figure 4.1: CHOReOS choreography

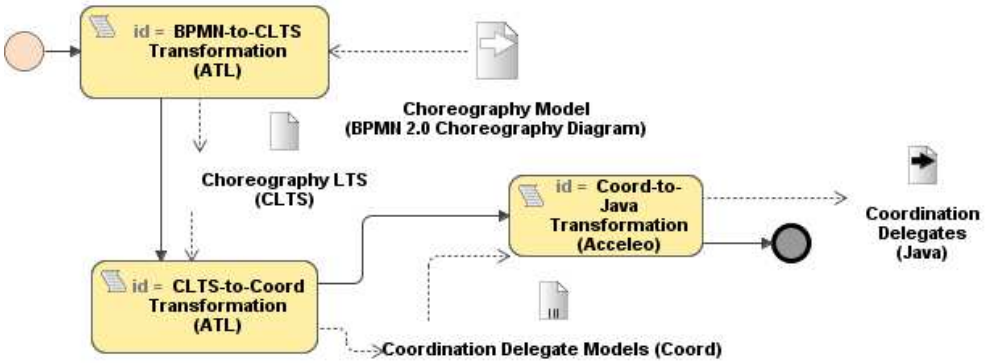


Figure 4.2: CHOReOS synthesis process

paths), parallel gateways (creation and merging of parallel flows), standard and sequential loops.

The approach to abstract the coordination logic out of a BMPN2 specification, as extended in this deliverable, constitutes the basis of the final methodology we are applying in CHOReOS to solve the problem of realizability enforcement. In fact, as already introduced in Deliverable D1.3 [90], for the choreography to be externally enforced, the coordination logic modeled by the CLTS is distributed between the CDs, whose goal is to coordinate (from outside) the interaction of the participant services in a way that the resulting collaboration realizes the specified choreography. In CHOReOS, we automatically synthesize the CDs and, in line with the initial idea sketched in Deliverable D1.3 [90], we interpose them among the participant services according to the CHOReOS architectural style (see Figure 4.1). CDs perform pure coordination of the services' interaction in a way that the resulting collaboration realizes the specified choreography. To this aim, the coordination logic is distributed among a set of *Coordination Models* that codify coordination information. Then, at run time, the CDs exploit the coordination information contained in these models to enforce the coordination of the external interaction of the services participating to the specified choreography in order to prevent possible undesired interactions. The latter are those interactions that do not belong to the set of interactions allowed by the choreography specification and can happen when the discovered services collaborate in an uncontrolled way.

4.2.1. From Choreography Specification to Choreography-based Coordination

In this section we outline the choreography synthesis process and the related model transformations (as enhanced with respect to Deliverable D2.2 [92]), devoted to the extraction of the coordination logic.

As shown in Figure 4.2, the choreography synthesis process mainly consists of three model trans-

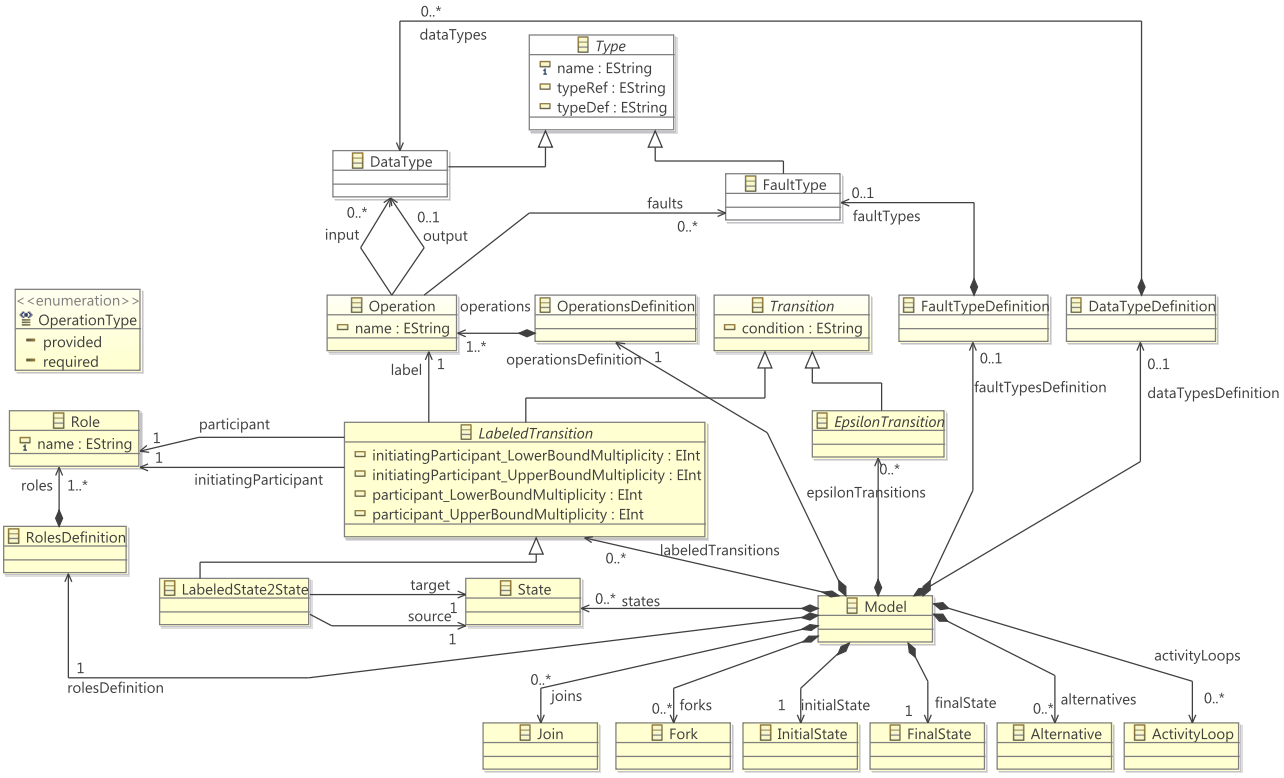


Figure 4.3: CLTS metamodel

formations. Before describing these model transformations, we introduce the *CLTS metamodel* and *Coord metamodel* the transformations rely on. The former has been defined for extending LTSs, and constitutes the foundation of the coordination logic extracted by the synthesis process. The latter, has been defined for modeling the coordination information contained within the Coordination Models.

The high-level view of the CLTS metamodel is shown in Figure 4.3. In brief, the metamodel extends the basic notion of state by introducing new elements to model complex states, i.e., `initial` and `final` states, `fork` and `join` states, as well as, `activity loop` and `alternative` states. The basic notion of labeled transition has been extended to have the possibility of specifying `participants roles`, `service operations request/response/fault` messages and related `types`, as well as, `conditions`.

The Coord metamodel is shown in Figure 4.4. It has been designed to define models containing coordination information codified as a set of tuples. In particular, for a tuple (see the metaclass `Tuple`), a coordination model specifies states, participants, allowed operations and conditions (see the metaclasses `State`, `Participant`, `AllowedOperation`, and `Condition`, respectively). Interestingly, each tuple has corresponding *wait* and *notify* specifications, each consisting of a number of elements (see the metaclasses `WaitElement` and `NotifiedElement`, respectively). For each coordination delegate, the tuples specify which operations are allowed, from which states to which states, the conditions to be checked, and the other coordination delegates to be notified or to be waited for.

We now review each model transformation applied throughout the CHOReOS development process.

BPMN-to-CLTS. By means of transformation rules implemented through the ATLAS Transformation Language (ATL) [50], the BPMN2 specification is transformed into an equivalent (modulo coordination logic) CLTS specification. As detailed in Section 4.2.2, the ATL transformation takes as input a BPMN2 choreography model conforming to the BPMN2 metamodel and produces a CLTS model conforming to the CLTS metamodel of Figure 4.3. In the current implementation we transform BPMN2 choreog-

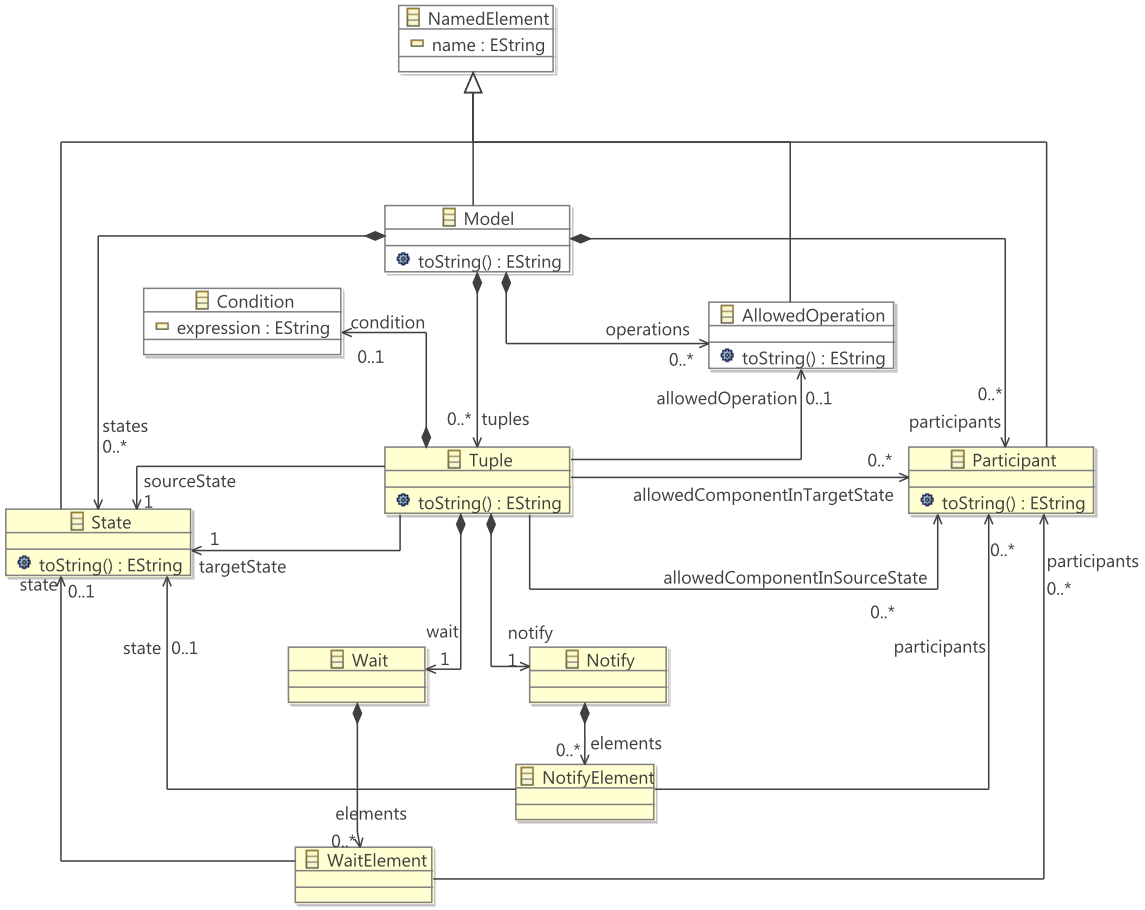


Figure 4.4: Coord metamodel

raphy specifications created with the Eclipse BPMN2 modeler[2], whose metamodel is compatible with the BPMN 2.0 specification. However, we are currently studying the transformation of BPMN2 choreography specification exported by the MagicDraw modeling tool[3]. In Deliverable D1.3, we used a lighter extension of LTSs, hence handling simpler BPMN2 choreography specifications, with respect to the ones we are able to handle now. The novel and much more expressive CLTS metamodel we present here allows us to fully automate the approach and to transform very complex choreography specifications into powerful coordination logics.

Note that, the CLTS model specifies the coordination logic implied by BPMN2 choreography specifications at the level of choreography tasks, hence being independent from the concrete services that actually play the participant roles of the task. Then, as recalled in the next transformation, the participant roles are used to discover service abstractions and related concrete services to be coordinated by the choreography.

The `BPMN-to-CLTS` transformation is further detailed in Section 4.2.2.

CLTS-to-Coord. An ATL transformation is defined to automatically distribute the CLTS into a set of coordination models conforming to the metamodel shown in Figure 4.4.

A Coord model M_{CD_i}, for a coordination delegate CD_i, specifies the local information that CD_i needs to know in order to properly cooperate with the other CDs in the choreography-based system. The aim of this cooperation is to prevent undesired interactions in the global collaboration of the participant services, hence enforcing choreography realizability in a distributed way. Concerning coordination in-

[2] http://eclipse.org/bpmn2-modeler/
[3] http://www.nomagic.com/

formation, in Deliverable D1.3 [90] we only considered basic coordination information that allowed us to solve race conditions deriving from unconditioned branching behaviors outgoing from simple LTS states. As already introduced, we are now able to automatically distribute more expressive coordination logics and to automatically derive the related set of coordination models. More specifically, coordination models can now contain refined and extended coordination information, as a set of tuples, that allow for coordinating more complex service interactions with, e.g., high degrees of parallelism, complex joins, conditional branching and looping (standard, sequential and parallel), together with request/response/-fault operations' messages, without introducing any centralized information flow. Moreover, the new graphical representation of the novel CLTS model, together with the human-readable plain-text format of the tuples that codify coordination information, provides the user with an intuitive means to precisely understand the not-easy-to-grasp coordination logic "hidden" into BPMN2 Choreography Diagrams. In this vein, it is important to stress that a clear understanding of the coordination logic implied by a choreography specification is of paramount importance and constitutes the basis of the notion of distributed enforcement as intended by the OMG BPMN 2.0 specification.

Although not in the focus of this deliverable, with reference to Deliverable D2.2 [92], it is worth to mention that the set of coordination models are derived after a set of services have been discovered as suitable services to play the roles of the choreography participants. Specifically, the discovery process returns, for each choreography participant, a set of candidate services. Then, for a given participant, to select the proper service among the set of candidate services returned by the discovery, an extra step is required. This step is performed by the synthesis processor, and checks if, within the set of service candidates, there is a service whose behavior is suitable to play the role of the participant. We further recall that to check for suitability, the synthesis processor applies a strong simulation algorithm to verify if the extended LTS, achieved by projecting the overall choreography CLTS to the given participant, can be simulated by the extended LTS specifying the behavior of the service as returned by the discovery process. To this end, the basic notion of strong simulation (and related algorithm) has been extended to deal with extended LTSs. Having selected abstractions for each role (as described in Chapter 2), the service implementations are decoupled from the actual dependency endpoints by declaring dependencies on roles (rather than on implementation) and by setting the actual invocation address at runtime (by invoking the *setInvocationAddress*() functionality offered by the Enactment Engine described in deliverable D3.2.2 [91]). A participant can then be bound to a functional abstraction that represents a set of concrete services. Binding a participant to a functional abstraction enables the adaptation of the actual concrete service that is represented by the functional abstraction as detailed in Deliverable D3.2.2 [91].

The `CLTS-to-Coord` transformation is further detailed in Section 4.2.3.

Coord-to-Java. The Coord model specifies the logic that a CD has to perform independently from any target technology. In CHOReOS, we have chosen Java as a possible target language of our Acceleo[4]-based model-to-code transformation. The Java implementation of a delegate CD_i exploits the information contained into its Coord model M_{CD_i}. Briefly, for each Coord model a Java class is generated by means of dedicated templates consisting of static and variable parts. The latter are fixed by means of the information retrieved from the source Coord model. The transformation `Coord-to-Java` is not in the focus of this deliverable, thus it is not discussed further. Still referring to Deliverable D3.2.2 [91], we just recall that the CDs are deployed on and run within the *Coordination Delegates Component* (which is part of the *eXecutable Service Composition* component of the CHOReOS middleware).

4.2.2. From BPMN2 to CLTS

In the cells of Figures 4.5 and 4.6, the left part contains sample BPMN2 choreography source elements and the right part contains the corresponding CLTS target elements. The developed ATL transformation consists of a number of rules, each devoted to the management of specific source BPMN2 modeling elements. Then, these rules are suitably combined to transform complex choreographies obtained as

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

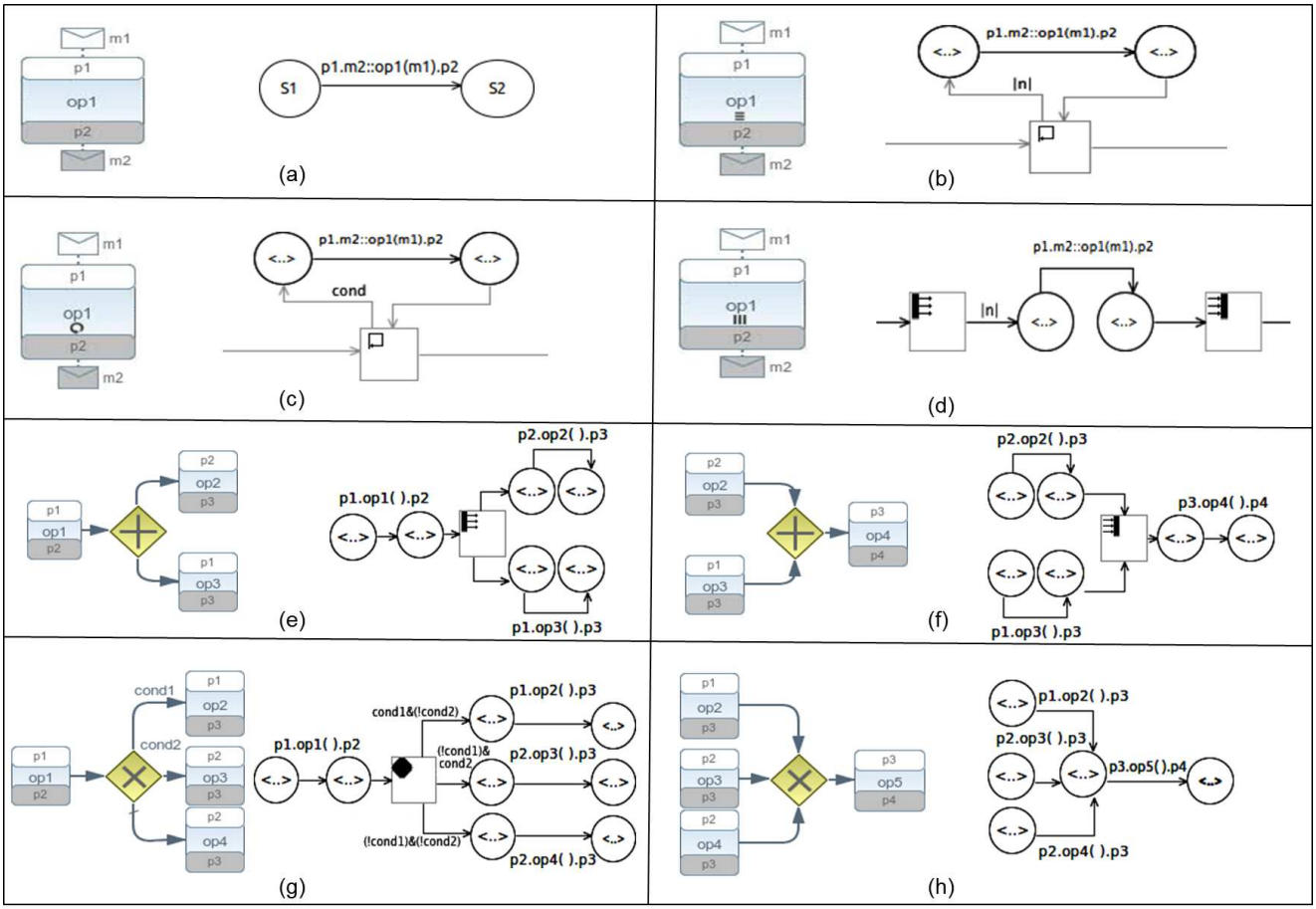


Figure 4.5: From BPMN2 choreography diagram to CLTS

any admissible combination (according the BPMN 2.0 specification) of the source elements. Note that, for presentation purposes, in the figure we only consider simple cases. The extensions to more general situations is straightforward.

Still referring to Figures 4.5 and 4.6, in the following we leverage the in-depth study of the "meanders" of the BPMN 2.0 specification we have been making so far, and introduce most of the BPMN2 choreography diagrams' constructs and provide a concise description of the crucial characteristics that must be taken into account for achieving a correct distributed coordination, and hence a correct enforcement. In this sense, as already anticipated in Section 4.1, OMG BPMN2 standard specification gives well-formedness modeling rules that must be respected by the choreography designer for the choreography to be enforceable. These rules were also advocated in other works that propose approaches for checking them, e.g., [24, 76, 40, 42, 18]. Thus, when saying "a correct distributed coordination, and hence a correct enforcement", we mean that the run-time interaction of the services (that have been discovered as suitable to play the choreography participant roles) will be coordinated according to a coordination logic that has been extracted by considering all the enforceability rules dictated by the standard specification.

We review each of the transformations of Figures 4.5 and 4.6 in turn below:

a..d A choreography Task is an atomic activity that represents an interaction by means of one or two (request and optionally response) message exchanges between two participants. As detailed in Chapter 3, in CHOReOS the communication between the services (as abstracted by CHOReOS components described in Chapter 2) that play the roles of the participants, is achieved through the GA connector (hence, supporting interoperability across different interaction paradigms).

A task may have one of the three markers: sequential multi-instance (b), standard loop (c), and

parallel multi-instance (d). Graphically, BPMN2 diagrams uses rounded-corner boxes to denote choreography tasks. Each of them is labeled with the roles of the two participants involved in the task, and the name of the service operation performed by the initiating participant and provided by the other one. A role contained in the white box denotes the initiating participant. In particular, we recall that the BPMN2 specification employs the theoretical concept of a token that, traversing the sequence flows and passing through the elements in a process, aids to define its behavior. The start event generates the token that must eventually be consumed at an end event.

Depending on its type (i.e., `ChoreographyLoopType` in the BPMN2 specification), a task is transformed into a basic state-to-state transition if no marker is specified (a), a CLTS `ActivityLoop` transition with a fixed number |n| of possible iterations if a BPMN2 sequential multi-instance marker is specified (b), a conditional CLTS `ActivityLoop` transition if a BPMN2 standard loop marker is specified (c), and a CLTS state-to-state transition that can be forked (and then joined) a fixed number |n| of times if a BPMN2 parallel multi-instance marker is specified (d). Note that the BPMN2 graphical elements do not show, neither the condition expressions nor the specified fixed number, which can only be internally specified. In/out messages (i.e., request/response messages) are reported in the CLTS transitions on the right-hand and left-hand sides of the operation name, respectively. To bound the number of times a loop is repeated, the CDs either evaluate the condition expressions (based on the data contained in the messages exchanged) in the case of a standard loops, or use internal counters (updated upon the observed message exchanges) in the case of sequential and parallel multi-instance loops. In any case, the task is performed at least once, before checking the condition or the counter. In this respect, it is worth to mention that one of the reported critical issues (issue number 16554 - published in the official web site) is about "underspecification of `ChoreographyLoopType`". The reported issue says that *the loop types of choreography activities do not provide any information about: (1) the "hard-coded" cardinality of the loop (how often is the loop repeated) or (2) the expression that must be evaluated during the choreography enactment based on the data contained in the messages exchanged up to that point during the enactment. In the latter case, specifying the expression is not enough: for reasons of enforceability, it should be possible to specified which participant(s) must evaluate the expression*. Accounting for this issue, in our model transformation we have anticipated its resolution.

e & f A Parallel Gateway is used to (e) create and/or (f) synchronize parallel flows without checking any condition. Each outgoing flow receives a token upon execution of this gateway. For incoming flows, this gateway will wait for all incoming flows before triggering the flow through its outgoing arrow. They create parallel paths of the choreography that all Participants are aware of. With respect to the constraints imposed by the BPMN 2.0 official specification, the initiator participant(s) of all the tasks after the gateway must be involved in all tasks that immediately precede such gateway. The task that precedes the chain must also satisfy this constraint in the case where there is a chain of gateways with no tasks in between.

When used to create parallel flows, the parallel diverging gateway is transformed using a CLTS `Fork` state that splits into all the outgoing flows. Note that, as it will be clear in Section 4.2.3, in order to enforce the coordination logic implied by a parallel gateway, the `Fork` state is used in a CLTS to model real parallelism (and not abstract parallelism by means of interleaving). Complementarily, when a parallel converging gateway is used to join parallel flows, a CLTS `Join` state is used. Similar considerations on concrete versus abstract parallelism applies.

g & h A Diverging (Decision) Exclusive Gateway (g) is used to create alternative paths within a choreography. If none of the conditional expressions (see `cond1` and `cond2`) evaluate to true, a default path can optionally be specified (see task `op4`). A Converging Exclusive Gateway (h) is used to merge alternative paths. Each incoming flow token is routed to the outgoing flow without synchronization. Being in a fully decentralized setting, there is no central mechanism to store the data that will be used in the condition expressions of the outgoing flows. The gateway's conditions may

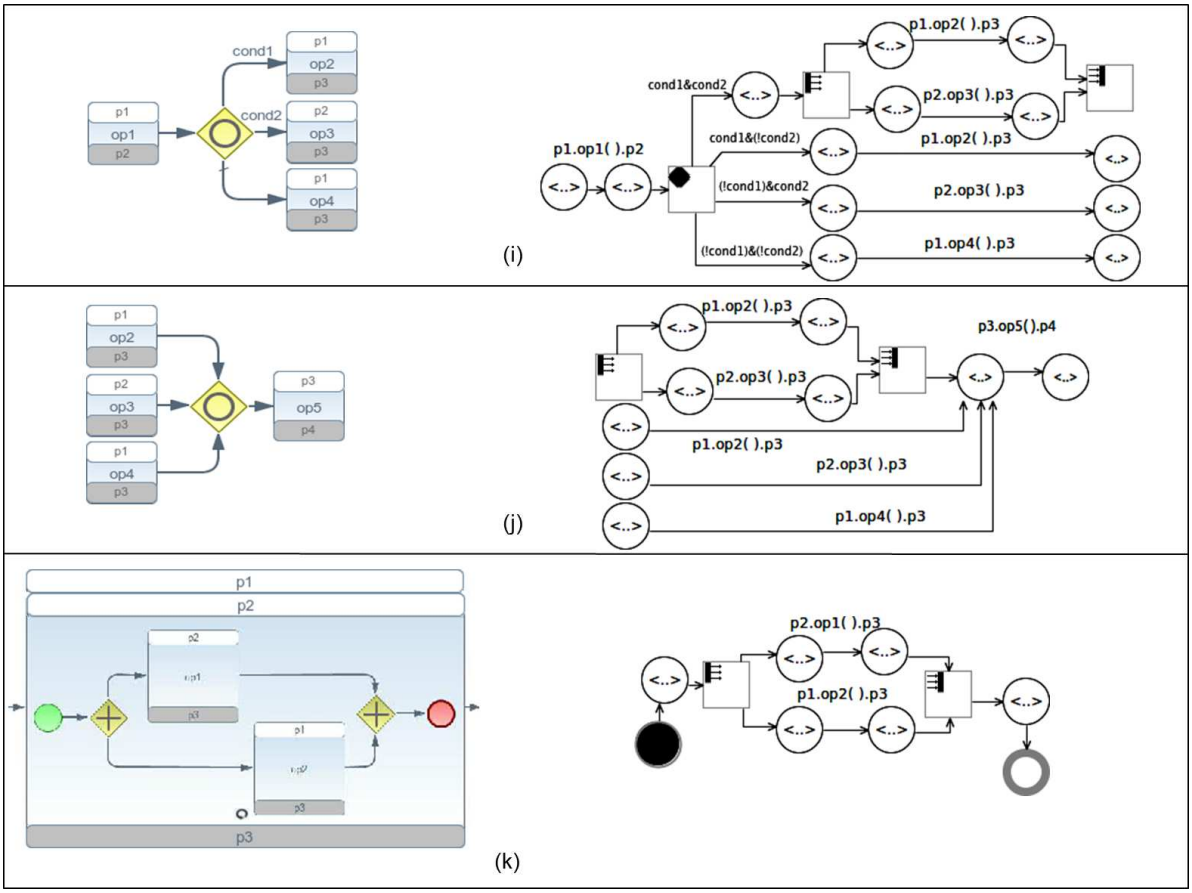


Figure 4.6: From BPMN2 choreography diagram to CLTS (Cont'd)

have natural language descriptions but, as clarified by the BPMN 2.0 official specification, such choreographies would be underspecified and would not be enforceable. To create an enforceable choreography, the gateway conditions must be formal expressions that can be precisely (and automatically for tool supported approaches) checked. Still according to the BPMN 2.0 official specification, the initiating participants of the choreography tasks that follow the gateway must have sent or received the message that provided the data upon which the conditional decision is made. In addition, the message that provides the data for the gateway conditional decision may be in any choreography task prior to the gateway (i.e., it does not have to immediately precede the gateway). Thus, for the gateway to be automatically enforced, we assume to have the specification of what messages provide the data upon which the conditional decision can be actually made.

When used to create alternative paths, a diverging exclusive gateway is transformed using a CLTS `Alternative` state. Note that the conditions `cond1` and `cond2` are suitably combined to achieve exclusivity. When used to merge alternative paths, a converging exclusive gateway is transformed using state-to-state transitions that, by modeling the flows immediately preceding the gateway, collapse into a further state-to-state transition that models the flow immediately following the gateway.

As a further clarification, note that the very same converging exclusive gateway behavior can be equivalently specified in BPMN 2.0 without using the gateway construct. That is, with reference to the figure, it is sufficient to have three arrows that directly connect the tasks on the left to the task on the right.

i & j A Diverging Inclusive (Decision) Gateway (i) can be used to create alternative but also parallel paths. Unlike the Exclusive Gateway, all condition expressions are evaluated. All flows that eval-

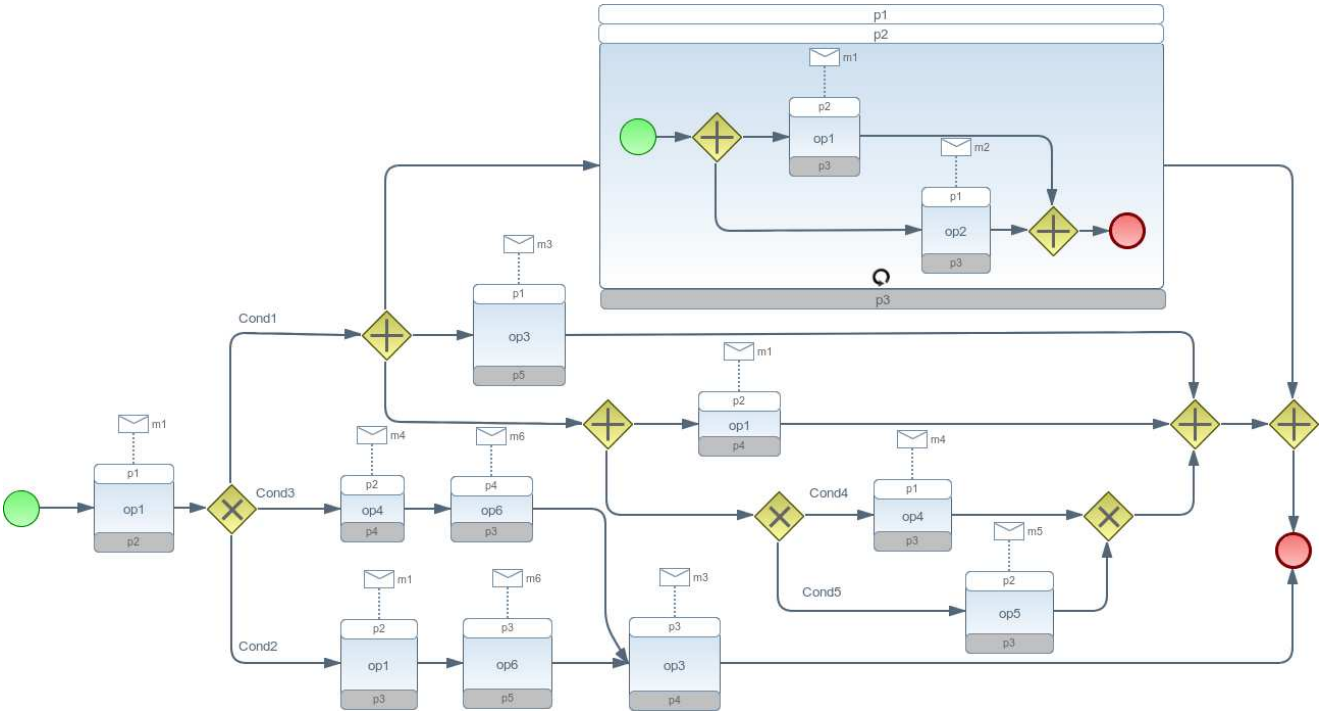


Figure 4.7: BPMN2 choreography diagram example

uate to true will be traversed by a token. Since each path is considered to be independent, any combination of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken. If none of the conditional expressions (see `cond1` and `cond2`) evaluate to true, a default path can optionally be specified. A converging Inclusive Gateway (j) is used to merge a combination of alternative and parallel paths. A control flow token entering an Inclusive Gateway may be synchronized with some other token that arrives later.

Similarly to a diverging exclusive gateway, diverging inclusive gateway is transformed using a CLTS `Alternative` state. However, to model that all combinations of the paths may be taken, combined forking and joining paths are used. To conform with this characteristic, the conditions `cond1` and `cond2` are suitably combined to achieve exclusivity between not only single paths, but also between the combined forking and joining paths and single paths. Considering the previous explanations for the converging exclusive gateway and the diverging inclusive gateway, the transformation for a converging exclusive gateway, when merging combinations of alternative and parallel paths, is rather intuitive.

k A Sub-Choreography is a compound activity task that defines a flow of other tasks. Each sub-choreography involves two or more participants.

Compound activities tasks are transformed by recursively applying the previous rules. In the figure, only a very simple case is shown.

By applying the transformation rules described above, the small, yet interesting, BPMN2 choreography diagram of Figure 4.7 is automatically transformed to the corresponding CLTS diagram in Figure 4.8 (the CLTS diagram has been drawn by means of the GMF-based editor we have developed in CHOReOS).

4.2.3. From CLTS to Coord Models

In this section we describe the `CLTS-to-Coord` ATL transformation that automatically distributes the CLTS into a set of coordination models. The latter contain coordination information codified as a set of

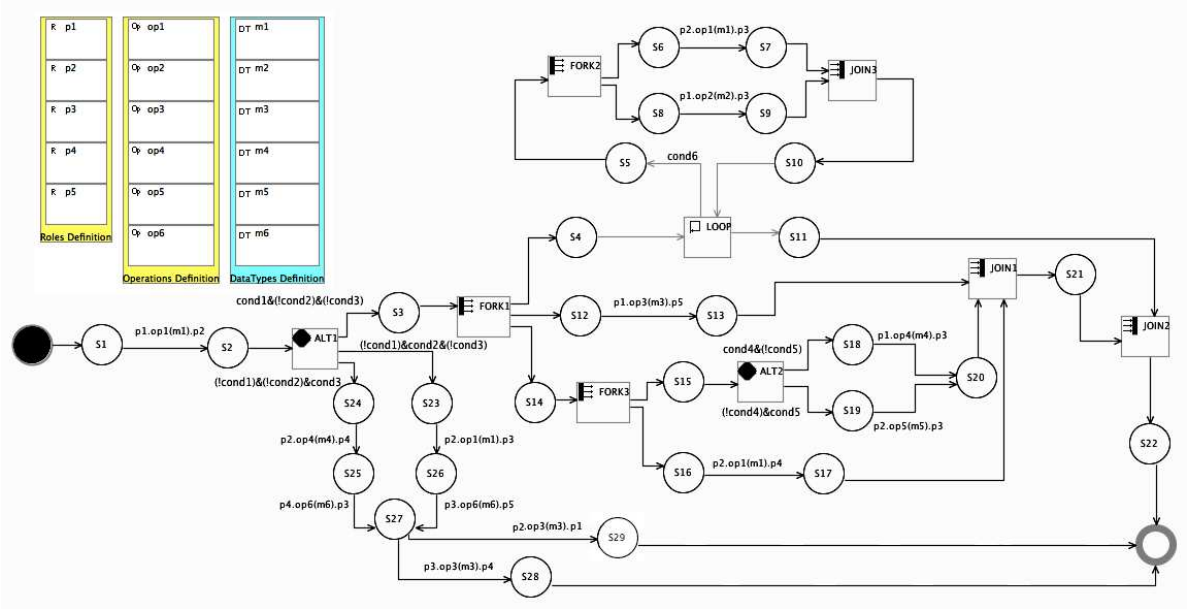


Figure 4.8: CLTS derived from the BPMN2 choreography diagram in Figure 4.7

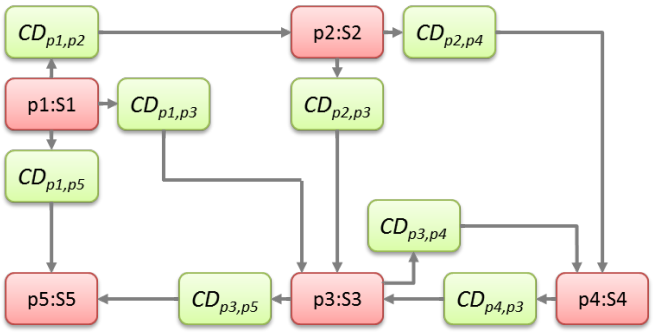


Figure 4.9: Architecture of the example

tuples (called coordination tuples). The coordination models are XML files that contains these tuples. As it will be clear soon, the coordination tuples that we are able to derive with the new synthesis process represent an extension of the coordination tuples described in the previous deliverable D1.3 [90].

The `CLTS-to-Coord` transformation takes as input the CLTS and, for each interface that a choreography participant p_i requires from another participant p_j, a coordination model $M_{CD_{pi.pj}}$ is derived. The model $M_{CD_{pi.pj}}$ will be then the input of the coordination delegate $CD_{pi.pj}$ that (according to the CHOReOS architectural constraints) is interposed between the services playing the roles of the choreography participants p_i and p_j.

For the convenience of the reader, before describing the format of the coordination tuples contained into the coordination models, Figure 4.9 shows the set of CDs that are generated for the BPMN2 choreography diagram of Figure 4.7 and how they are interposed between the services that play the roles of the involved participants. We further recall that, for a coordination delegate $CD_{pi.pj}$, $M_{CD_{pi.pj}}$ specifies the local information that $CD_{pi.pj}$ needs to know in order to properly cooperate with the other CDs in the system. The aim of this cooperation is to prevent undesired interactions in the global collaboration of the participant services, hence enforcing choreography realizability. More precisely, the coordination information exchanged between the coordination delegates (see the *additional communication* in Figure 4.1) serves to keep track of the global state of the coordination protocol implied by the choreography that each delegate can deduce from the observed exchange of messages (see *standard communication* in Figure 4.1).

Having this goal in mind, in the following we provide a plain-text representation of the coordination tuples as contained in all the coordination models derived for the example of Figure 4.7. Then, the elements constituting the tuples are described.

$$
\begin{aligned}
M_{CD_{p1.p2}}=\{&\langle S1, op1(m1), S2, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle S2, \{\}, ALT1, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle ALT1, \{\}, S3, Ask(), CD\{\},\\
&\quad cond1\&(!cond2)\&(!cond3), Notify(),\\
&\quad Wait()\rangle\\
&\langle ALT1, \{\}, S23, Ask(), CD\{2.3\},\\
&\quad (!cond1)\&cond2\&(!cond3), Notify(),\\
&\quad Wait()\rangle\\
&\langle ALT1, \{\}, S4, Ask(), CD\{2.4\},\\
&\quad (!cond1)\&(!cond2)\&cond3, Notify(),\\
&\quad Wait()\rangle\\
&\langle S3, \{\}, FORK1, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle FORK1, \{\}, S14, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle S14, \{\}, FORK3, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle FORK3, \{\}, S15, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle FORK3, \{\}, S16, Ask(), CD\{2.4\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle S15, \{\}, ALT2, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\langle ALT2, \{\}, S18, Ask(), CD\{1.3\},\\
&\quad cond4\&(!cond5), Notify(),\\
&\quad Wait()\rangle\\
&\langle ALT2, \{\}, S19, Ask(), CD\{2.3\},\\
&\quad (!cond4)\&cond5, Notify(),\\
&\quad Wait()\rangle\\
&\langle FORK1, \{\}, S12, Ask(), CD\{1.5\},
\end{aligned}
$$

$true, Notify(),$
$Wait()\rangle$

$\langle FORK1, \{\}, S4, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S4, \{\}, LOOP, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle LOOP, \{\}, S5, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S5, \{\}, FORK2, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S6, Ask(), CD\{2.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S8, Ask(), CD\{1.3\},$
$true, Notify(),$
$Wait()\rangle$

}

$M_{CD_{p2.p3}}$={$\langle S23, op1(m1), S26, Ask(), CD\{3.5\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S19, op5(m5), S20, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S20, \{\}, JOIN1, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle JOIN1, \{\}, S21, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S21, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S20\ to\ CD(2.3, 1.3, 1.5, 2.4)),$
$Wait(S11\ from\ CD(2.3\ or\ 1.3), S13\ from\ CD(1.5), S17\ from\ CD(2.4))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S6, op1(m1), S7, Ask(), CD\{\},$

$true, Notify(),$
$Wait()\rangle$

$\langle S7, \{\}, JOIN3, Ask(), CD\{\},$
$true, Notify(S7\ to\ CD(1.3)),$
$Wait(S9\ from\ CD(1.3))\rangle$

$\langle JOIN3, \{\}, S10, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S10, \{\}, LOOP, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle LOOP, \{\}, S5, Ask(), CD\{\},$
$cond6, Notify(),$
$Wait()\rangle$

$\langle S5, \{\}, FORK2, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S6, Ask(), CD\{2.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S8, Ask(), CD\{1.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle LOOP, \{\}, S11, Ask(), CD\{\},$
$!cond6, Notify(),$
$Wait()\rangle$

$\langle S11, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S11\ to\ CD(1.5, 1.3, 2.3, 2.4)),$
$Wait(S13\ from\ CD(1.5), S20\ from\ CD(1.3\ or\ 2.3), S17\ from\ CD(2.4))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

}

$M_{CD_{p1.p3}}$={$\langle S18, op4(m4), S20, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S20, \{\}, JOIN1, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle JOIN1, \{\}, S21, Ask(), CD\{\},$

$true, Notify(),$
$Wait()\rangle$

$\langle S21, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S20\ to\ CD(2.3, 1.3, 1.5, 2.4)),$
$Wait(S11\ from\ CD(2.3\ or\ 1.3), S13\ from\ CD(1.5), S17\ from\ CD(2.4))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S8, op2(m2), S9, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S9, \{\}, JOIN3, Ask(), CD\{\},$
$true, Notify(S9\ to\ CD(2.3)),$
$Wait(S7\ from\ CD(2.3))\rangle$

$\langle JOIN3, \{\}, S10, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S10, \{\}, LOOP, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle LOOP, \{\}, S5, Ask(), CD\{\},$
$cond6, Notify(),$
$Wait()\rangle$

$\langle S5, \{\}, FORK2, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S6, Ask(), CD\{2.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle FORK2, \{\}, S8, Ask(), CD\{1.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle LOOP, \{\}, S11, Ask(), CD\{\},$
$!cond6, Notify(),$
$Wait()\rangle$

$\langle S11, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S11\ to\ CD(1.5, 1.3, 2.3, 2.4)),$
$Wait(S13\ from\ CD(1.5), S20\ from\ CD(1.3\ or\ 2.3), S17\ from\ CD(2.4))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$

$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

}

$M_{CD_{p1.p5}}$=$\{\langle S12, op3(m3), S13, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S13, \{\}, JOIN3, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle JOIN3, \{\}, S21, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S21, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S13\ to\ CD(2.3, 1.3, 2.4)),$
$Wait(S11\ from\ CD(2.3\ or\ 1.3), S20\ from\ CD(1.3\ or\ 2.3), S17\ from\ CD(2.4))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

}

$M_{CD_{p2.p4}}$=$\{\langle S24, op4(m4), S25, Ask(), CD\{4.3\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S16, op1(m1), S17, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S17, \{\}, JOIN3, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle JOIN3, \{\}, S21, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S21, \{\}, JOIN2, Ask(), CD\{\},$
$true, Notify(S17\ to\ CD(2.3, 1.3, 1.5)),$
$Wait(S11\ from\ CD(2.3\ or\ 1.3), S13\ from\ CD(1.5), S20\ from\ CD(1.3\ or\ 2.3))\rangle$

$\langle JOIN2, \{\}, S22, Ask(), CD\{\},$
$true, Notify(),$
$Wait()\rangle$

$\langle S22, \{\}, FinalState, Ask(), CD\{\},$

$$
\begin{aligned}
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\}\\
M_{CD_{p4.p3}}=&\{\langle S25, op6(m6), S27, Ask(), CD\{3.4, 2.1\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\}\\
M_{CD_{p3.p5}}=&\{\langle S26, op6(m6), S27, Ask(), CD\{3.4, 2.1\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\}\\
M_{CD_{p2.p1}}=&\{\langle S27, op3(m3), S29, Ask(CD(3.4)\ for\ S27), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\quad \langle S29, \{\}, FinalState, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\}\\
M_{CD_{p3.p4}}=&\{\langle S27, op3(m3), S28, Ask(CD(2.1)\ for\ S27), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\quad \langle S28, \{\}, FinalState, Ask(), CD\{\},\\
&\quad true, Notify(),\\
&\quad Wait()\rangle\\
&\}
\end{aligned}
$$

Each tuple is composed of eight elements:

- The **first element** denotes the CLTS source state from which the related CD can either perform the operation specified as **second element** of the tuple or take a move without performing any operation (i.e., the CD can step over an epsilon transition). In both cases, the **third element** denotes the reached target state. For instance, the first tuple of $M_{CD_{p1.p2}}$ specifies that the coordination delegate $CD_{p1.p2}$ can perform the operation $op1$ with message $m1$ from the source state $S1$ to the target state $S2$; whereas, the second tuple of $M_{CD_{p1.p2}}$ specifies that the coordination delegate $CD_{p1.p2}$ can step over the state $S2$ and reach the state $ALT1$, from where alternative branches can be undertaken. That is, as specified by the third, fourth and fifth tuple, the coordination delegate $CD_{p1.p2}$ can reach either the state $S3$, or the state $S23$, or the state $S24$ according to the evaluation of the related conditions.
- The **fourth element** contains the set of states and related CDs that must be asked for to check whether the specified (allowed) operation can be forwarded or not. This means that race conditions can arise when, at a given execution point, more than one service wants to perform an operation but, according to the choreography specification, only one must be unconditionally elected. For instance, in the state $S27$, the coordination delegate $CD_{p2.p1}$ can be in a race condition with the coordination delegate $CD_{p3.p4}$ (and viceversa), whenever both $p2$ and $p3$ are ready to request the operation $op3$ with message $m3$ to $p1$ and $p4$, respectively. To solve this race condition, the tuple $\langle S27, op3(m3), S29, Ask(CD(3.4)\ \ for\ \ S27), CD\{\}, true, Notify(), Wait()\rangle$ contained in $M_{CD_{p2.p1}}$ informs the coordination delegate $CD_{p2.p1}$ that before forwarding the operation $op3$, it must ask the permission to the coordination delegate $CD_{p3.p4}$

about the inquired state $S27$. Complementarily, the same applies for the tuple $\langle S27, op3(m3), S28, Ask(CD(2.1)\ for\ S27), CD\{\}, true, Notify(), Wait()\rangle$ contained in $M_{CD_{p3.p4}}$. As extensively discussed in the previous deliverable D1.3 [90], race conditions are solved by applying a suitable extension of the seminal algorithm proposed in [55]. Thus, in this deliverable the resolution of race conditions is not further discussed.

- The **fifth element** contains the set of (identifiers of) those CDs whose supervised services became active in the target state, i.e., the ones that will be allowed to require some operation from the target state. This information is used by the "currently active" CD(s) to inform the set of "to be activated" CDs (in the target state) about the changing global state. For instance, upon the operation $op1$ is requested from $p2$ to $p3$, the coordination delegate $CD_{p2.p3}$ uses the fifth element $CD\{3.5\}$ of the first tuple in $M_{CD_{p2.p3}}$ to inform the CD $CD_{p3.p5}$ about the new global state $S26$.
- The **sixth element** reports the condition expression to be checked to select the correct tuple, and hence the correct flow(s) in the CLTS. For example, referring to the third tuple of $M_{CD_{p1.p2}}$, if the condition expression $cond1\&(!cond2)\&(!cond3)$ evaluates to true, then the coordination delegate $CD_{p1.p2}$ can step over the alternative state $ALT1$ and reach $S3$.
- The **seventh element** contains the joining state that a CD, when reaching a join state, must notify to the other CDs in the parallel path(s) of the same originating fork. Complementarily, the **eight element** contains the joining state(s) that must be waited for. For example, considering the tuple $\langle S7, \{\}, JOIN3, Ask(), CD\{\}, true, Notify(S7\ to\ CD(1.3)), Wait(S9\ from\ CD(1.3))\rangle$ of $M_{CD_{p2.p3}}$, the coordination delegate $CD_{p2.p3}$ notifies the joining state $S7$ to the coordination delegate $CD_{p1.p3}$, and wait for the state $S9$ from $CD_{p1.p3}$. On the other hand, considering the tuple $\langle S9, \{\}, JOIN3, Ask(), CD\{\}, true, Notify(S9\ to\ CD(2.3)), Wait(S7\ from\ CD(2.3))\rangle$ of $M_{CD_{p1.p3}}$, the coordination delegate $CD_{p1.p3}$ notifies the joining state $S9$ to the coordination delegate $CD_{p2.p3}$, and wait for the state $S7$ from $CD_{p2.p3}$.

4.3. CHOReOS-compliant Architecture of the Airport System

This section concludes the presentation of the CHReOS coordination model by applying the synthesis process to the Airport use case of WP6 in order to derive a CHOReOS-compliant architectural configuration for the whole system.

Figure 4.10 and Figure 4.11 show the (split) BPMN2 specification of the Arrival Handling scenario. It is worth noting that BPMN2 can be used for creating simple and rather abstract choreography specifications (that can be useful for providing, e.g., a business manager with a high-level view of a given business process), but also detailed and technical specifications (that can be parsed by a machine and automatically manipulated by developers and analysts, e.g., for tool-supported analysis and synthesis). Within CHOReOS, this peculiarity of BPMN2 creates the opportunity to align IT people and business people, as well as end-users, over the same goals and requirements, avoiding to provide business people and end-users with too many technical details, and IT people with inaccurate specifications. To this end, as described in the deliverable D2.2 [92], the CHOReOS development process gives the possibility to refine the BPMN2 specification into more detailed versions, in an iterative fashion, containing more technical details, hence being able to apply the synthesis process in an automatic way. That is, Figure 4.12 and Figure 4.13 show (and split) a refined version of the BPMN2 choreography specification of Figure 4.10 and Figure 4.11, respectively. In particular, we refined the names of the choreography tasks, we added the exchanged messages, and we specified the type of input and output parameters. The set of refinements are reported in Table 4.1.

Thus, in the following, we apply the synthesis process by taking as input the refined BPMN2 choreography specification depicted in Figure 4.12 and Figure 4.13, and show how the model transformations, the synthesis process relies on can be applied to extract from the BPMN2 specification the coordination logic to be applied to enforce the choreography. Finally, we further show the architecture of the whole

	Initial Task Name	Refined Task Name and In/Out Types
1.	Confirm Approach	Confirm Approach
2.	Book Amenity	Book Amenity
3.	Request Amenities	Request Amenities
4.	Get Available Amenities	Get Available Amenities
5.	Measure ambient noise level	double::Get Average Sound Level
6.	Adjust Speakers Volume Level	Adjust Speakers Volume Level(double)
7.	Initiate passengers tracking	boolean::Initiate Passengers Tracking
8.	Report passengers passed through	int::Report Passed Passengers
9.	Push up-to-date information (in the right areas)	Say(string) Display(string) Set Sign States(arrayList)
10.	Book All Available Amenities	Book All Available Amenities
11.	Get Confirmation	Get Luggage Confirmation Get Bus Confirmation Get Security Confirmation
12.	Push up-to-date information to all MIDs	Display(string)
13.	Airport Noise Sensors aggregator	double::Get Average Sound
14.	Track landed passengers (by areas)	booleanArray::Get Landing States intArray::Get Average Position

Table 4.1: Initial task names and refined task names with In/Out types

	Participant Name	Participant Acronym
1.	Airport	AIR
2.	ATC	ATC
3.	Stand and gate management	SGM
4.	Luggage handling company	LHC
5.	Security company	SC
6.	Airport bus company	ABC
7.	Amenity provider	AP
8.	Airport noise sensor aggregator	ANSA
9.	Airport speaker actuators aggregator	ASAA
10.	Airport infrared sensors aggregator	AISA
11.	Airport pressure sensors aggregator	APSA
12.	MID location sensors aggregator	MIDLSA
13.	MID microphone sensors aggregator	MIDMSA
14.	Airport display actuators aggregator	ADAA
15.	Airport sign actuators aggregator	ASAA
16.	MID display actuators aggregator	MIDDAA

Table 4.2: Participant names and corresponding acronyms

Airport system after the coordination delegates derived by synthesis processor have been interposed between the Airport services, by exploiting the CHOReOS connectors described in Chapter 3.

Figures 4.14 and 4.15 show the (split) CLTS that has been derived for the arrival handling scenario of the Airport use case, by using the acronyms for the participant names reported in Table 4.2.

The coordination models, containing all the tuples for all the coordination delegates, are reported in the Appendix A.1. The architecture of the whole system after the coordination delegates have been interposed between the Airport services, is shown (and split for more readability) in Figure 4.16 and Figure 4.17.

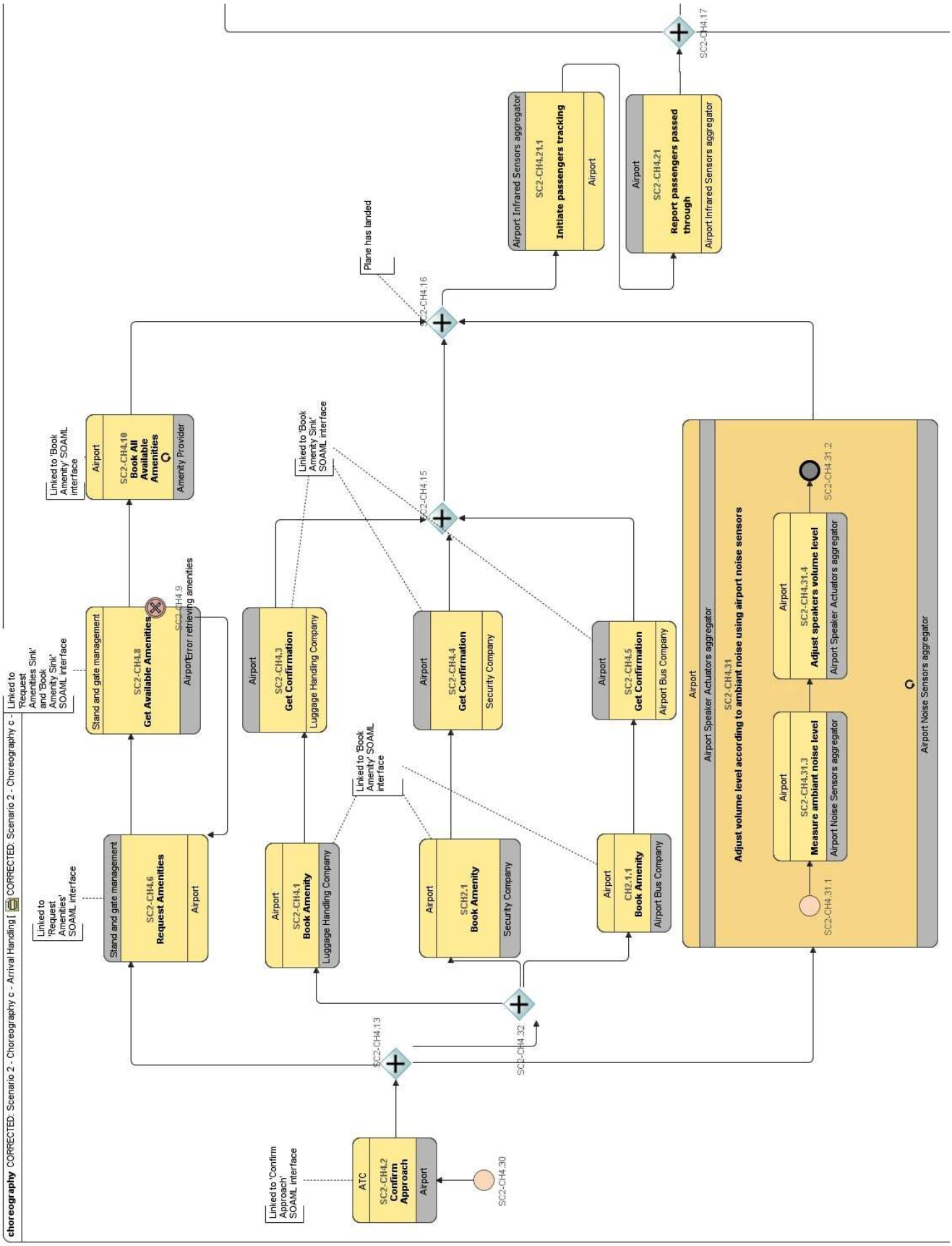


Figure 4.10: (a) Choreography of the arrival handling scenario

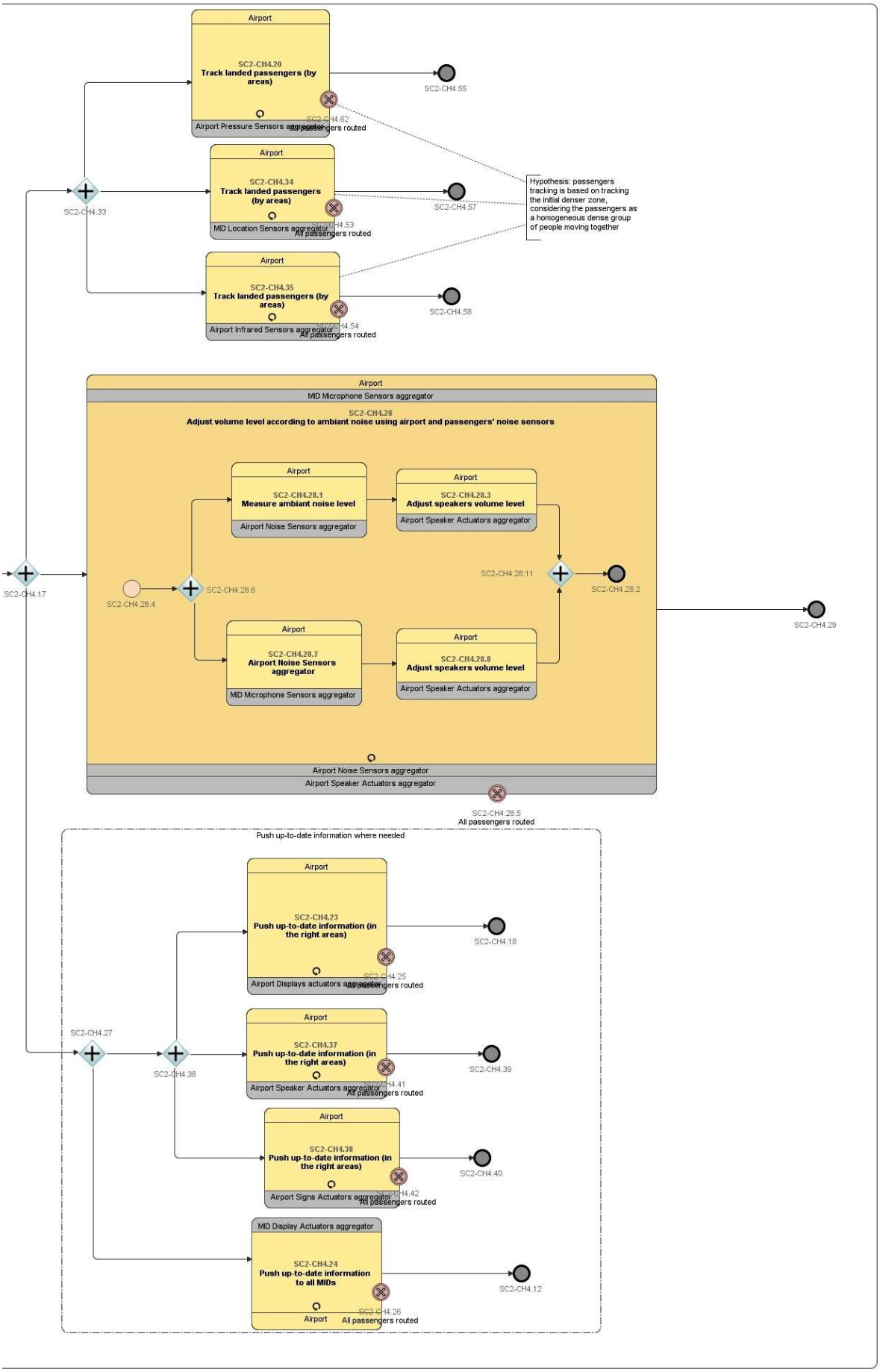


Figure 4.11: (b) Choreography of the arrival handling scenario

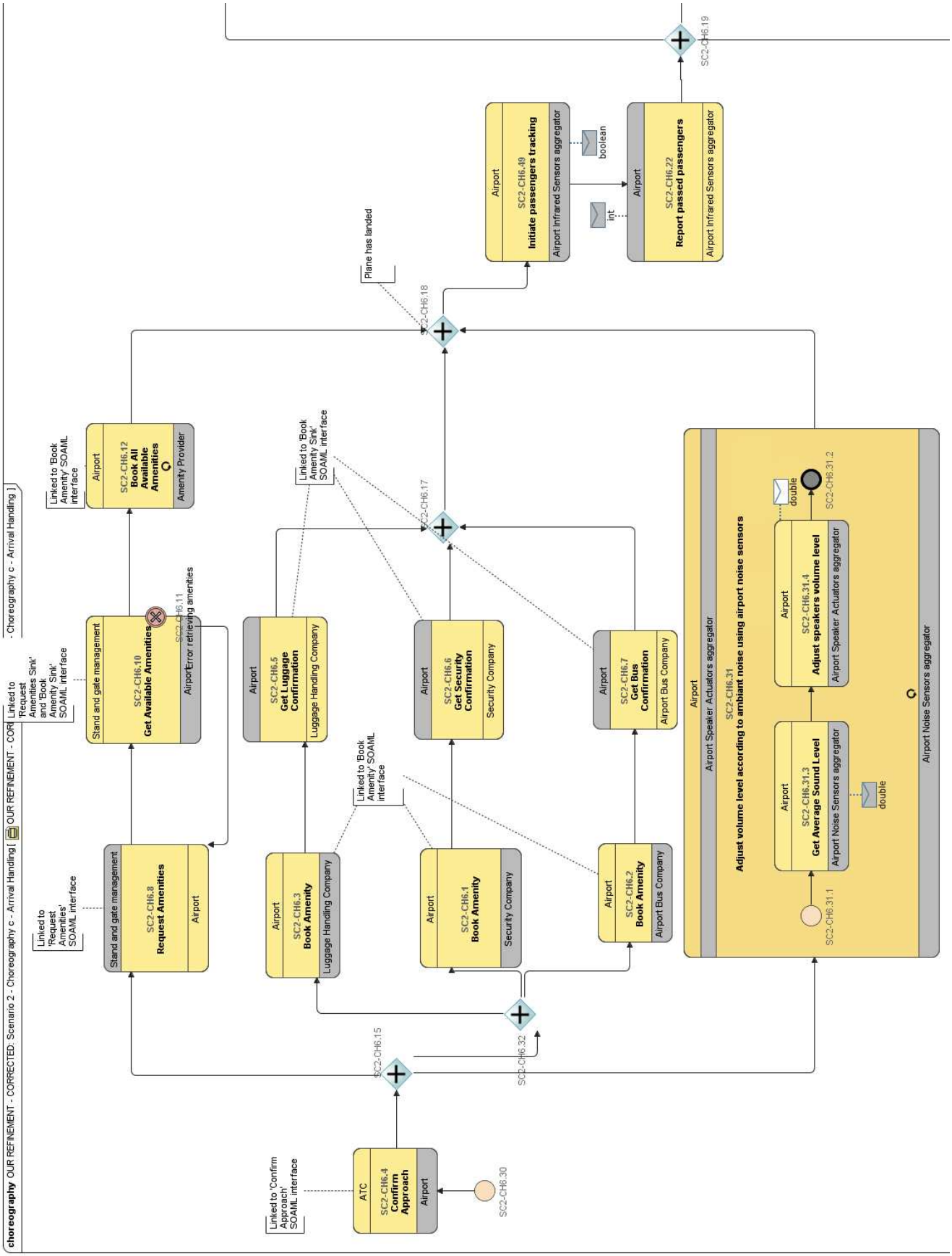


Figure 4.12: (a) Refined choreography of the arrival handling scenario

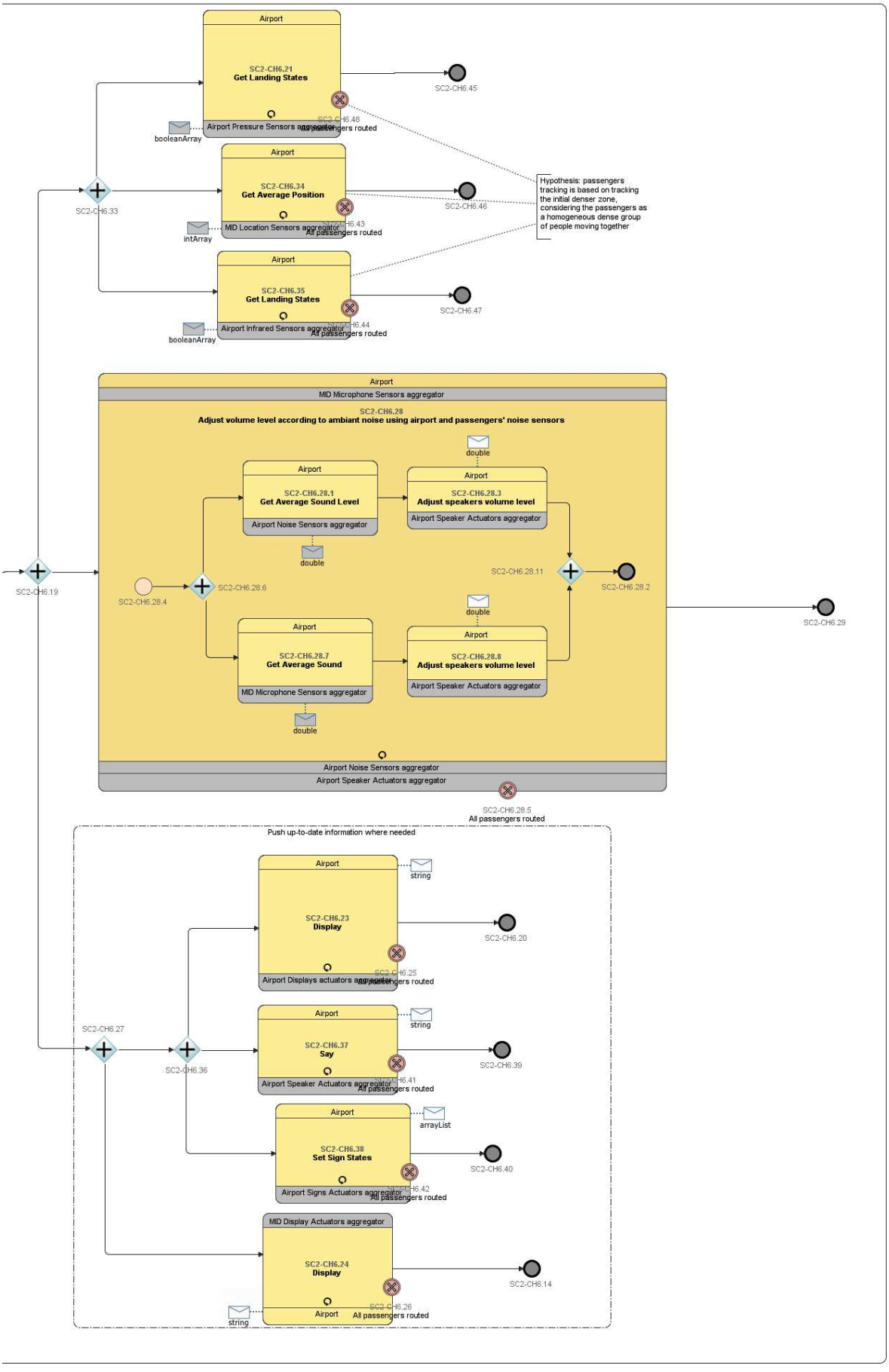


Figure 4.13: (b) Refined choreography of the arrival handling scenario

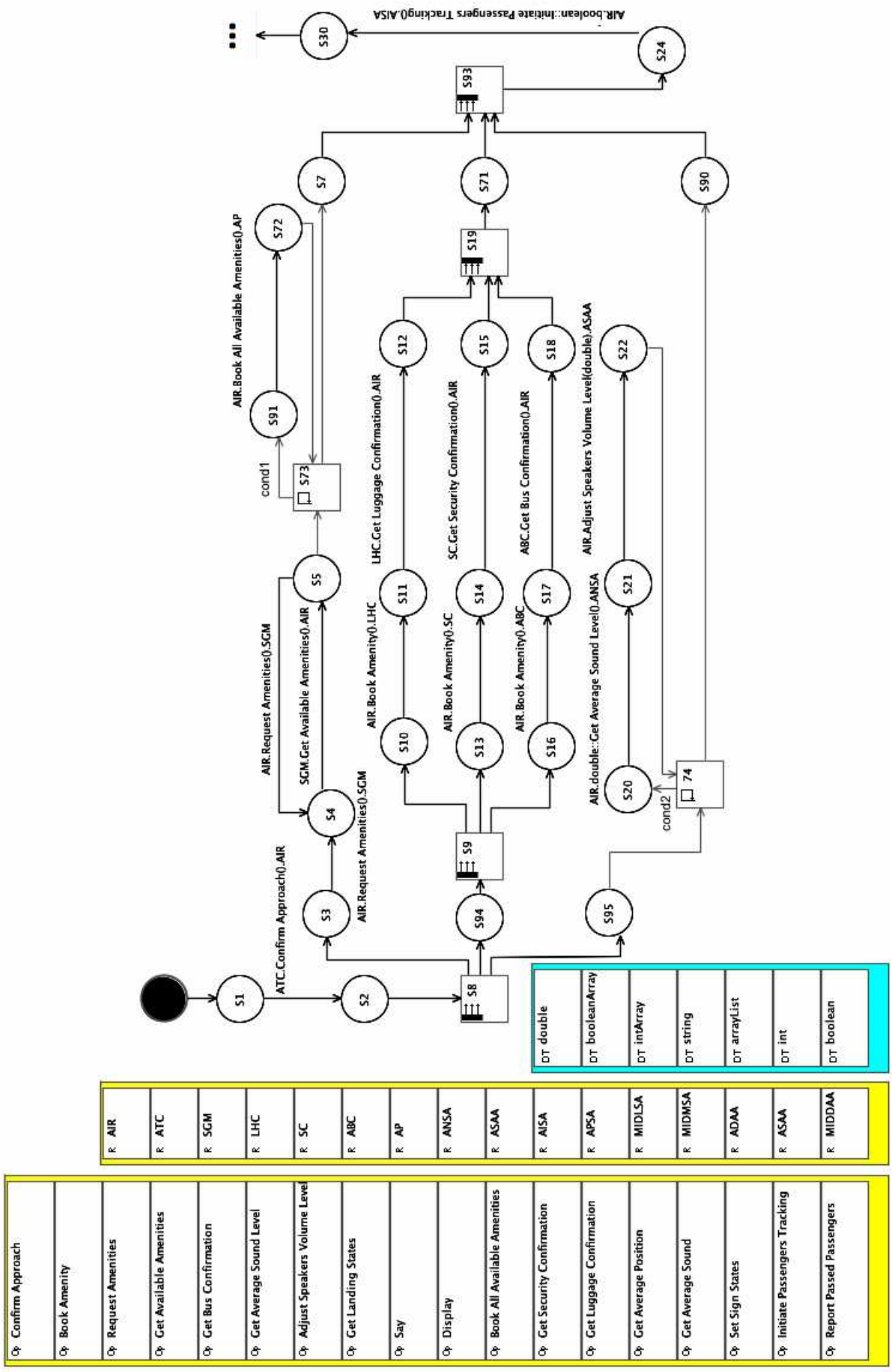


Figure 4.14: (a) Choreography of the arrival handling scenario

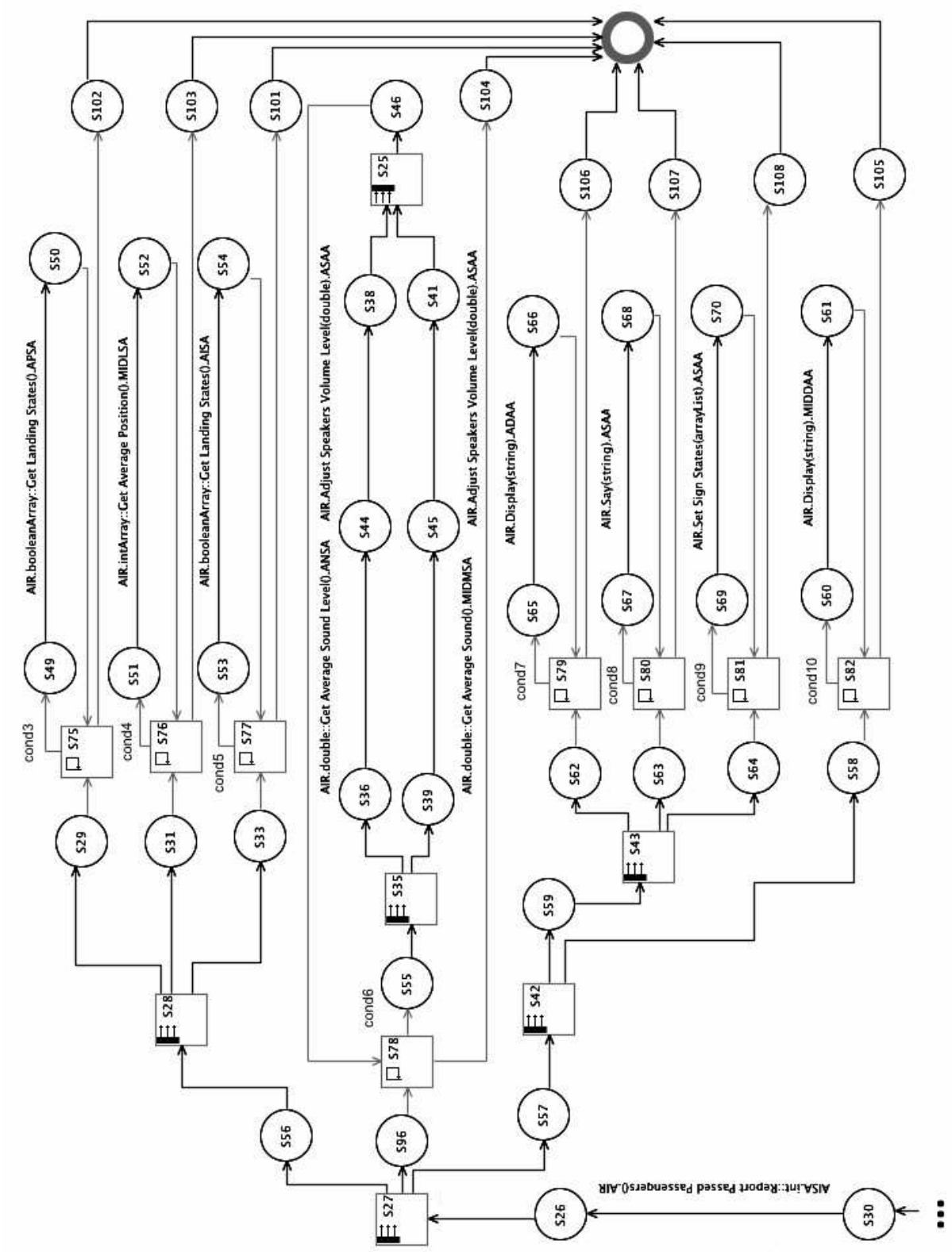


Figure 4.15: (b) Choreography of the arrival handling scenario

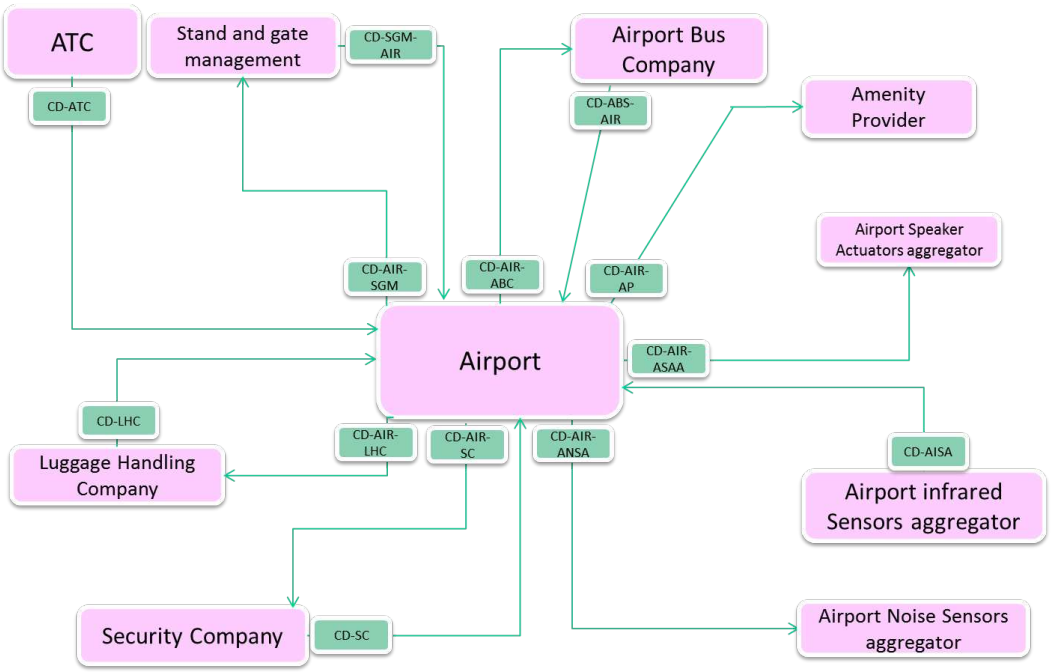


Figure 4.16: (a) Architecture of the airport use case

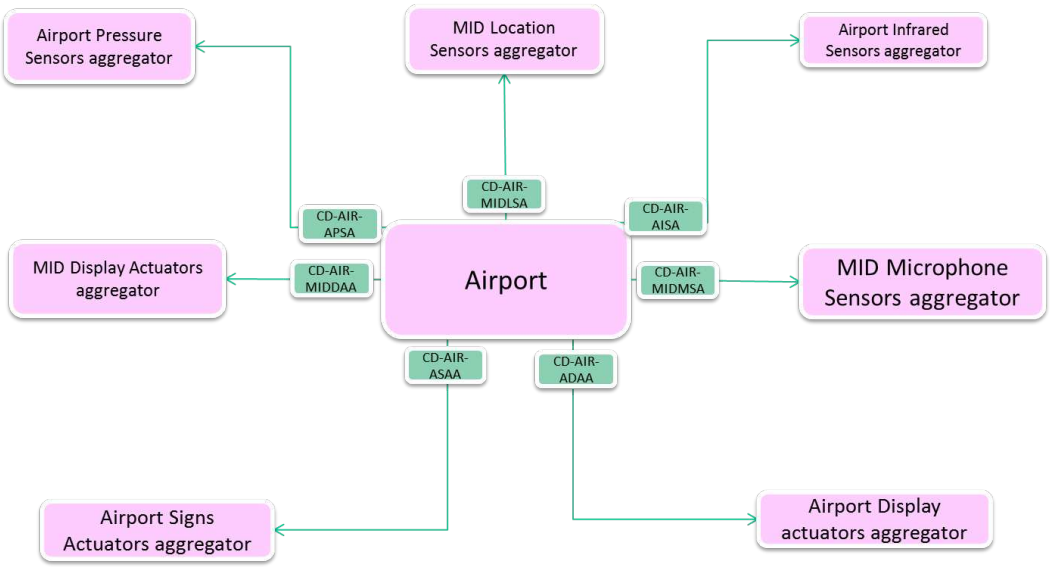


Figure 4.17: (b) Architecture of the airport use case

5 Conclusion: Relation with the CHOReOS Development Process and IDRE

This document has reported on the definition of the final CHOReOS architectural style, that is, the definition of core types of components, connectors, and related coordination protocols that are oriented toward easing the development of service-oriented systems in the ULS, heterogeneous, and mobile Future Internet. The initial definition of the architectural style was introduced in Deliverable D1.3 [90] and informed the development of the CHOReOS technologies integrated within the CHOReOS IDRE, which support the application of the CHOReOS development process whose schematic description is recalled in Figure 5.1. Then, based on the lessons learnt from the development of the CHOReOS IDRE and embedded technologies, we have refined the definition of the CHOReOS architectural style, as detailed in this report.

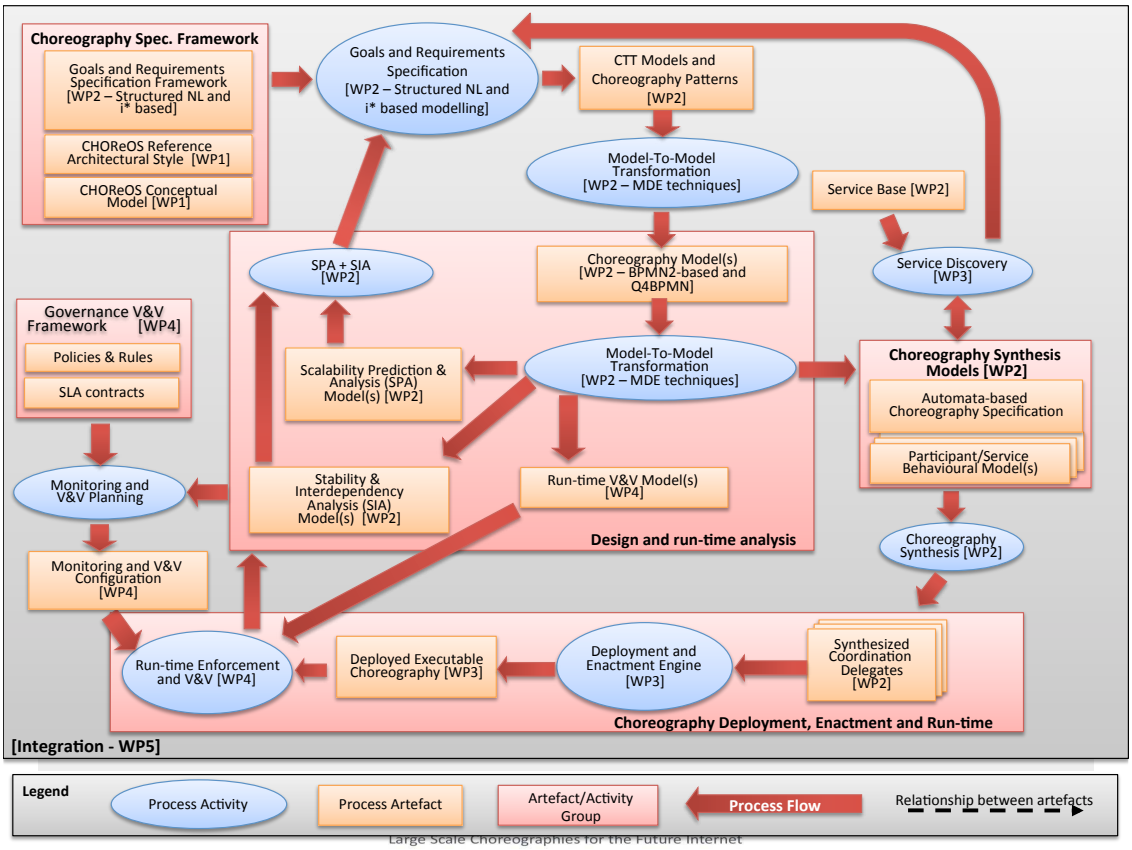


Figure 5.1: The CHOReOS development process

In brief, the definition of the CHOReOS architectural style features:

- *The service-oriented CHOReOS component model* that overcomes the heterogeneity of the services encountered in the FI, including Thing-based services. A core aspect of the CHOReOS component model relates to the introduction of the related concept of service abstractions, which represent groups of alternative services that provide similar functional/non-functional properties through different interfaces. Groups of services then constitute a major enabler to face the scale and adaptation requirements of the Future Internet.
- *The CHOReOS connector types* allow for interactions with services based on the various interaction paradigms that are available today, considering both discrete and continuous interactions, the latter being critical for interacting with Thing-based services that inform systems about the physical world that is continuously changing. A major contribution of the CHOReOS connector definition relates to the introduction of a new multi-paradigm connector type, called GA connector, which allows component services to interoperate even if they are based on heterogeneous paradigms.
- *The formal abstractions for FI choreography-based coordination* enable the automated synthesis of concrete distributed coordination protocols, from BPMN2 abstract specification and associated concrete services discovered in the environment, while enforcing the realizability of choreographies despite the autonomy of the composed services.

As detailed in the previous chapters, all the above elements are oriented toward facing the challenges posed by the Future Internet and especially:

- The *ULS of the Future Internet* impacts the definition of the three architectural elements: (i) the UL base of CHOReOS components is made manageable through the introduction of service abstractions, (ii) the CHOReOS connectors promote the adoption of loosely coupled interaction paradigms that are essential for developing scalable distributed systems, and (iii) the CHOReOS choreography-based coordination of service oriented-systems enables the development of a fully decentralized system out of individual services and things, which is also essential to foster the development of scalable distributed service-oriented systems.
- The *heterogeneity of the Future Internet* informed the definition of the CHOReOS component and connector types. Indeed, CHOReOS components are technology-agnostic by definition and allow modeling the rich diversity of services composing the Future Internet, including the Thing-based services. As for CHOReOS connectors, they support the various interaction paradigms exploited by the rich diversity of services composing the Future Internet and feature the new GA connector type that enables cross-paradigm interoperability. As CHOReOS choreographies compose CHOReOS components through CHOReOS connectors, they are obviously suited for the highly heterogeneous Future Internet.
- The *mobility of the Future Internet services and consumers* is implicitly supported by the CHOReOS architectural style as CHOReOS components may in particular be mobile and CHOReOS connectors may be instantiated using middleware technologies oriented toward mobile networking.
- The *awareness and adaptability* required by the Future Internet is primarily supported by the CHOReOS service abstractions associated with the CHOReOS component model, as services implementing the same abstractions can substitute one another. The support for weakly coupled connector types also contribute to easing the adaptation process.

The development of the CHOReOS technologies is further significantly impacted by the CHOReOS architectural style:

- The definition of the CHOReOS component model and related service abstraction has guided the definition of the AoSBM, which structures the service base according to service abstractions (see WP2 deliverables). Service abstractions further serve dealing with the adaptation of choreography, whose dedicated support is implemented by the CHOReOS middleware (see WP3 deliverables).
- The definition of the CHOReOS connector model has influenced the evolution of the ESB paradigm into the XSB paradigm, which allows multi-paradigm interoperability (see WP3 deliverables). Further evolution of the XSB paradigm is also being examined to support both discrete and continuous interactions, while current implementation of the XSB is oriented toward discrete interactions. In addition, acknowledging the importance of streaming protocols for the Future Internet, and especially its Internet of Things constituent, we are currently developing a DSMS as part of the CHOReOS middleware, which leverages the rich background on DSMS, from both the technological and theoretical points of view.
- The definition of the CHOReOS coordination model constitutes a central part of the CHOReOS IDE toolset, which features MDE-based tools for the selection of services according to behavioral matching rules and the synthesis of coordination delegates, provided the BPMN2 specification of the target choreography and the discovery of functionally matching services within the AoSBM.

Obviously, the proposed software architecture style may be leveraged to develop alternative technologies to CHOReOS solutions, either competitive or complementary, to support the development of highly distributed service-oriented systems within the Future Internet.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, 2005.

[2] DanielJ. Abadi, Don Carney, Ugur etintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12, 2003.

[3] I.F. Akyildiz, T. Melodia, and K.R. Chowdury. Wireless multimedia sensor networks: A survey. *Wireless Communications, IEEE*, 14(6), December.

[4] E. Al-Masri and Q. H. Mahmoud. Discovering Web Services in Search Engines. *IEEE Internet Computing*, 12(3):74–77, 2008.

[5] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3), 1997.

[6] G. Amato, P. Baronti, and S. Chessa. Mad-wise: Programming and accessing data in a wireless sensor networks. In *Computer as a Tool, 2005. EUROCON 2005.The International Conference on*, volume 2, Nov.

[7] Giuseppe Amato, Stefano Chessa, and Claudio Vairo. Mad-wise: a distributed stream management system for wireless sensor networks. *Software: Practice and Experience*, 40(5), 2010.

[8] Emil Andriescu, Amel Bennaceur, Paola Inverardi, Valerie Issarny, Romina Spalazzese, and Roberto Speicys-Cardoso. Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment. Research report, December 2012.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *Data-Stream Management: Processing High-Speed Data Streams*, chapter STREAM: The Stanford Data Stream Management System. Springer-Verlag, 2005.

[10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, jun 2006.

[11] D. Athanasopoulos and A. Zarras. Fine-grained Metrics of Cohesion Lack for Service Interfaces. In *Proceedings of the 9th International Conference on Web Services (ICWS)*, 2011.

[12] D. Athanasopoulos, A. Zarras, and P. Vassiliadis. Service Selection for Happy Users: Making User-Intuitive Quality Abstractions. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 32–35, 2012.

[13] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15), 2010.

[14] Edgardo Avilés-López and J. García-Macías. TinySOA: a service-oriented architecture for wireless sensor networks. *Service Oriented Computing and Applications*, 3(2), jun 2009.

[15] C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008.

[16] A. Bakshi, A. Pathak, and V.K. Prasanna. System-level Support for Macroprogramming of Networked Sensing Applications. In *Int. Conf. on Pervasive Systems and Computing (PSC)*. Citeseer, 2005.

[17] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: A continous query language for rdf data streams. *International Journal of Semantic Computing*, 04(01), 2010.

[18] Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 795–804, 2011.

[19] G.S. Blair, A. Bennaceur, G. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In *Proceedings of Middleware'2011*, 2011.

[20] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. Secret: a model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[21] Cabri, G. and Leonardi, L. and Zambonelli, F. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, pages 237–248, 1998.

[22] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[23] Jason Campbell, Phillip B. Gibbons, Suman Nath, Padmanabhan Pillai, Srinivasan Seshan, and Rahul Sukthankar. Irisnet: an internet-scale architecture for multimedia sensors. In *Proceedings of the 13th annual ACM international conference on Multimedia*, MULTIMEDIA '05, New York, NY, USA, 2005. ACM.

[24] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. volume 4421 of *LNCS*, pages 2–17. 2007.

[25] A. Carzaniga and A.L. Wolf. Content-based Networking: A New Communication Infrastructure. *Lecture Notes in Computer Science*, pages 59–68, 2002.

[26] Mohammad Dezfuli and Mostafa Haghjoo. Xtream: A system for continuous querying over uncertain data streams. In Eyke Hllermeier, Sebastian Link, Thomas Fober, and Bernhard Seeger, editors, *Scalable Uncertainty Management*. Springer Berlin / Heidelberg, 2012.

[27] Mohammad G. Dezfuli and Mostafa S. Haghjoo. Probabilistic querying over uncertain data streams. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 20(05), 2012.

[28] Robert Dickerson, Jiakang Lu, Jian Lu, and Kamin Whitehouse. Stream feeds - an abstraction for the world wide sensor web. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and SanjayE. Sarma, editors, *The Internet of Things*, pages 360–375. Springer Berlin Heidelberg, 2008.

[29] Nihal Dindar, Nesime Tatbul, ReneJ. Miller, LauraM. Haas, and Irina Botan. Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal*, 2012.

[30] Adam Dunkels. Full tcp/ip for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, New York, NY, USA, 2003. ACM.

[31] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The web of things: Interconnecting devices with high usability and performance. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, may 2009.

[32] Eugster, Patrick Th. and Felber, Pascal A. and Guerraoui, Rachid and Kermarrec, Anne-Marie. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[33] Brian Frank, Zach Shelby, Klaus Hartke, Carsten Bormann, et al. Constrained Application Protocol (CoAP), 2012.

[34] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic Service Substitution in Service-Oriented Architectures. In *Proceedings of the IEEE International Conference on Service Computing (SCC)*, pages 3443–3448, 2008.

[35] M. Fredj, A. Zarras, N. Georgantas, and V. Issarny. *Service Intelligence and Service Science*, chapter Dynamic Maintenance of Service Orchestrations, pages 57–78. IGI, 2011.

[36] Freeman, E. and Arnold, K. and Hupfer, S. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd. Essex, UK, UK, 1999.

[37] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[38] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, 2(4), Oct.-Dec.

[39] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, IMC '03. ACM, 2003.

[40] Gregor Gössler and Gwen Salaün. Realizability of choreographies for services interacting asynchronously. In *FACS*, volume 7253 of *LNCS*, pages 151–167. 2012.

[41] W.I. Grosky, A. Kansal, S. Nath, Jie Liu, and Feng Zhao. Senseweb: An infrastructure for shared sensing. *MultiMedia, IEEE*, 14(4), Oct.-Dec.

[42] Matthias Güdemann, Pascal Poizat, Gwen Salaün, and Alexandre Dumont. Verchor: A framework for verifying choreographies. In *FASE*, volume 7793 of *LNCS*, pages 226–230. 2013.

[43] S. Hadim and N. Mohamed. Middleware: middleware challenges and approaches for wireless sensor networks. *Distributed Systems Online, IEEE*, 7(3), March.

[44] Sylvain Hallé and Tevfik Bultan. Realizability analysis for message-based interactions using shared-state projections. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 27–36, 2010.

[45] A. Hornsby, P. Belimpasakis, and I. Defee. Xmpp-based wireless sensor network and its integration into the extended home environment. In *Consumer Electronics, 2009. ISCE '09. IEEE 13th International Symposium on*, May.

[46] Valérie Issarny and Amel Bennaceur. Composing distributed systems: Overcoming the interoperability challenge. In *Submitted for publication*, 2013.

[47] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM*, pages 217–255, 2011.

[48] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadis, Marco Autili, Marco Aurelio Gerosa, and Amira Ben Hamida. Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions. *Journal of Internet Services and Applications*, 2(1):23–45, May 2011.

[49] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2), August 2008.

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

[51] J. Kramer and J. Magee. The Evolving Philosophers Problem. *IEEE Transactions on Software Engineering*, 15(1):1293–1306, 1990.

[52] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1), April 2009.

[53] G. Lam and D. Rossiter. A web service framework supporting multimedia streaming. *Services Computing, IEEE Transactions on*, PP(99), 2012.

[54] S. S. Lam. Protocol conversion. *IEEE Trans. Softw. Eng.*, 14(3):353–362, March 1988.

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

[56] Danh Le-Phuoc, Hoan Quoc Nguyen-Mau, Josiane Xavier Parreira, and Manfred Hauswirth. A middleware framework for scalable management of linked streams. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16(0), 2012.

[57] Danh Le-Phuoc, Josiane Xavier Parreira, and Manfred Hauswirth. Linked stream data processing. In Thomas Eiter and Thomas Krennwallner, editors, *Reasoning Web. Semantic Technologies for Advanced Query Answering*. Springer Berlin / Heidelberg, 2012.

[58] LeonardoA.F. Leite, Gustavo Ansaldi Oliva, GuilhermeM. Nogueira, MarcoAurlio Gerosa, Fabio Kon, and DejanS. Milojicic. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, pages 1–18, 2012.

[59] Tao Lin, Hai Zhao, Jiyong Wang, Guangjie Han, and Jindong Wang. An embedded web server for equipment. In *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, may 2004.

[60] B. Liskov and J.M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(6):1811–1841, 1994.

[61] Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Andreas Kotsopoulos, and Kyle Kim. Tinyrest - a protocol for integrating sensor networks into the internet. In *Proceedings of Workshop on Real-World Wireless Sensor Networks*, June 2005.

[62] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), March 2005.

[63] Jeff Magee and Jeff Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.

[64] Monson-Haefel, R. and Chappell, D. *Java Message Service*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000.

[65] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3), April 2011.

[66] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, New York, NY, USA, 2007. ACM.

[67] Murphy, A.L. and Picco, G.P. and Roman, G.C. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):328, 2006.

[68] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM.

[69] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, New York, NY, USA, 2004. ACM.

[70] M.F. O"Connor, K. Conroy, M. Roantree, A.F. Smeaton, and N.M. Moyna. Querying xml data streams from wireless sensor networks: An evaluation of query engines. In *Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on*, April.

[71] B. Ostermaier, F. Schlup, and K. Römer. Webplug: A framework for the web of things. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 2010.

[72] D. Papadimitriou. Future Internet–The Cross-ETP Vision Document. *European Technology Platform, Alcatel Lucent*, 8, 2009.

[73] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11), 2007.

[74] Jyotishman Pathak, Robyn Lutz, and Vasant Honavar. Moscoe: An approach for composing web services through iterative reformulation of functional specifications. *International Journal on Artificial Intelligence Tools*, 17:109–138, 2008.

[75] Jorge Pereira. From autonomous to cooperative distributed monitoring and control: Towards the Internet of smart things. In *ERCIM Workshop on eMobility*, 2008.

[76] Pascal Poizat and Gwen Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proceedings of SAC 2012*, pages 1927–1934, 2012.

[77] Antony Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web*, 1:167–179, 1998. 10.1023/A:1019263731139.

[78] Antony I. T. Rowstron and Alan Wood. Solving the linda multiple rd problem. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 357–367, London, UK, 1996. Springer-Verlag.

[79] E. Rukzio, M. Paolucci, M. Wagner, H. Berndt, J. Hamard, and A. Schmidt. Mobile service interaction with the web of things. In *13th International Conference on Telecommunications (ICT 2006)*, 2006.

[80] Gwen Salaün. Generation of service wrapper protocols from choreography specifications. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 313–322, 2008.

[81] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[82] Sugoog Shon. Protocol Implementations for Web Based Control Systems. *International Journal of Control, Automation, and Systems*, March 2005.

[83] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *Proceedings of ICSE*, 2003.

[84] P. Stuckmann and R. Zimmermann. European research on future Internet design. *IEEE Wireless Communications*, 16(5), 2009.

[85] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software architecture : foundations, theory, and practice*. Hoboken (N.J.) : Wiley, 2009.

[86] CHOReOS Project Team. CHOReOS State of the Art, Baseline and Beyond - Public Project deliverable D1.1, December 2010.

[87] CHOReOS Project Team. CHOReOS dynamic development model definition - Public Project deliverable D2.1, September 2011.

[88] CHOReOS Project Team. CHOReOS Middleware Specification - Public Project deliverable D3.1, September 2011.

[89] CHOReOS Project Team. CHOReOS Perspective on the Future Internet and Initial Conceptual Model - Public Project deliverable D1.2, March 2011.

[90] CHOReOS Project Team. Initial Architectural Style for CHOReOS Choreographies - Public Project deliverable D1.3, September 2011.

[91] CHOReOS Project Team. CHOReOS Middleware Implementation - Public Project deliverable D3.2.2, September 2012.

[92] CHOReOS Project Team. Definition of the Dynamic Development Process for Adaptable QoS-aware ULS Choreographies, September 2012.

[93] I. Toma, E. Simperl, A. Filipowska, G. Hench, and J. Domingue. Semantics-driven interoperability on the Future Internet. In *IEEE International Conference on Semantic Computing (ICSC)*, 2009.

[94] Vlad Trifa, Dominique Guinard, Vlatko Davidovski, Andreas Kamilaris, and Ivan Delchev. Web messaging for open and scalable distributed sensing applications. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *Web Engineering*. Springer Berlin / Heidelberg, 2010.

[95] Vlad Trifa, Dominique Guinard, Vlatko Davidovski, Andreas Kamilaris, and Ivan Delchev. Web messaging for open and scalable distributed sensing applications. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[96] R. H. Weber and R. Weber. *Internet of Things*. Springer, 2010.

[97] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In Kay Rmer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, pages 5–20. Springer Berlin Heidelberg, 2006.

[98] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3), September 2002.

A Appendix

A.1. Coordination Models of the Airport Use Case: Arrival Handling scenario

$$
\begin{aligned}
M_{CD_{ATC.AIR}}=\{ & \\
& \langle S1, ConfirmApproach, S2, Ask(), \\
& \quad CD\{\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S2, \{\}, S8, Ask(), \\
& \quad CD\{\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S8, \{\}, S3, Ask(), \\
& \quad CD\{AIR.SGM\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S8, \{\}, S94, Ask(), \\
& \quad CD\{\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S94, \{\}, S9, Ask(), \\
& \quad CD\{\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S9, \{\}, S10, Ask(), \\
& \quad CD\{AIR.LHC\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S9, \{\}, S13, Ask(), \\
& \quad CD\{AIR.SC\}, \\
& \quad true, \\
& \quad Notify(), Wait()\rangle, \\
& \langle S9, \{\}, S16, Ask(), \\
& \quad CD\{AIR.ABC\}, \\
& \quad true,
\end{aligned}
$$

$Notify(), Wait()\rangle,$

$\langle S8, \{\}, S95, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S95, \{\}, 74, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle 74, \{\}, S20, Ask(),$
$CD\{AIR.ANSA\},$
$true,$
$Notify(), Wait()\rangle,$

}

$M_{CD_{AIR.SGM}}$={

$\langle S3, RequestAmenities, S4, Ask(),$
$CD\{SGM.AIR\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S5, RequestAmenities, S4, Ask(CD(AIR.AP)\ for\ S91),$
$CD\{SGM.AIR\},$
$true,$
$Notify(), Wait()\rangle,$

}

$M_{CD_{SGM.AIR}}$={

$\langle S4, GetAvailableAmenities, S5, Ask(),$
$CD\{AIR.SGM\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S5, \{\}, S73, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S73, \{\}, S91, Ask(),$
$CD\{AIR.AP\},$
$true,$
$Notify(), Wait()\rangle,$

}

$M_{CD_{AIR.AP}}$={

$\langle S91, BookAllAvailableAmenities, S72, Ask(CD(AIR.SGM)\ for\ S5),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S72, \{\}, S73, Ask(),$

$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,

$\langle S73, \{\}, S91, Ask()$,
$CD\{AIR.AP\}$,
$cond1$,
$Notify(), Wait()\rangle$,

$\langle S73, \{\}, S7, Ask()$,
$CD\{\}$,
$!cond1$,
$Notify(), Wait()\rangle$,

$\langle S7, \{\}, S93, Ask()$,
$CD\{\}$,
$true$,
$Notify(S7\ to\ CD(LHC.AIR, SC.AIR, ABC.AIR, AIR.ASAA))$,
$Wait(S12\ from\ CD(LHC.AIR), S15\ from\ CD(SC.AIR),$
$S18\ from\ CD(ABC.AIR), S90\ from\ CD(AIR.ASAA))\rangle$,

$\langle S93, \{\}, S24, Ask()$,
$CD\{AIR.AISA\}$,
$true$,
$Notify(), Wait()\rangle$,

}

$M_{CD_{AIR.LHC}}$={

$\langle S10, BookAmenity, S11, Ask()$,
$CD\{LHC.AIR\}$,
$true$,
$Notify(), Wait()\rangle$,

}

$M_{CD_{LHC.AIR}}$={

$\langle S11, GetLuggageConfirmation, S12, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,

$\langle S12, \{\}, S19, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,

$\langle S19, \{\}, S71, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,

$\langle S71, \{\}, S93, Ask()$,
$CD\{\}$,
$true$,

$Notify(S12\ to\ CD(AIR.AP, SC.AIR, ABC.AIR, AIR.ASAA)),$
$Wait(S7\ from\ CD(AIR.AP), S15\ from\ CD(SC.AIR),$
$S18\ from\ CD(ABC.AIR), S90\ from\ CD(AIR.ASAA))\rangle,$
$\langle S93, \{\}, S24, Ask(),$
$CD\{AIR.AISA\},$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{AIR.SC}}$={
$\langle S13, BookAmenity, S14, Ask(),$
$CD\{SC.AIR\},$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{SC.AIR}}$={
$\langle S14, GetSecurityConfirmation, S15, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S15, \{\}, S19, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S19, \{\}, S71, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S71, \{\}, S93, Ask(),$
$CD\{\},$
$true,$
$Notify(S15\ to\ CD(AIR.AP, LHC.AIR, ABC.AIR, AIR.ASAA)),$
$Wait(S7\ from\ CD(AIR.AP), S12\ from\ CD(LHC.AIR),$
$S18\ from\ CD(ABC.AIR), S90\ from\ CD(AIR.ASAA))\rangle,$
$\langle S93, \{\}, S24, Ask(),$
$CD\{AIR.AISA\},$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{AIR.ABC}}$={
$\langle S16, BookAmenity, S17, Ask(),$
$CD\{ABC.AIR\},$
$true,$
$Notify(), Wait()\rangle,$
}

$M_{CD_{ABC.AIR}}$={
⟨$S17, GetBusConfirmation, S18, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S18, \{\}, S19, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S19, \{\}, S71, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S71, \{\}, S93, Ask()$,
$CD\{\}$,
$true$,
$Notify(S18\ to\ CD(AIR.AP, LHC.AIR, SC.AIR, AIR.ASAA))$,
$Wait(S7\ from\ CD(AIR.AP), S12\ from\ CD(LHC.AIR),$
$S15\ from\ CD(SC.AIR), S90\ from\ CD(AIR.ASAA))$⟩,
⟨$S93, \{\}, S24, Ask()$,
$CD\{AIR.AISA\}$,
$true$,
$Notify(), Wait()$⟩,
}
$M_{CD_{AIR.ANSA}}$={
⟨$S20, double :: GetAverageSoundLevel, S21, Ask()$,
$CD\{AIR.ASAA\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S36, double :: GetAverageSoundLevel, S44, Ask()$,
$CD\{AIR.ASAA\}$,
$true$,
$Notify(), Wait()$⟩,
}
$M_{CD_{AIR.ASAA}}$={
⟨$S21, AdjustSpeakersVolumeLevel(double), S22, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S22, \{\}, S74, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()$⟩,
⟨$S74, \{\}, S20, Ask()$,

$CD\{AIR.ANSA\},$
$cond2,$
$Notify(), Wait()\rangle,$
$\langle S74, \{\}, S90, Ask(),$
$CD\{\},$
$!cond2,$
$Notify(), Wait()\rangle,$
$\langle S90, \{\}, S93, Ask(),$
$CD\{\},$
$true,$
$Notify(S90\ to\ CD(AIR.AP, LHC.AIR, SC.AIR, ABC.AIR)),$
$Wait(S7\ from\ CD(AIR.AP), S12\ from\ CD(LHC.AIR),$
$S15\ from\ CD(SC.AIR), S18\ from\ CD(ABC.AIR))\rangle,$
$\langle S93, \{\}, S24, Ask(),$
$CD\{AIR.AISA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S44, AdjustSpeakersVolumeLevel(double), S38, Ask(),$
$CD\{\},$
$true,$
$Notify(S38\ to\ CD(AIR.ASAA)), Wait(S41\ from\ CD(AIR.ASAA))\rangle,$
$\langle S45, AdjustSpeakersVolumeLevel(double), S41, Ask(),$
$CD\{\},$
$true,$
$Notify(S41\ to\ CD(AIR.ASAA)), Wait(S38\ from\ CD(AIR.ASAA))\rangle,$
$\langle S41, \{\}, S25, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S38, \{\}, S25, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S25, \{\}, S46, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S46, \{\}, S78, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S78, \{\}, S55, Ask(),$
$CD\{\},$
$cond6,$

$Notify(), Wait()\rangle,$

$\langle S55, \{\}, S35, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S35, \{\}, S36, Ask(),$
$CD\{AIR.ANSA\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S35, \{\}, S39, Ask(),$
$CD\{AIR.MIDMSA\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S78, \{\}, S104, Ask(),$
$CD\{\},$
$!cond6,$
$Notify(), Wait()\rangle,$

$\langle S104, \{\}, FinalState, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S67, Say(string), S68, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S68, \{\}, S80, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S80, \{\}, S67, Ask(),$
$CD\{AIR.ASAA\},$
$cond8,$
$Notify(), Wait()\rangle,$

$\langle S80, \{\}, S107, Ask(),$
$CD\{\},$
$!cond8,$
$Notify(), Wait()\rangle,$

$\langle S107, \{\}, FinalState, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S69, SetSignStates(arrayList), S70, Ask(),$
$CD\{\},$

$true$,
$Notify(), Wait()\rangle$,
$\langle S70, \{\}, S81, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,
$\langle S81, \{\}, S69, Ask()$,
$CD\{AIR.ASAA\}$,
$cond9$,
$Notify(), Wait()\rangle$,
$\langle S81, \{\}, S107, Ask()$,
$CD\{\}$,
$!cond9$,
$Notify(), Wait()\rangle$,
$\langle S107, \{\}, FinalState, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,
}
$M_{CD_{AIR.AISA}}$={
$\langle S24, boolean :: InitiatePassengersTracking, S30, Ask()$,
$CD\{AISA.AIR\}$,
$true$,
$Notify(), Wait()\rangle$,
$\langle S53, booleanArray :: GetLandingStates, S54, Ask()$,
$CD\{\}$,
$true$,
$Notify(), Wait()\rangle$,
$\langle S54, \{\}, S77$,
$CD\{\}, Ask()$,
$true$,
$Notify(), Wait()\rangle$,
$\langle S77, \{\}, S53$,
$CD\{\}, CD\{AIR.AISA\}$,
$cond5$,
$Notify(), Wait()\rangle$,
$\langle S77, \{\}, S101$,
$CD\{\}, Ask()$,
$!cond5$,
$Notify(), Wait()\rangle$,
$\langle S101, \{\}, FinalState$,
$CD\{\}, Ask()$,
$true$,
$Notify(), Wait()\rangle$,

}

$M_{CD_{AISA.AIR}}$={

$\langle S30, int :: ReportPassedPassengers, S26, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S26, \{\}, S27, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S27, \{\}, S56, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S56, \{\}, S28, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S28, \{\}, S29, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S28, \{\}, S31, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S28, \{\}, S33, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S29, \{\}, S75, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S31, \{\}, S76, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S33, \{\}, S77, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S75, \{\}, S49, Ask(),$
$CD\{AIR.APSA\},$

$true,$
$Notify(), Wait()\rangle,$
$\langle S76, \{\}, S51, Ask(),$
$CD\{AIR.MIDLSA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S77, \{\}, S53, Ask(),$
$CD\{AIR.AISA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S27, \{\}, S96, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S96, \{\}, S78, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S78, \{\}, S55, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S55, \{\}, S35, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S35, \{\}, S36, Ask(),$
$CD\{AIR.ANSA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S35, \{\}, S39, Ask(),$
$CD\{AIR.MIDMSA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S27, \{\}, S57, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S57, \{\}, S42, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S42, \{\}, S59, Ask(),$

$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S42, \{\}, S58, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S59, \{\}, S43, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S43, \{\}, S62, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S43, \{\}, S63, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S43, \{\}, S64, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S62, \{\}, S79, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S63, \{\}, S80, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S64, \{\}, S81, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S58, \{\}, S82, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S79, \{\}, S65, Ask(),$
$CD\{AIR.ADAA\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S80, \{\}, S67, Ask(),$
$CD\{AIR.ASAA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S81, \{\}, S69, Ask(),$
$CD\{AIR.ADAA\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S82, \{\}, S60, Ask(),$
$CD\{AIR.MIDDAA\},$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{AIR.APSA}}$={
$\langle S49, booleanArray :: GetLandingStates, S50, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S50, \{\}, S75,$
$CD\{\}, Ask(),$
$true,$
$Notify(), Wait()\rangle,$
$\langle S75, \{\}, S49,$
$CD\{\}, CD\{AIR.APSA\},$
$cond3,$
$Notify(), Wait()\rangle,$
$\langle S75, \{\}, S102,$
$CD\{\}, Ask(),$
$!cond3,$
$Notify(), Wait()\rangle,$
$\langle S102, \{\}, FinalState,$
$CD\{\}, Ask(),$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{AIR.MIDLSA}}$={
$\langle S51, intArray :: GetAveragePosition, S52, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
$\langle S52, \{\}, S76,$
$CD\{\}, Ask(),$
$true,$
$Notify(), Wait()\rangle,$

$\langle S76, \{\}, S51,$
$CD\{\}, CD\{AIR.MIDLSA\},$
$cond4,$
$Notify(), Wait()\rangle,$

$\langle S76, \{\}, S103,$
$CD\{\}, Ask(),$
$!cond4,$
$Notify(), Wait()\rangle,$

$\langle S103, \{\}, FinalState,$
$CD\{\}, Ask(),$
$true,$
$Notify(), Wait()\rangle,$

}

$M_{CD_{AIR.ADAA}}$={

$\langle S65, Display(string), S66, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S66, \{\}, S79, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S79, \{\}, S65, Ask(),$
$CD\{AIR.ADAA\},$
$cond7,$
$Notify(), Wait()\rangle,$

$\langle S79, \{\}, S106, Ask(),$
$CD\{\},$
$!cond7,$
$Notify(), Wait()\rangle,$

$\langle S106, \{\}, FinalState, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

}

$M_{CD_{AIR.MIDDAA}}$={

$\langle S60, Display(string), S61, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S61, \{\}, S82, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$

$\langle S82, \{\}, S60, Ask(),$
$CD\{AIR.MIDDAA\},$
$cond10,$
$Notify(), Wait()\rangle,$
$\langle S82, \{\}, S105, Ask(),$
$CD\{\},$
$!cond10,$
$Notify(), Wait()\rangle,$
$\langle S105, \{\}, FinalState, Ask(),$
$CD\{\},$
$true,$
$Notify(), Wait()\rangle,$
}
$M_{CD_{AIR.MIDMSA}}$={
$\langle S39, double :: GetAverageSound, S45, Ask(),$
$CD\{AIR.ASAA\},$
$true,$
$Notify(), Wait()\rangle,$
}