

DOCTORAL THESIS

**MUSYC: A Model-Driven Methodology to Develop
Ubiquitous Systems**

Author:

Carlos Rodríguez Domínguez

Supervisors:

José Luis Garrido Bullejos

Kawtar Benghazi Akhlaki

UNIVERSITY OF GRANADA

Department of Computer Languages and Systems

2014

Editor: Editorial de la Universidad de Granada
Autor: Carlos Rodríguez Domínguez
D.L.: GR 1948-2014
ISBN: 978-84-9083-113-7

El doctorando Carlos Rodríguez Domínguez y los directores de la tesis José Luis Garrido Bullejos y Kawtar Benghazi Akhlaki garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis y, hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se han utilizado sus resultados o publicaciones.

Granada, 11 de abril de 2014.

Directores de la Tesis

José Luis Garrido Bullejos Kawtar Benghazi Akhlaki

Doctorando

Carlos Rodríguez Domínguez

The PhD candidate Carlos Rodríguez Domínguez and the thesis supervisors José Luis Garrido Bullejos and Kawtar Benghazi Akhlaki guarantee, by signing this thesis, that the work has been done by the PhD candidate under the guidance of the directors of the thesis and, as far as our knowledge reaches, in the realization of the work, the copyrights of the cited authors have been respected, when their results or publications have been used.

Granada, April 11, 2014.

Thesis Supervisors

José Luis Garrido Bullejos Kawtar Benghazi Akhlaki

PhD Candidate

Carlos Rodríguez Domínguez

To my supervisors, José Luis and Kawtar. Thanks for all your support during these years, your patience and your care. To Ony, who is the key figure of my life. To my family in general (which includes my friends), and to my parents, brother, grand mother, uncles, aunts and cousins in particular. To Álvaro, Manolo and Tomás, because you have the best degree that can be obtained: you are true friends. To the rest of the members of MYDASS research group, because you have supported and encouraged me from the beginning. To Federico, for his constant help, energy and teachings. To my best friends Jaby and Gilberto (and their respective partners), for being such crucial people in my life, and for sharing with me so many life-changing moments. We will always be “partners in crime”.

Resumen

En 1991, Mark Weiser visionó un estilo de vida futuro que giraría en torno a dispositivos de cómputo que se integrarían de manera transparente en nuestros entornos físicos. Debido a la amplia disponibilidad de dispositivos pequeños, móviles y asequibles (*smartphones*, *tablets*, etc.) que son capaces de ayudarnos en nuestras tareas diarias, la idea de Weiser, también conocida como el *paradigma de la computación ubicua*, se puede considerar hoy día como el “siguiente paso” en la esperada evolución de las tecnologías actuales.

A pesar de ello, el desarrollo de un sistema ubicuo (i.e., sistemas software diseñados conforme al paradigma de la computación ubicua) es todavía un reto, y su complejidad está parcialmente ligada a la gestión de los mecanismos asociados a las comunicaciones (tecnologías de red, protocolos software, etc.).

Técnicamente, la complejidad a la hora de gestionar las comunicaciones apropiadamente se ha relacionado directamente con su naturaleza espontánea y volátil, debido a que los usuarios de estos sistemas están en continuo movimiento mientras portan sus dispositivos de cómputo. Sin embargo, otros problemas pueden complicar también la gestión de las comunicaciones y el desarrollo de sistemas

ubicuos.

Particularmente, no hay conceptualizaciones bien establecidas acerca de los elementos que deberían formar parte de un sistema ubicuo. Por tanto, los mecanismos (y sus propiedades) para dar soporte a las comunicaciones en un sistema ubicuo no están definidos explícitamente.

Consecuentemente, debido a la ausencia de modelos bien establecidos, es complicado desarrollar metodológicamente un sistema ubicuo cuyo diseño capture todos los requisitos del usuario y que pueda ser compartido (y entendido) por diferentes diseñadores de software. Por consiguiente, diferentes diseñadores pueden idear mecanismos de comunicación heterogéneos que, finalmente, pueden presentar un bajo nivel de interoperabilidad y compatibilidad.

Más aún, las diferentes plataformas subyacentes que pueden existir para ocuparse de la complejidad a la hora de gestionar las comunicaciones en sistemas ubicuos (como middleware), de nuevo, no pueden basarse en modelos bien establecidos. En consecuencia, su uso requiere unos amplios conocimientos técnicos, y los conceptos presentes en algunas de estas plataformas pueden no estar presentes en otras, o pueden incluso tener una semántica diferente.

Esta tesis propone una conceptualización de los sistemas ubicuos, con un particular énfasis en los mecanismos que deben estar

presentes para dar soporte y gestionar las comunicaciones en estos sistemas. La conceptualización está basada en la noción más abstracta de *sistema de comunicación*, que se ha propuesto para extender y completar las teorías de comunicación existentes. Ambas conceptualizaciones de un sistema de comunicación y de un sistema ubicuo se han formalizado a través de ontologías.

Sobre la base de las conceptualizaciones definidas y un conjunto de reglas de transformación, se ha propuesto una metodología dirigida por modelos, llamada MUSYC, para el desarrollo de sistemas ubicuos. MUSYC ha sido validada a través de diversos proyectos de I+D y de un middleware para sistemas ubicuos llamado BlueRose, que, a su vez, muestra que MUSYC es también apropiado para el desarrollo de tecnologías de soporte para facilitar la gestión de las comunicaciones en este tipo de sistemas.

Abstract

In 1991, Mark Weiser envisioned a future lifestyle centered around computing devices that would be seamlessly integrated into our physical environments. Due to the wide availability and success of small, mobile and affordable devices (smartphones, tablets, etc.) that are able to assist us during our daily tasks, Weiser's idea, also known as the *ubiquitous computing paradigm*, can be considered nowadays as the "next step" in the expected evolution of the current technologies.

However, the development of a ubiquitous system (i.e., a software system designed according to the ubiquitous computing paradigm) is still challenging, and its complexity is partially related to the appropriate management of the mechanisms associated with the communications (networking technologies, software protocols, etc.).

Technically, the complexity to appropriately manage the communications has been directly linked to their spontaneous and highly volatile nature, since the users of these systems are in constant movement while they are carrying their computing devices. Nonetheless, other different problems may also complicate

the management of the communications and the development of ubiquitous systems.

Particularly, there are not any well-established conceptualizations of the elements that should be present in a ubiquitous system. Therefore, the mechanisms (and their properties) supporting the communications in a ubiquitous system are not defined explicitly.

In consequence, due to the absence of well established models, it is challenging to methodologically develop a ubiquitous system whose design captures all the users' requirements and that can be shared (and understood) among different software designers. Hence, different designers may devise heterogeneous communication mechanisms that, ultimately, may present a low degree of interoperability and compatibility.

Moreover, the different underlying platforms that may exist to deal with the complexity of managing the communications in ubiquitous systems (like middleware), again, can not be based on well established models. Therefore, their use requires a high technical expertise, and the concepts present in some of these platforms may not be present in others, or they may even have different semantics.

This thesis work proposes a conceptualization of the ubiquitous systems, with a particular emphasis on the mechanisms that

should be present to support and manage the communications in these systems. The conceptualization is based on the more abstract notion of *communication system*, which has been proposed to extend and complete the existing communication theories. Both conceptualizations of a communication system and a ubiquitous system have been formalized through ontologies.

On the basis of the defined conceptualizations and a set of transformation rules, a model-driven methodology to develop ubiquitous systems, called MUSYC, has been proposed. MUSYC has been validated through the development of several R&D projects and a middleware for ubiquitous systems called BlueRose, which, in turn, shows that MUSYC is also suitable to develop supporting technologies to facilitate the management of the communications in these systems.

Contents

List of Figures	XIX
List of Tables	XXV
Prologue	XXVII
CHAPTER	Page
1. Introduction	1
1.1. The Ubiquitous Computing Era	1
1.2. Description of the Problem and Motivation	3
1.3. Hypothesis and Objectives	5
1.4. Structure of the Thesis	6
2. Foundations for the Specification and Development of Ubiquitous Systems	9
2.1. Distributed Systems	10
2.1.1. Architectures of a Distributed System	11
2.1.2. Communication Paradigms	16
2.1.3. Some Notations for Representing Distributed Systems	23
2.1.4. Supporting Communications: Middleware	28
2.2. Ubiquitous Systems	36
2.2.1. Context Awareness	38
2.2.2. Communication Paradigms in Ubiquitous Systems	41
2.2.3. Middleware Technologies for Ubiquitous Systems	48

2.2.4.	Communications in Ubiquitous Systems: Technical Issues	51
2.3.	Model-Driven Engineering (MDE)	55
2.3.1.	Model-Driven Architecture (MDA)	60
2.3.2.	A Comparison between MDE and Code-Centric Developments	63
2.3.3.	Developing Communication Mechanisms on the basis of MDE	68
2.4.	Conclusions	70
3.	A Model for Communication Systems	73
3.1.	An Introduction to the Communication Theory	74
3.2.	A Model to Conceptualize a Communication System	77
3.2.1.	Structural View	78
3.2.2.	Behavioral View	85
3.2.3.	Formal Specification as an Ontology	93
3.3.	Quality Attributes of the Communication Model	97
3.4.	Conclusions	100
4.	A Communication Model for Ubiquitous Systems	103
4.1.	Communication Functionalities of a Ubiquitous System	104
4.2.	General Communication Model for Ubiquitous Sys- tems	107
4.2.1.	Structural View	109
4.2.2.	Behavioral View	131
4.2.3.	Formal Specification as an Ontology	138
4.2.4.	Ontological Representation of a Ubiquitous System as a Communication System	143
4.3.	Quality Attributes of the Communication Model for Ubiquitous Systems	147
4.4.	Conclusions	148
5.	MUSYC: An MDA-based Methodology to Develop Ubiquitous Systems on the Basis of the Communications	153
5.1.	Overview	154
5.2.	Stage 1: Communication Requirements Analysis	162
5.2.1.	Initial Analysis through Use Cases and Choreography Models	163

5.2.2.	CS-CIM Specification	166
5.3.	Stage 2: Ubiquitous System Design	175
5.3.1.	CS-CIM to US-PIM Transformation: Structural View	177
5.3.2.	CS-CIM to US-PIM Transformation: Behavioral View	186
5.4.	Stage 3: Implementation of the Ubiquitous System .	193
5.4.1.	Transformation from a US-PIM to a US-PSM	194
5.4.2.	Code Generation from a US-PSM	196
5.5.	CASE Tools Supporting MUSYC	199
5.6.	Conclusions	205
6.	Validation of MUSYC through the Development of a Middleware and a Software Framework for Ubiquitous Systems: BlueRose	209
6.1.	Applying MUSYC to the Development of Middleware Solutions for Ubiquitous Systems	210
6.1.1.	Communication Requirements Analysis	211
6.1.2.	Ubiquitous System Requirements Analysis	216
6.1.3.	Implementation	224
6.2.	BlueRose as a Software Framework for the Development of Ubiquitous Systems	226
6.2.1.	Structural Elements	227
6.2.2.	Behavioral Elements	234
6.3.	Quality Attributes of BlueRose	235
6.3.1.	Performance Efficiency	235
6.3.2.	Additional Quality Attributes	240
6.4.	Practical Validation	247
6.4.1.	Mobile Forensic Workspace	247
6.4.2.	VIRTRA-EL: A Web Platform to Support a Collaborative Virtual Training for Elderly People	253
6.4.3.	Domo and Kora: Management of Home Automation Environments	256
6.4.4.	Sherlock: A Location Service for Both Outdoors and Indoors	259
6.5.	Conclusions	260
7.	Conclusions	263

7.1. Results and Discussion	263
7.1.1. Conceptualization	264
7.1.2. Methodology	267
7.1.3. Technology	269
7.2. Future Work	271
Acknowledgements	275
Publications	277
Patents	283
Bibliography	285
Appendices	301
I. Implementation of the CI-CS ontology in OWL	301
II. Quality Attributes of the CI-CS Metamodel	328
III. Implementation of the PI-US Ontology in OWL	337
IV. Quality Attributes of the PI-US Metamodel	374
V. Detailed SPEM 2.0 diagram describing the develop- ment process proposed in MUSYC	383
VI. Proposed QVT rules to transform a CS-CIM into a US-PIM	384
VII. ATL transformation rules that can be applied to the behavioral view of a CS-CIM to produce a UML se- quence diagram	398
VIII. Specification of a CS-CIM for BlueRose middleware	420
IX. Specification of a US-PIM for BlueRose middle- ware, automatically obtained from the CS-CIM through the proposed QVT transformation rules	424

List of Figures

2.1. A SoaML model example of a search service	14
2.2. Pull-based communication model	21
2.3. Push-based communication model	21
2.4. An example Petri net	23
2.5. An example of a UML 2.x communication diagram .	25
2.6. An example of a UML 2.x activity diagram	26
2.7. An example of a BPMN choreography diagram and a summary of its elements	27
2.8. An example of a MANET with three devices: There is a total connection between them since the Node B can route the data transmissions between A and C . .	52
2.9. Relationship between MDE, MDD and other methodologies	56
2.10. Graphical illustration of the MDE development methodology	58
2.11. Metamodel-based transformations in Model-Driven Architecture (MDA)	61
2.12. Scheme of the MDE process	64
2.13. Scheme of the code-centric development process . .	65
3.1. The communication model proposed by Shannon and Weaver in the <i>The Mathematical Theory of Communication</i> , 1949 [115]	75
3.2. The communication model described by Berlo in the <i>The Process of Communication</i> , 1960 [11]	75
3.3. The communication model described by Barnlund in <i>A Transactional Model of Communication</i> , 1970 (re-printed in 2008 [6])	76

3.4.	A UML activity diagram depicting the dynamic behavior of a CI-CS	79
3.5.	The structural view of the CI-CS metamodel, depicted as a UML class diagram	81
3.6.	Behavioral view of the CI-CS metamodel, depicted as a UML class diagram. The model is inspired by the BPMN 2.0 Choreography Metamodel Specification [89]	88
3.7.	Graphical representation of an ontology of a CI-CS	94
3.8.	The class hierarchy and properties of an ontology of a CI-CS, as represented in Protégé	95
4.1.	Simplified interaction process involved in the message exchanging communication functionality of a ubiquitous system	110
4.2.	Assumed interaction process involved in the event distribution communication functionality of a ubiquitous system	112
4.3.	Assumed interaction process to dynamically discover participants in a ubiquitous system	114
4.4.	A UML class diagram depicting the structural view of the PI-US metamodel	116
4.5.	An extract of the UML class diagram representing the PI-US metamodel, with a focus on the adopted event model	124
4.6.	A UML class diagram depicting the behavioral view of the PI-US metamodel	133
4.7.	A UML sequence diagram representing how software agents interact during a communication activity, as it is assumed in the proposed PI-US metamodel	136
4.8.	An ontology of the communication mechanisms supporting a PI-US	141
4.9.	The class hierarchy and properties of an ontology of the communication mechanisms supporting a PI-US, as represented in Protégé	142
4.10.	Some screenshots of Protégé that show how a reasoner can automatically infer that a ubiquitous system is a communication system	146

5.1. MDA approach to the development of ubiquitous systems	156
5.2. A simplified description of the development process proposed in MUSYC, depicted as an SPEM 2.0 diagram	158
5.3. Overall development process specified in MUSYC	159
5.4. First development stage specified in MUSYC	164
5.5. A sample UML use case model that could be specified during the initial analysis of a Ubiquitous Medical Environment	165
5.6. A sample choreography specified during the initial analysis of a Ubiquitous Medical Environment, depicted as a BPMN 2.0 Choreography diagram	167
5.7. A UML activity diagram that specifies the process that should be followed to identify the elements present in the structural view of a CS-CIM, as it has been defined in MUSYC	169
5.8. A UML class diagram depicting the structural elements of the sample UME as an instance of the elements present in the structural view of the CS-CIM metamodel	170
5.9. Second development stage specified in MUSYC	177
5.10. An excerpt of the QVT rules to transform a participant of a CS-CIM into a software agent in a US-PIM	178
5.11. An excerpt of the QVT rules to include the communication functionalities of a ubiquitous system to each transformed software agent	181
5.12. An excerpt of the QVT rules to transform a message of a CS-CIM into the corresponding elements in a US-PIM	182
5.13. An excerpt of the result of applying the QVT transformation rules to the sample CS-CIM of a UME	184
5.14. An example of a transformation from a BPMN choreography diagram into a sequence diagram	187
5.15. An excerpt of the QVT rules to transform a choreography of a CS-CIM into a choreography in a US-PIM	187

5.16. An excerpt of the QVT rules to transform a choreography activity of a CS-CIM into a set of choreography activities in a US-PIM	188
5.17. An excerpt of the QVT rules to transform a link of a CS-CIM into choreography activities in a US-PIM . . .	190
5.18. An excerpt of the result of applying the QVT transformation rules to the sample CS-CIM of a UME . . .	192
5.19. Third development stage specified in MUSYC	194
5.20. Some example QVT rules to transform from software agents in a US-PIM to the corresponding elements in an unspecified target US-PSM	197
5.21. A sample UME defined using the implementation of an Eclipse Plug-in to define and check CS-CIMs	200
5.22. The result of transforming the CS-CIM of a UME into a US-PIM using QVT rules, depicted using the implementation of an Eclipse Plug-in to define and check US-PIMs	202
5.23. A sample transformation from an event listener defined in an undefined US-PSM to Java code, implemented in MOFM2T standard notation	203
5.24. An excerpt of a sample WSDL service interface automatically derived from a US-PSM	204
6.1. A use case model representing the functionalities that are carried out by proxies and servants in BlueRose middleware	213
6.2. The elements in the behavioral view of a CS-CIM supporting the design of the BlueRose middleware, represented as a BPMN 2.0 Choreography	214
6.3. The elements in the structural view of a CS-CIM supporting the design of the BlueRose middleware, represented as a UML class diagram	215
6.4. Some structural elements, represented in XMI notation, of the US-PIM that results from the transformation of the BlueRose CS-CIM with the proposed QVT rules	219

6.5. Some behavioral elements, represented in XMI notation, of the US-PIM that results from the transformation of the BlueRose CS-CIM with the proposed QVT rules	222
6.6. An excerpt of the US-PIM in XMI standard notation representing the different conditional events that are delivered to activate the corresponding activities in BlueRose	223
6.7. The XMI result of transforming the applications and services in the US-PIM to the corresponding elements in the US-PSM	225
6.8. New elements incorporated to the BlueRose design and related to the message exchanging functionality, depicted as a UML class diagram	228
6.9. Run-time operation of the proposed <i>BRBroker</i> , depicted as a UML sequence diagram	231
6.10. The <i>semantic servant</i> present in the BlueRose design, depicted as a UML class diagram	232
6.11. A comparison of the amount of time that is needed by CORBA, ICE and BlueRose to complete the same benchmark	238
6.12. A comparison of the average throughputs (messages per second) of CORBA, ICE and BlueRose	239
6.13. A comparison of the amount of memory that is needed by CORBA, ICE and BlueRose to complete the same benchmark	239
6.14. A comparison of the average CPU use of CORBA, ICE and BlueRose to complete the same benchmark	240
6.15. Deployment Architecture for Mobile Forensic Workspace	249
6.16. The Mobile Forensic Workspace in iOS devices	253
6.17. Component-based architecture of VIRTRA-EL	255
6.18. An iPhone application acting as a client of the Domo service	257
6.19. Two sample adaptations of the Kora user interface for different users	259
6.20. Overview of the architecture of the Sherlock positioning service	260

List of Tables

2.1. Quality properties promoted by the PubSub and the RR paradigms	43
2.2. Some of the most remarkable middleware technologies for ubiquitous systems	50
3.1. Concepts present in the structural view of a CI-CS	82
3.2. Relationships between the concepts present in the structural view of a CI-CS	83
3.3. Definition of the concepts present in the behavioral view of a CI-CS	89
3.4. Description of the relationships between the concepts present in the behavioral view of a CI-CS	90
3.5. Quality attributes of the CI-CS metamodel	100
4.1. Description of the elements of the PI-US metamodel that are shared between the different communication functionalities supported by a ubiquitous system	117
4.2. Description of the elements of the PI-US metamodel, grouped by the communication functionalities that they support	118
4.3. Description of the relationships between the elements present in the structural view of the PI-US metamodel	119
4.4. Definition of the elements that are present in the behavioral view of the PI-US metamodel. Some of the elements shared with the structural view are not described again	134

4.5.	Description of the relationships between the elements of the behavioral view of the PI-US metamodel	135
4.6.	Quality attributes of the PI-US metamodel	148
5.1.	Description of the matchings between a BPMN 2.0 Choreography and the concepts in the behavioral view of the CS-CIM metamodel	175
5.2.	Description of the matchings between the gateways of a BPMN 2.0 Choreography and the concepts of the behavioral view of the CS-CIM metamodel . . .	176
5.3.	Description of the matchings between the events of a BPMN 2.0 Choreography and the concepts of the behavioral view of the CS-CIM metamodel	176

Prologue

Chapter 1

Introduction

1.1. The Ubiquitous Computing Era

In 1991, Mark Weiser envisioned a future lifestyle centered around small, mobile and continuously connected devices that would be seamlessly integrated into our physical environments [127]. This visionary computational paradigm was called *Ubiquitous Computing*. At that time, in the 90's, personal computers were the predominant computing devices. Personal computers were shared by small groups of people (family, friends, etc.) and were rarely connected to the Internet. As time passed by, personal computers became smaller, more affordable, easier to use and more “connected”. By the early 00's, nearly each person owned a personal computer. At the same time, Internet connections were

popularized and many persons became accustomed to browse the Web to retrieve and share information.

In the mid 00's, the so called “smartphones”, that is, small, mobile, phone-like computers with a permanent wireless Internet connection, became more affordable and powerful, consequently, also becoming very popular. Considering that people were familiar with Internet browsing and information sharing with other people through computers, these devices, which simplified those tasks, were quickly integrated in a lot daily routines: working, socializing, playing, traveling, etc. Suddenly, everyone was surrounded by all types of computing technologies derived from smartphones (tablets, portable consoles, etc.), at anytime and everywhere. From that moment on, personal computers started to be replaced by those new technologies, thus beginning the so called *post-PC era*.

In any case, the progress has not stopped. The Moore's law has proven to be highly accurate [76]: hardware devices become more powerful, cheaper and smaller at a very steady rate. Weiser's vision, that is, the transparent integration of highly connected, mobile and small devices into our physical environments, which was considered in the 90's to be science fiction rather than actual science, nowadays it can just be considered the “next step” in the expected evolution of the current technologies. Pervasive Internet connections and the integration of very advanced computing technologies into everyday

objects (glasses, watches, furniture, clothing, etc.) will point out the rise of the *ubiquitous computing era* in the next few years.

1.2. Description of the Problem and Motivation

As it was defined by Weiser, a ubiquitous system¹ is comprised by a set of software and hardware entities with whom the user transparently interacts.

At software level, those entities are either applications, directly manipulated by the user, or services, which could serve as information providers to the applications. As a result, in ubiquitous systems, applications and services have to continuously exchange information in order to expose complex capabilities to the end users. Thus, managing communications is a key aspect of the development of any ubiquitous system.

However, the complexity of managing the communications in these systems has been commonly associated to the constant movement of the users while they are carrying the devices in which the applications are executed, which makes the communications spontaneous and highly volatile. Nonetheless, other different problems

¹To simplify, in this thesis, software systems designed according to the ubiquitous computing paradigm are simply referred as *ubiquitous systems*

associated to the ubiquitous systems may also be related to the complexity of managing the communications in these systems.

For instance, conceptually, there are not any well established models representing the elements that should be present in a ubiquitous system [32]. Therefore, the mechanisms (and their properties) supporting the communications in a ubiquitous system are not defined explicitly.

In consequence, due to the absence of well established models, it is challenging to methodologically develop a ubiquitous system whose design captures all the users' requirements and that can be shared (and understood) among different software designers. Hence, different designers may devise heterogeneous communication mechanisms that, ultimately, may present a low degree of interoperability and compatibility.

Moreover, the different underlying platforms that may exist to deal with the complexity of managing the communications in ubiquitous systems, again, can not be based on well established models. Therefore, their use requires a high technical expertise, and the concepts present in some of these platforms may not be present in others, or they may even have different semantics.

1.3. Hypothesis and Objectives

In this thesis, it is considered that **the correct management of the communications in ubiquitous systems is a problem that may not only be related to the availability and use of the appropriate technologies at implementation level, but to the conceptualization and methodological development of the ubiquitous systems themselves.** In consequence, the objectives of this thesis are to:

- Define models for conceptualization of the communications in general, and for the communication requirements found in ubiquitous systems in particular.
- Devise a methodology to systematically approach the development of ubiquitous systems, with a special focus on the communication management and using a set of proposed transformation rules applied to the previous models.
- Systematize (and automatize) some of the tasks associated to the design and implementation of technologies to support the interoperability between the applications and services in ubiquitous systems.
- Demonstrate that it is feasible to appropriately manage the communications in a ubiquitous system using both the conceptualizations and the methodological approach to the devel-

opment of these systems.

1.4. Structure of the Thesis

This thesis work is structured as follows.

In this chapter, the initial motivation, hypothesis and objectives of this research work have been established.

In Chapter 2, some previous works are presented in order to provide the suitable background about communication management, ubiquitous systems and model-driven development. Their analysis also motivates this thesis work.

Chapter 3 describes a general conceptual model for the communications, without focusing on the particularities of the communications in ubiquitous systems. The general idea is to define some aspects of the communications that are not captured by the existing communication theories.

In Chapter 4, a communication model for ubiquitous systems is proposed by using the conceptual foundations devised in previous chapter.

On the basis of the conceptual models presented in Chapters 3 and 4, a methodology to develop ubiquitous systems is presented in 5.

In Chapter 6 it is described how the proposed methodology can also be used to develop supporting technologies to assist during the implementation of ubiquitous systems.

Finally, Chapter 7 presents the conclusions and results drawn from this thesis work, and proposes some lines of future work.

Chapter 2

Foundations for the Specification and Development of Ubiquitous Systems

In this chapter, the methods, techniques and technologies that are currently applied to the development of ubiquitous systems are explored. Considering that this thesis work focuses on the communication, the existing bibliography related to that field will be specifically approached, even if it relates to the more general concept of **distributed systems**. Nonetheless, several basic concepts related to the software communication research field will be introduced. Likewise, the notion of **middleware** will be presented as a supporting technology related to the communications in distributed systems and, consequently, in ubiquitous computing environments. Moreover, the relationship between middleware, software frame-

works and design patterns will be explored, so as to highlight the influence that a middleware may have during the design of a software. In addition, some different, well-known middleware technologies specifically designed for ubiquitous systems will be briefly described.

Furthermore, Model-Driven Engineering (MDE) is introduced as a means to develop software on the basis of abstract models. Particularly, an standard approach to MDE known as Model-Driven Architecture (MDA) is described. Additionally, a comparison between MDE and code-centric development processes is provided in order to emphasize the benefits of using MDE in software development, but also standing out some of its negative aspects. Specifically, the advantages of using this technique to develop communication mechanisms are also highlighted. Finally, it is provided a summary of some standard notations that may help to model the communication aspects of a system.

2.1. Distributed Systems

A distributed system is a software system consisting in a set of different, non-located processes that communicate each other by exchanging messages [64]. The theoretical research work associated with this field in very extent, and closely related to the field of

the concurrent systems. However, in this thesis work, the following subsections provide a glimpse of the most important notions related to the communications in distributed systems.

2.1.1. Architectures of a Distributed System

This subsection explores the main reference architectures that are currently applied to the design of distributed systems. During the description of these reference architectures two notions that are widely present in the design of these systems are introduced: **services** and **events**.

2.1.1.1. Service-Oriented Architecture (SOA)

Currently, in the field of the distributed systems, the term **service** is one of the most outstanding concepts that it is possible to encounter. A service is a mechanism to access certain functionalities, whose implementation is opaque to any entity that is external to the service itself and that can be only accessed through a pre-defined public interface [84]. Services mainly foster reusability and maintainability, among other quality properties.

The term **WebService**, which was introduced by the W3C group [16], refers to a service that can be accessed through Internet and whose underlying technologies are related to eXtensive Markup

Language (XML) [19]. That way, their public interface is defined in WebService Description Language (WSDL) [26] and they use Simple Object Access Protocol (SOAP) to exchange information [47], which are, in fact, technologies based on XML.

WebServices have motivated a shift from the perspective that all the resources should be locally available in a personal computer [128]. It is becoming increasingly common to use an Internet connection to access different services providing several resources: storage, applications, improved calculation capabilities, etc. This is known as **cloud computing** [50].

Services can also be considered to be **context-aware** if they have the capability to adapt their own operation depending on the context that surrounds a service requester [57]. This type of services are currently very popular, specially due to the success of the mobile systems. As an example, iAd (<http://advertising.apple.com/>) and AdMob (<http://www.admob.com>) services, respectively provided by Apple and Google, offer commercial advertisements adapted to the user location.

The Service Oriented Architecture (SOA) is a software architecture design paradigm that promotes the encapsulation of certain application functionalities as different interoperating services. Its reference model was defined by the OASIS committee [84]. The

same committee has established that SOA is a paradigm to arrange functionalities (as services) that could be under the control of different organizations or domains. In this architecture, sometimes there will not be a one-to-one relationship between services and functionalities. On the contrary, providing a certain functionality may involve the interaction between several services. In fact, one of the key notions in SOA is the **interaction**, which, within that scope, is defined as the required activity to provide a functionality. Moreover, in the SOA reference model [84], services are recommended to be designed as loosely coupled entities: they should be separately implemented and managed, just using a shared infrastructure to allow their interaction. Therefore, services have a strong cohesion, since they usually depend on other services to provide their functionalities.

In relation to the design of software systems using SOA, the Object Management Group (OMG, <http://www.omg.org>) committee has specified a Unified Modeling Language (UML, <http://www.uml.org>) profile and metamodel, known as SoaML [91]. As an example, in Figure 2.1, it is shown a search service that has been modeled using SoaML.

In that figure, it is shown how a service is modeled as a system component called *Participant*, which may have several service points (*ServicePoint*) and request points (*RequestPoint*):

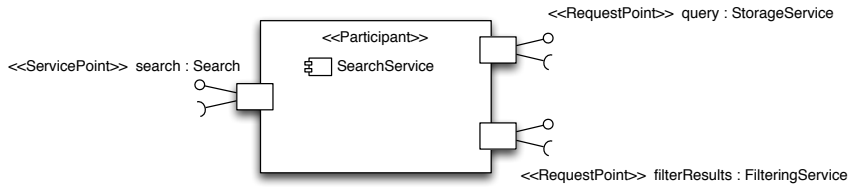


Figure 2.1: A SoaML model example of a search service

- The *Participants* are components that provide or consume services in a system.
- The *ServicePoints* are interaction points from other services to the modeled one.
- The *RequestPoints* are interaction points from the modeled service to others.

In the sample illustration, the search service is a *Participant* that provides a *search ServicePoint* and that requires to interact with storage and filtering services through the *query* and *filter RequestPoints*.

2.1.1.2. Event-Driven Architecture (EDA)

The Event-Driven Architecture (EDA) complements SOA by introducing services whose interaction is based on **events** [75]. Events are used to notify changes in the state of a software entity, along with some contextual information associated to those

changes. For example, an event could notify a user connection to a chat service. More precisely, an event is defined as a significant change in the state of a software at a concrete spatio-temporal location [105] [7]¹.

Event-based communications usually avoid **polling** operations, that is, continuous requests for the same piece of information, so as to check if it contains any changes. For instance, a service in a social network could notify an event whenever a user publishes a new post. That way, the other users would not have to continuously access the service to check if some new information was published.

Avoiding polling operations is very important from the point of view of the efficiency, since those operations consume a lot of energy, CPU, bandwidth, etc., while the underlying hardware supporting them may have a limited amount of resources (smartphones, sensors, embedded systems, etc.). As a consequence, EDA is increasingly being applied to the design of embedded systems.

In addition to generally improving efficiency, EDA incorporates some mechanisms to promote a low cohesion and a loose coupling between the communicating entities in a system. Particularly, in addition to SOA services, EDA incorporates the notion of **event**

¹In order to improve the legibility of this research work, the term *event* will also be used to refer to the message that notifies the occurrence of an event in an entity of a software system

emitters and receivers:

- An event emitter is an entity that notifies any changes in its own state to the rest of entities that conform the system in which they reside.
- An event receiver will capture the notifications produced by the emitters, process them and, accordingly, execute some actions.

Emitters and receivers promote a low cohesion and a loose coupling since there are not any interaction points connecting them (i.e., in SoaML, no ServicePoints or RequestPoints should connect emitters with receivers, or viceversa).

2.1.2. Communication Paradigms

The Object Management Group (OMG) committee describes as an appendix to SoaML [91] three communication paradigms to exchange information between services: **Request-Response**, **Publish/Subscribe** and **Document Centric Messaging**. The next subsections will detail those communication paradigms, their main characteristics and the scenarios in which they are mainly applied.

2.1.2.1. Request-Response (RR)

The Request-Response paradigm (RR) is the most traditional way of communicating information in a distributed system. It defines a simple way to exchange information through **message passing**: A sender requests certain information to a receiver, which replies with a message including the required information. RR is widely used in distributed systems, in particular when they are designed on the basis of SOA, since it allows to easily interoperate with services. Also, the message passing semantics of this paradigm have been applied as a **primitive** to develop more complex communication schemes [33] (like PubSub, which is described in next subsection) and to model very common communication protocols, like HTTP.

Several variations of the RR paradigm have been proposed in order to achieve different goals: one-way requests (the response is only a status message), batch requests (several requests codified as a single one in order to improve efficiency), **RPC** (requests codify a remote procedure call [14] or a method invocation [66], whereas the response is the result of its execution), etc. It is worth to mention that the RPC is the most widely used variant of the RR paradigm. In fact, several authors have even considered RPC as a separate communication paradigm [3] [117] [79], so as to highlight its importance.

Finally, at implementation level, it is very frequent to design *proxy* classes or functions whose interfaces are equivalent to those exposed by the public interface of the services. The idea is to make it **transparent** to the developer whether the communications are local (method or function calls) or not (service invocations). At high level, the developer just interoperates with a set of objects or functions through a certain interface. Internally, those calls can be translated into an invocation to a service through a set of communication technologies and adopting the RR communication scheme.

2.1.2.2. Document-Centric Messaging (DCM)

In the Document Centric Messaging (DCM) paradigm the basic interaction units are the **documents**. This way, services receive different types of documents, they process them and try, in consequence, to execute an operation, which could lead to the exchange of other documents. As specified in the SoaML standard, the DCM paradigm can be considered as a way to distribute messages to *inboxes* placed in the services. Correspondingly, it emulates how human beings exchange mails.

The DCM paradigm avoids the need to establish a well-defined public interface for the services, in contrast with the RR paradigm. Services just need to incorporate a communication protocol that allows the exchange of documents. Anyhow, the

format of the documents must be understandable by all the services exchanging them. Thus, it is very common to structure the documents using languages supporting schemas, like XML.

The communication model behind this paradigm can be either synchronous or asynchronous, depending on if it is required to immediately process and operate over a document after receiving it (synchronous model), or if the document can be stored in a processing queue and processed/operated afterwards (asynchronous).

2.1.2.3. Publish-Subscribe (PubSub)

The PubSub paradigm emulates the human procedure of subscribing to a publication: from the moment a subscriber expresses its interest in certain information, it will automatically receive a copy of the information each time it is released [5].

The PubSub paradigm is mainly used to **notify** changes in the internal state of a sender (publisher) to a set of interested receivers (subscribers). For example, in a home automation system, if a light is switched from *off* to *on*, then this occurrence could be notified to end-user applications, which could update their corresponding user interfaces. Hence, the PubSub paradigm is commonly applied when designing systems on the basis of EDA. In fact, the basic interaction unit in the PubSub paradigm is the **event**.

From a technological perspective, this paradigm is usually implemented by designing an **event broker** (usually, as a service) that stores the subscriptions and receives all the publications [68]: when a new publication is received, the event broker distributes it among the subscribers. Additionally, there are two main ways to design an event broker: either with a **pull-based** or with a **push-based** communication model.

In the pull-based communication model, the entities are in charge of detecting the occurrence of new events. Therefore, in order to receive them, the entities must periodically **poll** the event-broker, which will have to store the published events while any subscribers remain active. Besides, the broker will have to poll the emitters in order to check if they need to publish a new event. Figure 2.2 illustrates the operation mode of the pull-based communication model.

Likewise, Figure 2.3 illustrates the operation mode of the push-based communication model, in which the emitters **directly** transmit the new events to the event broker. Meanwhile, the event broker is in charge of transferring the received events to the subscribers, without requiring them to perform a previous request.

In most cases, event notifications are unpredictable and very separated in time, which makes the push-based communication

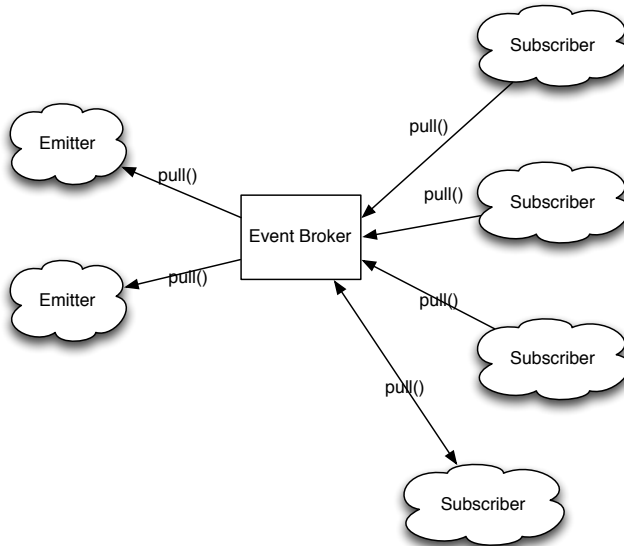


Figure 2.2: Pull-based communication model

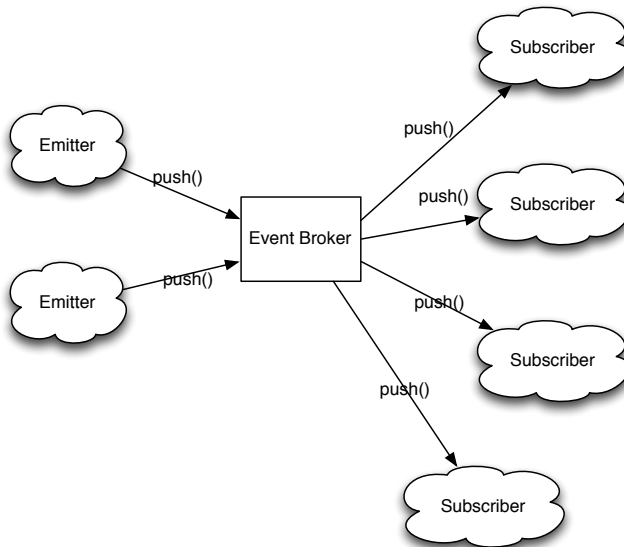


Figure 2.3: Push-based communication model

model more advantageous, as it generally consumes less resources (i.e., it does not require an intermediate storage, polling operations are avoided, bandwidth use is decreased, and so forth). However, when the number of events to be notified is very high or they occur at very regular intervals, then using the pull-based communication model could have some benefits over using the push-based one, since a single polling operation could be used to trigger several event transfers at the same time, and the time between polling operations could be adjusted to fit the intervals in which the events are notified.

In spite of the benefits that the push-based communication model may offer, it is **not possible to implement it** within some technical scopes, since it is required to keep long-term connections alive and to be able to transfer information without a previous request from the receivers. For instance, note that current HTML and JavaScript standards do not provide the required techniques to implement push-based communications. Nonetheless, the new **HTML5** standard [10], to be published in the next few years by the W3C committee, incorporates **WebSockets** and **Server-Side Events**, which allow to develop push-based communication models.

2.1.3. Some Notations for Representing Distributed Systems

Several notations have been proposed to represent the interactions between the different elements of a system. In consequence, these notations can be used to help into designing a distributed system. The following subsections provide a brief description of some of the most relevant ones.

2.1.3.1. Petri Nets

A Petri net is a formal mathematical modeling language that has been widely used to analyze, design and validate distributed systems [123]. A sample Petri net is represented in Figure 2.4.

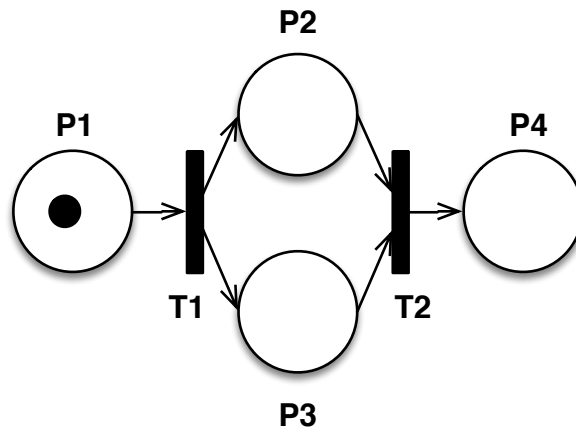


Figure 2.4: An example Petri net

A petri net represents a set of places and transitions in a di-

rected graph. A transition represents an event and a place represents a condition that is satisfied. Places are linked to transitions by arcs, which represent that a satisfaction of a condition produces an event, or an event produces the satisfaction of a condition. Since the execution of a Petri net is usually nondeterministic, if several transitions are activated at the same time, then one of them is randomly triggered. To activate a transition, an appropriate amount of *tokens* must be present in the input places. A token is graphically represented through a black point contained in some places. When a transition is executed, it consumes tokens from the input places and produce the same amount of tokens in the output place.

2.1.3.2. UML 2.x Communication Diagram

UML 2.x Communication Diagrams (formerly called collaboration diagrams in UML 1.x) are the standard OMG approach to graphically represent the interactions between the parts of a system in terms of a sequence of ordered messages [39]. As so, they emphasize the interactive relationships between the elements present in a system. A sample UML communication diagram is depicted in Figure 2.5.

In a communication diagram, an actor interacts with a set of objects that, in turn, collaborate between them through a set of exchanged messages. The sequence of messages is represented

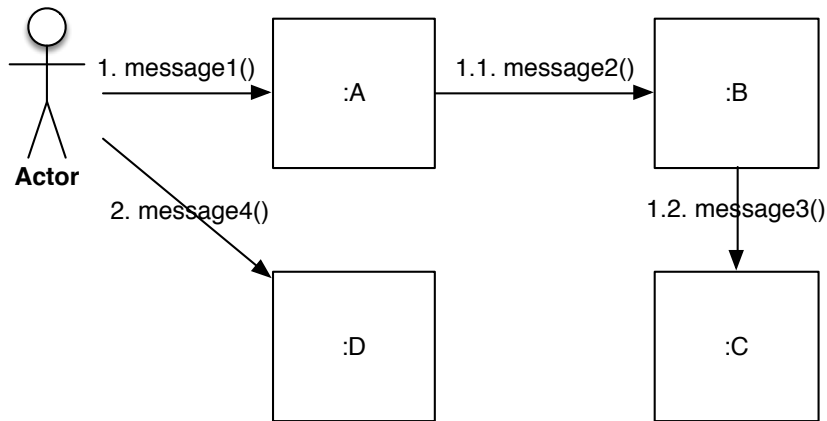


Figure 2.5: An example of a UML 2.x communication diagram

through a numbering scheme in these diagrams.

2.1.3.3. UML 2.x Activity Diagram

UML 2.x Activity Diagrams are the standard OMG approach to graphically represent the workflow of activities that are carried out in a system [39]. In UML 2.x, activity diagrams are based on Petri nets [116]. Consequently, they can precisely depict the behavior of a set of interacting processes. An example of a UML activity diagram is depicted in Figure 2.6.

In an activity diagram, activities are conformed by a set of actions linked through a control flow. Each activity is started by an initial node, and ended by one or more final nodes. Actions can be concurrently executed, and some conditions can prevent or allow

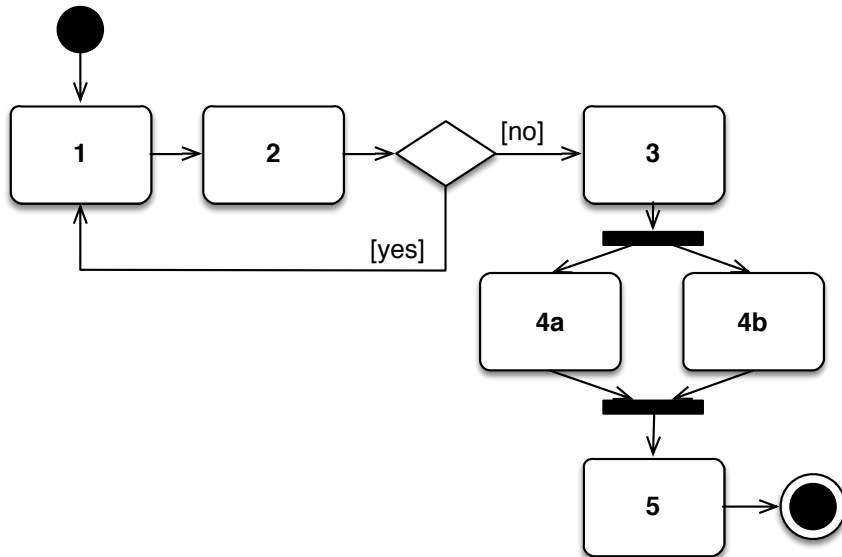


Figure 2.6: An example of a UML 2.x activity diagram

their execution.

2.1.3.4. Business Process Modeling and Notation (BPMN)

Business Process Modeling and Notation 2.0 (BPMN 2.0) is the OMG standard for modeling business processes using a graphical notation [89]. BPMN can be used to represent the interactions between a set of processes that collaborate to achieve certain goals. The focus is to provide a notation that can be easily understood by stakeholders, and that it is as much separated from software aspects as possible. It is very similar to UML 2.x Activity Diagrams, but it intends to provide a more understandable, informal and simple no-

tation than UML. Therefore, it can be adequate to depict a highly abstract diagram representing the overall interactions that the different processes that conform a distributed system may carry out.

BPMN 2.0 also includes a notation for representing choreographies, that is, an ordered set of interactions between the participants of a business process. As a consequence, it is a suitable notation to represent the communication aspects of a system from a highly abstract perspective. An example of a BPMN choreography diagram is depicted in Figure 2.7.

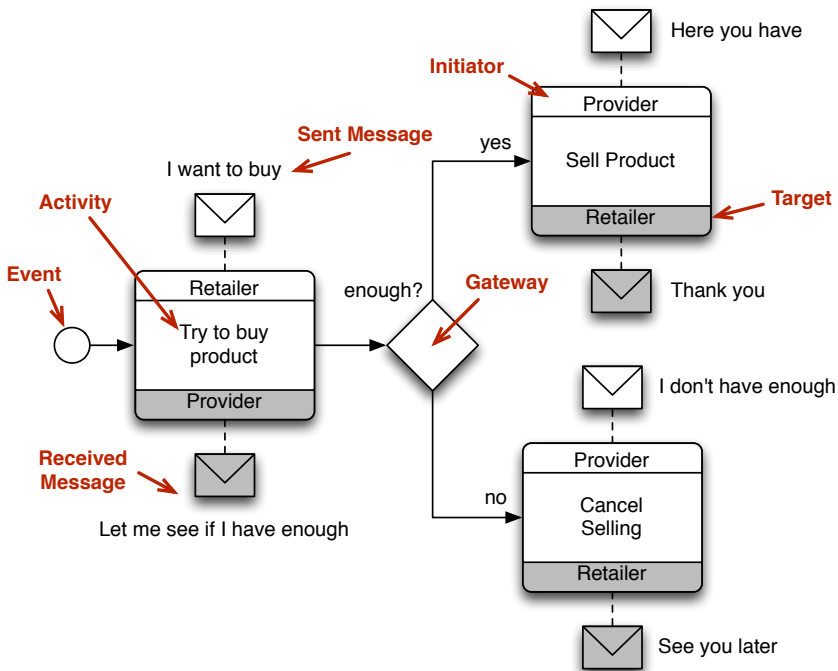


Figure 2.7: An example of a BPMN choreography diagram and a summary of its elements

Previous example represents different interactions between a retailer and a provider to buy a product. The notation includes elements that are widely used in BPMN choreography diagrams: activities, participants, messages, events and gateways. An activity is an interaction between two participants and may have sub-activities with different interactions between the participants. A message is an information unit transferred between participants, and it can be communicated in a non-specific mode (i.e., synchronous or asynchronous). A gateway represents a branch or a merge between different activities. Finally, an event is an exceptional occurrence between two activities and/or gateways.

2.1.4. Supporting Communications: Middleware

This subsection exposes some research work related to the design and management of communication schemes through middleware technologies.

A middleware is defined as a software layer that is located between the operating system and the end-user applications, hiding the heterogeneity of different physical computer architectures, operating systems and programming languages, hence, simplifying the process of transferring information between the different machines that are part of a distributed system [12].

Several of the most remarkable (and traditional) middleware technologies for distributed systems are described in the following subsection. Moreover, the relationships between design patterns, software frameworks and middleware are explored, so as to emphasize the key role of a middleware in the design (not only the implementation) of complex distributed systems. Additionally, it will make it easier to the reader to differentiate these concepts, which are usually mixed-up.

2.1.4.1. Traditional Middleware Technologies

Several middleware technologies have been proposed to support the development of distributed systems. The following list highlights the most remarkable ones and their main characteristics:

- **CORBA** (Common Object Request Broker Architecture [87]) was specified by the OMG committee to support an object-oriented approach to RPC. It is widely considered as the main reference in the field of the communication middleware technologies. There are three main categories in the CORBA specification: (1) a language to define public interfaces for services, which is known as Interface Description Language (IDL); (2) a specification in IDL of the basic CORBA services [87]; and (3) a specification of the CORBA communication protocol, which is called General Inter-ORB Protocol (GIOP)

[88]. In CORBA, it is remarkable how IDL makes the specification independent from the implementation in any specific platform (programming language, operating system, etc.). In fact, there are many implementations of CORBA, like OmniORB (<http://omniorb.sourceforge.net>), TAO (<http://www.cs.wustl.edu/~schmidt/TAO.html>) or Mico (<http://www.mico.org>).

- **DDS** (Data Distribution Service [86]) is another middleware specified by the OMG committee. Its focus is to support the PubSub paradigm in embedded and real-time systems. Additionally, it supports an extensive set of parameters to tune the quality of service (QoS), like reliability, bandwidth, etc. DDS structures the information as a set of key-value pairs associated to a *topic*. The entities can subscribe to a certain topic to receive the information associated to it whenever is published. The communication protocol is known as Real-Time Publish Subscribe (RTPS). Like CORBA, the public interfaces of DDS are defined through IDL, which makes them independent from any specific platform.
- **ICE** (Internet Communications Engine [130]) is the natural successor to CORBA. In contrast with CORBA, ICE is both a specification and an implementation of that specification for multiple platforms. It was defined and implemented by several

original members of OMG committee that defined CORBA, as a response to its slow evolution. A new protocol, called IceP, replaces GIOP, since it was considered to be inefficient. Most of the CORBA services have a corresponding ICE one. The IDL has been extended in some ways and simplified in others, removing some deprecated constructions. Additionally, the specification contains a mapping between the IDL and some of the most relevant programming languages nowadays: C++, Java, PHP, Python, Ruby, C#, Objective-C, and so on.

- **RMI** (Remote Method Invocation, <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html>) is the default approach to object-oriented RPC that is available in Java. It is simple to use, but its capabilities are limited in comparison with CORBA, DDS or ICE. For example, RMI can only be (officially) used in software written in Java, thus limiting its interoperability possibilities. To workaroud that issue, some mapping technologies have been implemented in order to be able to transform RMI messages to other protocols. An example of that is RMI-IIOP (<http://docs.oracle.com/javase/7/docs/technotes/guides/rmi-iiop>), which allows the interaction between RMI-based and

CORBA-based software systems. In spite of the functional and technical limitations of this middleware, it is widely used in practice, specially due to its simplicity in comparison with CORBA, DDS or ICE and to the success of the Java platform.

- **SOAP** (Simple Object Access Protocol [47]) is a protocol to exchange structured information in distributed systems. It was defined by the W3C consortium. Albeit its low efficiency, as a consequence of the XML-based textual notation that is used to structure the messages, SOAP is a technology of great interest, since it works on top of standard HTTP requests. Therefore, it allows to implement the RR paradigm in the Web. Likewise, the textual notation of the messages using XML allows to represent structured documents, thus supporting the DCM paradigm. Technically, it can be used in any platform supporting XML and HTTP and is very easy to be debugged (i.e., the messages are formatted in XML, which makes them very legible).
- **WCF** (Windows Communication Foundation, <http://msdn.microsoft.com/en-US/library/dd456779.aspx>) is a set of APIs and a programming model to produce services from classes developed by using the Microsoft .NET framework. It is the successor of DCOM (<http://www.microsoft.com/com/default.msp>) and

.NET Remoting ([http://msdn.microsoft.com/en-US/library/kwdt6w2k\(v=vs.100\).aspx](http://msdn.microsoft.com/en-US/library/kwdt6w2k(v=vs.100).aspx)), which were other middleware technologies developed by Microsoft for the Windows operating system. WCF is similar to RMI, but has more functionalities. Moreover, the messages can be formatted in either XML or JSON, which increases its compatibility with other middleware technologies, the Web and with any software platform supporting XML and HTTP.

Although these middleware technologies are widely used and provide important functionalities to promote several quality properties (e.g., efficiency, scalability, compatibility, etc.), they can not fulfill all the specific technical requirements that the more modern systems usually require (e.g., mobility in ubiquitous systems, volatility of the connections, etc.). In consequence, some newer technologies have been proposed, as it will be described in Subsubsection 2.2.3 in relation to the support of the communications in ubiquitous systems.

2.1.4.2. Relationships between Software Frameworks, Patterns and Middleware²

Middleware-based technologies are mainly accepted to overcome interoperability issues, also providing portability between different underlying platforms and sometimes satisfying other certain quality properties (efficiency, scalability, security, etc.).

Software frameworks [98] are reusable abstractions of code wrapped in a well-defined Application Programming Interface (API). They usually turn out to be a common choice in software engineering since they are focused on facilitating the development of software systems and on promoting reusability. Software frameworks comprise a set of hot and frozen spots. Hot spots represent the abstractions that are provided in order to adapt the functionalities of the framework to the specific requirements of a particular system [96], while frozen spots define basic components (and the relationships between them) that remain unchanged in any instantiation of the framework [95]. A framework differs from other approaches intended to support software development (e.g., libraries, toolkits, etc.) in that it provides “inversion of control”, that is, the framework is responsible of executing the instantiations of the hot spots when required.

²In this subsection, the relationship between patterns, frameworks and middleware is extracted from [110].

Patterns allow to solve several well-studied design issues through the use of some predefined solutions that have been shared by experienced developers and architects [40]. In fact, patterns were originated by Christopher Alexander as an architectural concept [1]. Design patterns were mainly popularized in the software engineering field by the book *Design Patterns: Elements of Reusable Object-Oriented Software* [41]. In that book, patterns are classified as follows:

- **Creational:** They simplify the process of instantiating objects.
- **Structural:** To define ways of composing objects to enhance the functionality and quality properties of a software.
- **Behavioral:** To establish how to transfer information between different objects.

As can be noticed, middleware, frameworks and patterns are oriented towards reusing knowledge across software design and implementation, so as to reduce development costs and produce optimal solutions to common problems. Since they share a common focus, they are usually interrelated: Middleware is designed using several patterns and incorporate frameworks to, respectively, facilitate software design and implementation. Therefore, from an archi-

tectural point of view, a middleware is a pattern-driven combination of frameworks.

Finally, it is worth to be pointed out that using a middleware should not be considered as not a technical-only task, since it involves incorporating several frameworks and design patterns to a software, which, in turn, influences both its design and implementation.

2.2. Ubiquitous Systems

A ubiquitous system is intrinsically a distributed system, since they are formed by a set of non-located elements that exchange information. Consequently, the notions presented in previous section can also be applied to the ubiquitous systems. However, from a technical point of view, ubiquitous systems exhibit two key differences with traditional distributed systems [82]:

- **They are volatile:** The interoperation is spontaneous and the associations between devices are constantly created and destroyed. Moreover, communication links usually fail and the bandwidth and latency are continuously changing.
- **They have a different device model:** Personal computers are replaced by sensors/actuators, mobile devices and “social”

devices (interactive walls, furniture, etc.). Consequently, resources might be constrained (energy, memory, computing capabilities, etc.) in a way that requires to develop software in a different manner than in personal computers or in traditional distributed systems.

These characteristics make it possible to establish a relationship between ubiquitous computing and other paradigms, remarkably to:

- **Mobile computing.** It is focused on enabling users the possibility to carry their personal computers and establish wireless connections with other devices.
- **Wearable computing.** It involves the miniaturization of personal computers, so that they can be incorporated into clothing, or even into the body.
- **Context-aware computing.** It refers to the possibility of automatically adapting software operation to the user context (location, nearby persons, available resources, etc.).

Whereas mobile and wearable computing is more related to hardware research (i.e., to develop small electronic devices with computing capabilities), context-aware computing is more associ-

ated to software research, even if some physical devices may support context detection and adaptation (e.g., sensors and actuators). The next subsection details context-aware computing, since it is the ubiquitous-computing-associated field that is more relevant for this thesis work.

2.2.1. Context Awareness

In the field of the ubiquitous computing, context-aware computing refers to the idea of adapting software operation to the user context, that is, the location, nearby persons, available computing resources and the changes to those elements along time [107].

In context-aware computing, software should be designed to support the following functionalities [94]:

- **Reception of information about the context:** To detect the user context through a set of sensors and to present that information to the user.
- **Contextual discovery of resources:** To automatically discover relevant resources and to make use of them in an adequate way.
- **Contextual adaptation:** To automatically execute a task, or to modify its default operation, on the basis of the detected

context and a set of predefined rules.

- **Augmented context:** To associate digital information to the physical context, in order to provide useful messages or reports to the user.

Each of these activities involve solving very difficult challenges, like supporting interoperability and mobility, that are currently being studied by the scientific community. Furthermore, some of these problems are also present in ubiquitous systems. Consequently, ubiquitous and context-aware computing are widely considered interconnected disciplines. In fact, from a technical perspective, ubiquitous and context-aware systems need analogous communication mechanisms. For that reason, the proposals presented in this thesis can be considered to be applicable to both types of systems. Moreover, a ubiquitous system should contain context awareness capabilities to support a transparent adaptation to different real environments [109], which further interconnects both research fields.

2.2.1.1. The Notion of Context and its Modelling

One of the main issues that may arise when designing a context-aware system is to correctly model the context itself. Even if in previous subsection it is given a definition of context that is

widely accepted (i.e., a conjunction of the location, nearby persons, available resources and the changes to those elements along time [107]), it can be considered that the exact conceptualization depends on the domain or the specific problems to be solved. For instance, some authors mention that the context is more related to the environment of the applications [126], whereas other authors relate the context to the user environment [20] or to the user behavior and feelings [30].

Due to the different conceptualizations of the context, it is usual to model domain-specific ontologies to represent it. From a computing perspective, an ontology is defined as a formal specification of a conceptualization [45]. It describes the concepts associated to a domain, their relationships, properties and constraints. OWL language [124] is commonly used to specify ontologies. It is based on RDF/XML [8] and its formal semantics are founded on descriptive logic. OWL allows to represent semantic classes, properties, individuals and values. Moreover, it is possible to execute deduction operations over the represented concepts by using *reasoners*, like Pellet (<http://clarkparsia.com/pellet/>). This is of great importance in context-aware computing, since it allows to detect the context on the basis of some incoming information (from sensors, applications, services, and so forth) and to adapt a system operation on the basis of a set of logic rules.

Even if domain-specific ontologies to represent the context are common, there have been several efforts to propose domain-independent or *generic* ontologies to specify it. One prominent example is the *Standard Ontology for Ubiquitous and Pervasive Applications* (SOUPA) [25], which includes two different sets of ontologies: SOUPA Core and SOUPA Extension. SOUPA Core incorporates elements that are present in any ubiquitous and/or context-aware system: *Person*, *Politic-Action*, *BDI-Agent* (Beliefs, Desires and Intentions), *Time*, *Space* and *Event*. SOUPA Extension includes elements to extend SOUPA and to support very particular, but widespread, applications: *Meeting*, *Agenda*, *Document*, *Screenshot*, *Connected Region* (i.e., to relate different physical spaces) and *Location*.

Although it is possible to use SOUPA to represent most of the notions of context, some particular scenarios still require specific ontologies or, at least, to substantially extend SOUPA. One example is COBRA-ONT [24], which has been proposed by the same authors of SOUPA to support *smart rooms*.

2.2.2. Communication Paradigms in Ubiquitous Systems

In ubiquitous systems, it is very common to establish communication schemes based on either the PubSub or RR paradigms.

Each communication paradigm provides orthogonal functionalities and promotes different quality properties, while most existing ubiquitous systems actually need to fulfill a combination of the functional and non-functional requirements fostered by each paradigm. For example, in a home automation environment, it is usually required to directly interact with specific devices through well-known interfaces or through message passing, thus being appropriate to choose RR-based communications. On the other hand, when a device changes its state (a door is opened, for instance), the applications should be notified, so as to update their GUI. In this case, the use of PubSub-based communications is more suitable.

Table 2.1 outlines the contribution of each communication paradigm to the quality properties that are very often sought for ubiquitous systems [28] [106] [3] [53]. It is important to note that the quality properties that are mentioned in this section can be achieved with the appropriate implementations of either RR or PubSub mechanisms. The problem is the impact that they will have in other requirements and the high level of complexity needed to fulfill them. These problems will negatively affect the performance of the systems that are built on top of them. For example, a PubSub proxy could ensure reliable delivery, however, by using a proxy, all the communication will need to be centralized in it. This choice would avoid using decentralized implementations of the PubSub

paradigm and would require to apply replication techniques in order to avoid bottlenecks. However, by using the RR paradigm, reliable delivery requirements are directly met.

Property	PubSub	RR
Efficiency	partial	partial
Mobility Support	✓	
Adaptability	✓	
Reliable Delivery		✓
Security	partial	✓
Timeliness		✓

Table 2.1: Quality properties promoted by the PubSub and the RR paradigms

A more detailed explanation of the information included in the previous table is described in the following subsections. This analysis could motivate the proposal of model that integrates the PubSub and the RR paradigms, which may contribute to seamlessly take advantage of the semantics of both paradigms and the quality properties that each of them helps to promote.

2.2.2.1. Efficiency

PubSub paradigm is, in general, more efficient for distributing the state of the entities and for delivering a message to several receivers. To do those tasks, the RR paradigm semantics require to periodically execute polling operations, which are usually considered very inefficient in comparison with the scheme supported by

the PubSub paradigm [97], since such changes infrequently occur and a lot of resources are wasted when sending useless messages (memory, CPU, energy, bandwidth, etc.). Moreover, in RR, to distribute information to a set of receivers, the number of messages to be sent must be equal to the number of receivers. In PubSub, publishers always distribute one message, regardless of the number of subscribers. Anyhow, the RR paradigm can be more convenient if the notifications always occur at very regular intervals or if power consumption must be periodically controlled. As a consequence, both the PubSub and RR paradigms may help to achieve efficiency in ubiquitous systems. The choice between the two paradigms depends on the specific constraints of each system.

2.2.2.2. Mobility Support

The PubSub paradigm promotes the decoupling between publishers and subscribers. In particular, in PubSub-based communications, it is totally transparent if either a publisher or a subscriber is present or not in a system. In RR, if a receiver is no longer available in a system, due to the coupling between senders and receivers, the execution flow of a sender could be indefinitely blocked waiting for a response that could never be received since the provider could never be present again. Additionally, the execution flow of a sender usually depends on the specific results that are extracted from

the responses of the receivers. Thus, in some cases, senders may not be able to continue their execution if specific recipients are not available. Therefore, the PubSub paradigm contributes to support mobility in ubiquitous systems, whereas the RR paradigm offers no mechanisms to support it [106].

2.2.2.3. Adaptability

RR-based communications require establishing well-defined interfaces to exchange messages between senders and receivers. However, in ubiquitous systems, the support to context-awareness features involves to dynamically adapt the functionality provided by services and applications to the information retrieved from the context (that is, nearby users, their tasks, available resources, etc.) [129]. Consequently, RR communications are not flexible enough to promote adaptability [28]. However, in PubSub communications, subscriptions may be dynamically established and dropped depending on the context. Thus, the PubSub paradigm is more suitable for building adaptable, ubiquitous systems.

2.2.2.4. Reliable Delivery

Reliable delivery means that a receiver (or a set of receivers) has to send an acknowledgement for each received message in order to confirm their reception. In RR, receiving a response to a re-

quest implies that the request was delivered correctly. However, in PubSub communications, reliable delivery implies detecting from a publisher (i.e., not only from the **event broker**, that is, the intermediary entity between publishers and subscribers) whether a set of subscribers have received a specific notification or not. This is only possible by increasingly reducing the decoupling between publishers and subscribers [28]. For example, in order to provide reliable delivery in the PubSub paradigm, the publishers should know, at least, the number of subscribers and an identification associated with each subscriber. Consequently, the publishers should receive an acknowledgement message from each subscriber. As a consequence, it is not possible to assume that a notification is always received when the decoupling between publishers and subscribers is a strong requirement. Thus, when reliable delivery must be ensured, RR should be used instead.

2.2.2.5. Security

Security is an important concern in ubiquitous systems. Hence, the information to be exchanged should be encrypted and trusting mechanisms established for senders and receivers. Obviously, information can be encrypted in both the PubSub and the RR paradigms. However, trusting mechanisms such as digital signatures or certificates are easy to establish only in RR-based

communications. In the PubSub paradigm, it is difficult to detect the source or the recipient of a notification, due to the decoupling between publishers and subscribers. Moreover, event brokers enable trusting mechanisms between publishers and brokers or between brokers and subscribers, but never directly between publishers and subscribers. Thus, a publisher is not able to detect if the recipients of a notification can be trusted, while subscribers are not able to detect if a notification has been sent from a trusted source. Overcoming this weakness involves considering additional complex trusting mechanisms that decrease efficiency [35].

2.2.2.6. Timeliness

Real-time applications require controlling the timeliness of delivered messages. In PubSub-based communications it is not even possible to establish if a notification will ever be received, (see *Reliable Delivery*), thus making it impossible to delimit the time of a notification delivery from the point of view of a publisher (i.e., event brokers can be implemented to guarantee timeliness). Additionally, if there is more than one subscriber, then the delivery time and order will depend on the specific implementation of the event broker, which could vary delivery times even between consecutive notifications received by the same subscribers. Hence, timeliness cannot be enforced for publishers in PubSub-based communications [28]. In

this way, the RR paradigm would be required.

2.2.3. Middleware Technologies for Ubiquitous Systems

Ubiquitous systems should meet some specific requirements that traditional middleware technologies do not currently accomplish [69], like:

- They do not support ad-hoc networks, which are of great importance in ubiquitous systems (see Subsection 2.2.4). The reason is that traditional middleware technologies are intended to work in infrastructure networks (e.g., LANs, Internet, etc.). Consequently, ad-hoc transmission interfaces (like Bluetooth or infrared) are neither supported by design nor supported in the existing implementation of the middleware. An example of the last case is CORBA, which could support any transmission interface, but the existing implementations do not support ad-hoc interfaces.
- Mobility support is not provided, or it is utterly limited. Therefore, it is not possible to correctly deal with the dynamic changes in the networking connectivity that ubiquitous systems commonly suffer.
- Traditional middleware technologies foster the use of a spe-

cific communication paradigm, allowing partial or none use of others. However, a combination of paradigms is usually required in ubiquitous systems, so as to meet the quality properties that are usually expected to be accomplished (see Subsection 2.2.2).

Due to the previous limitations, several authors have proposed middleware technologies to specifically support communications in ubiquitous systems. Some of the most remarkable ones are summarized in Table 2.2, whose contents have been extracted from [48], simplifying its structure and contents.

These middleware technologies deal with communications in ubiquitous systems in a different manner (i.e., different communication paradigms, underlying networking technologies or protocols, etc.), focusing on some requirements but not taking into account others. Moreover, the employ of textual protocols, commonly based on XML or JSON, has recently overtaken the use of middleware technologies, since they can pose as middleware and they are easier to use and understand. However, their efficiency and scalability (among other properties) is lower in comparison with most previously mentioned middleware technologies.

Consequently, software engineers usually have to integrate diverse communication technologies into a shared communication

Name	Summarized Description
PubSub and DCM Middleware	
STEAM [73]	PubSub-based middleware that manages groups of nearby users and provides partial support for power management, mobility and interoperability.
EMMA [81]	Message-oriented middleware supporting one-to-one and one-to-many communications. Messages are full pre-formatted documents.
P2P Middleware	
Expeerience [15]	It supports mobile code, to discover shared resources and P2P information exchanging.
Middleware based on mobile agents and components	
SELMA [43]	It ensures a balanced resource management between producers and consumers. It supports dynamic discovery and mobile agents.
Mobile-Gaia [114]	PubSub-based middleware with coordination support and management of clusters of entities. It provides a <i>WYNIWYG</i> (What You Need Is What You Get) platform.
Middleware based on tuple spaces	
LIME [80]	A LINDA-based tuple space [22][42] to share resources.
MeshMdl [52]	Object-oriented tuple space. It supports mobile agents and makes use of an asynchronous communication model (<i>Xector</i>).
Middleware based on shared resources	
XMIDDLE [72]	Shared information is structured in XML. It is able to manage network disconnections, which are usual in ubiquitous systems.
Middleware based on virtual machines	
Mate [65]	It only supports TinyOS (http://www.tinyos.net), which is an OS and an interpreter for embedded systems. It provides synchronous communications and mobility support.

Table 2.2: Some of the most remarkable middleware technologies for ubiquitous systems

component [44], so as to provide a holistic support to the different communication requirements that are expected to be fulfilled. For instance, DDS (Data Distribution Service) service specification [86] is commonly used to support real-time event distribution, and UPnP (<http://www.upnp.org>) or Apache River (Jini) (<http://river.apache.org>) are software frameworks to support dynamic discovery of nearby entities. This combination of different communication technologies, usually results on the decrease of maintainability and reusability of the resulting software solutions [118].

2.2.4. Communications in Ubiquitous Systems: Technical Issues

There are several important technical issues that need to be taken into account when developing a ubiquitous system, specially when dealing with communications. For instance, traditional networking infrastructures (LANs, Internet, etc.), which combine routers and network nodes (i.e., personal computers and other computing devices with networking capabilities, like smartphones, tablets, etc.), can not correctly deal with the dynamicity and mobility requirements of the ubiquitous systems. As an example, in rural areas in which those networking infrastructures are not present, the network nodes can not exchange information between

them, even if they are physically close to each other.

As a result, Mobile Ad-hoc NETWORKS (MANETs) are increasingly being adopted to support information exchange in ubiquitous systems. These networks do not have a static infrastructure, but they are automatically reconfigured by themselves to self-adapt to the available nodes at any specific moment. In order to achieve that goal, these networks use ad-hoc connection standards, like IEEE 802.11s [21] or BlueTooth (<http://www.bluetooth.com/>). Certain network nodes are automatically chosen to behave as routers in order to scatter information to other nodes. Figure 2.8 illustrates a MANET in which all the nodes can transfer information between them. To do so, Node B must act as a router, since the Node A is not able to directly reach the Node C.

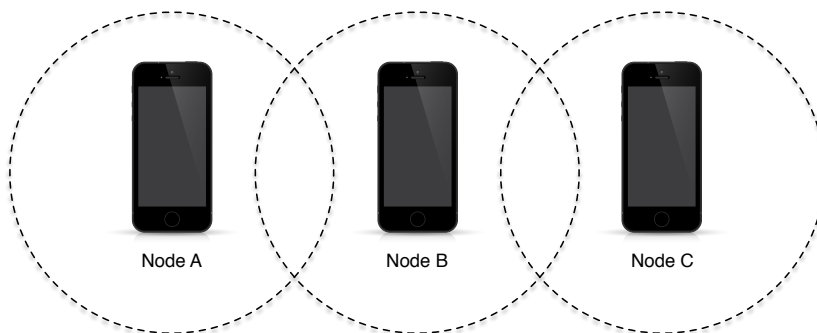


Figure 2.8: An example of a MANET with three devices: There is a total connection between them since the Node B can route the data transmissions between A and C

Moreover, broadcasting a message in a MANET involves us-

ing different information dissemination techniques, like:

- *Clustering* [51] [31]. A node is in charge of propagating messages to the devices surrounding its physical location. The selection of the propagating node is based on the use of different metrics, like battery life, the distances between nodes, etc.
- *Simple flooding* [56] [61]. Each node disseminates all the received broadcast messages to its nearby nodes.
- *Probabilistic, area-based and neighbour-knowledge broadcasts* [122]. These methods are similar to simple flooding, but each node only propagates each received broadcast message to a subset of its nearby nodes, so as to decrease redundancy and network traffic. The subset of nodes is dynamically chosen whenever a node receives a broadcast message, on the basis of different techniques (e.g., the distances between nodes, the topology of the network, randomly, etc.).

Broadcasting is not only important to disseminate messages, but also to dynamically discover the devices that belong to a MANET, which can be a complex task due to the coincidence of several MANETs in the same geographic area. To avoid this issue, it is common to make a “virtual” association between devices and MANETs [122].

Additionally, MANETs may have one or more nodes connected to the Internet, so as to access remote services. These MANETs are known as Internet-Based Mobile Ad-hoc Network or, simply, as iMANETs. iMANETs allow “off-line” nodes to access remote services through the “on-line” ones. Consequently, iMANETs combine the dynamic infrastructure of an ad-hoc network with the universal information access provided by Internet. Nonetheless, certain technical issues have limited the success of these networks. For instance, it is still necessary to figure out suitable strategies to maintain a cache of information [67] and to decrease the use of resources (battery, CPU, memory, etc.) of the nodes that are connected to the Internet.

Ubiquitous systems are also challenging in terms of privacy and security, since, in fact, most of the exchanged information is personal and confidential. For example, in MANETs, the information could be easily captured and stored by the intermediate devices that behave like *routers*. Consequently, designing ubiquitous systems involves taking into account the following security aspects [69]:

- *Authentication*. An entity (i.e., a physical device or software) can not adopt the identity of another.
- *Authorization*. The access to shared resources should be controlled by permissions.

- *Non-repudiation.* An entity can not reject a valid message, and if a valid message is received, then it is not possible to negate its reception. $\text{if } i \text{ then } j$

Finally, physical networks supporting communications in ubiquitous systems usually deal with a lot of traffic and have a limited bandwidth. Therefore, it could be important to take into account certain quality of service (QoS) requirements at different levels. For instance, the QoS could be adapted depending on the user role or the functionality provided by the ubiquitous system could be dynamically adapted to the level of QoS at a given moment.

2.3. Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) is a software development approach that is focused on producing and using models to reduce platform complexity [17]. It is also a “promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively” [111]. In general, MDE improves the development process by separating concerns and allowing the systematic automation of production, integration and validation processes [119].

MDE is also considered as a superset of a development methodology known as Model-Driven Development (MDD). This development paradigm uses models (and their transformations) as the main artifacts of the development process, even generating code from them [74]. MDE, on the other hand, is not only focused on the development tasks, but also on the complete engineering process (evolution, reverse engineering, validation, testing, simulation, analysis of costs, etc.) [112]. So as to clarify these commonly mixed terms, the relationship between MDE, MDD and several other methodologies yet to be mentioned in this section has been illustrated in Figure 2.9.

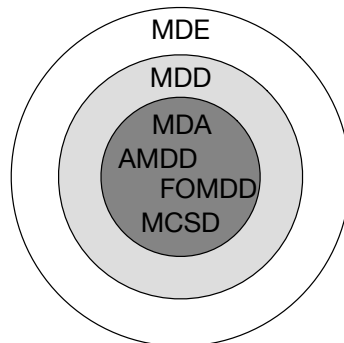


Figure 2.9: Relationship between MDE, MDD and other methodologies

MDE is based on two primary mechanisms:

- *Abstraction:* In MDE, it consists of the definition of a set of domain models representing the specification of a software. A domain model is a conceptual model (i.e., a conjunction

of entities and relationships between them) that encompasses all the topics related to a specific problem [38]. Moreover, a domain model defines the scope of the problem domain and serves as a shared vocabulary between different stakeholders [55]. Domain models are usually specified in Domain-Specific Modeling Languages (DSMLs) [111].

- *Refinement or generation*: It is carried out by transforming domain models into other models, so as to obtain different perspectives of the problem to be solved, or the solution to be developed. The OMG defines a transformation as *the process of converting one model to another one of the same system* [85]. Transformations are specified through a set of rules applied to the domain models.

MDE methodology is depicted in Figure 2.10. Several domain models (obtained through **abstraction** mechanisms) and transformation rules serve as an input to a set of **refinement** mechanisms, which produce as an output other domain models that can be re-used (with another set of rules) to produce, again, more domain models.

There are different variations of the MDE (or MDD) approach, depending on the degree of use of the above-mentioned mechanisms: some of them trend to produce many abstractions to solve a problem, others are more focused on refinement and the rest try

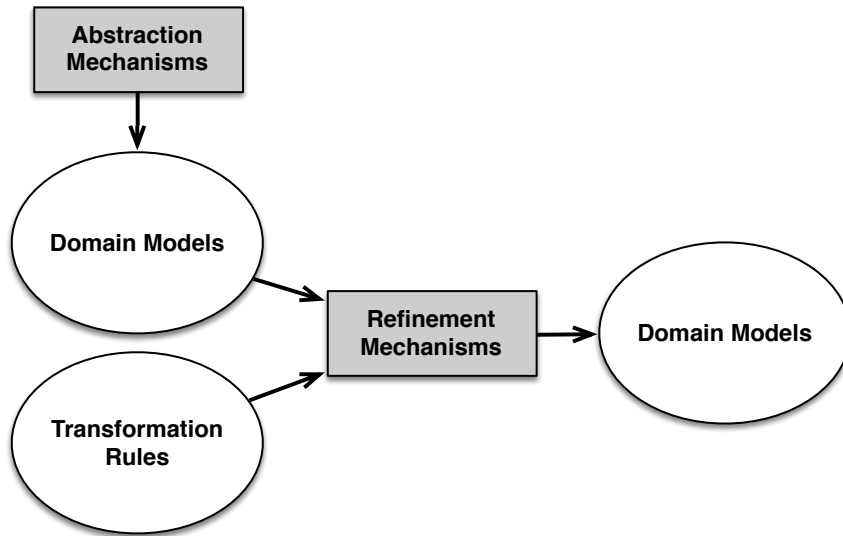


Figure 2.10: Graphical illustration of the MDE development methodology

to balance the efforts between producing abstractions and refining them. Some of the most important variations are [111] [60]:

- **Model Driven Architecture (MDA) [85].** It is the OMG standard approach to MDE. Models are divided into three main abstraction levels: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). Each model must conform to a metamodel, and a set of transformation rules, which are applied to the corresponding metamodels, are intended to automatically derive a PSM from a CIM. The main objective of MDA is to separate a software design from the technical details of an implementation.

- **Agile Model Driven Development (AMDD) [2].** Implementation efforts are guided by “good enough” agile models that should be as simple as possible, easy to understand and sufficiently accurate, consistent and detailed. The main idea is to use models during development, but to decrease the efforts of defining or using them as much as possible, and to keep them simple enough for stakeholders.
- **Feature Oriented Model Driven Development (FOMDD) [121].** A model is refined by composing or deriving *features* from other (existing or new) models. A feature is “a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained” [59]. FOMDD tries to improve the reusability and maintainability of a software by mapping the representations of features across all the phases of the software life cycle: analysis, design, implementation and testing.
- **Model Centric Software Development (MCSD) [125].** Models are central to all phases of the development process. All the aspects of the software are modeled through Domain-Specific Modeling Languages (DSMLs) to represent *aspects of interest*. The models are mapped to the corresponding elements of the implementation. Consequently, it is possible to automatize most of the code generation, so as to produce

nearly-complete implementations of the components or artifacts involved in a software development. Reverse engineering is considered as a method to obtain models (from existing code). Finally, due to the well-defined link between models and implementation artifacts, model verification and checking can be achieved through rapid-prototype generation and run-time performance analysis.

Even if previous variations of MDE are of great significance, this thesis work will focus on MDA. The reason is that this methodology tries to clearly separate abstractions from technical issues. Thus, it can be suitable in order to deal with the communication aspects of ubiquitous systems at design level, without taking into account the technical issues that encompass the use of networking technologies, protocols, middleware, etc. Moreover, it is currently the only approach to MDD that has been defined and evaluated by an standard's committee (Object Management Group, OMG), thus ensuring its quality, well-defined specification and interoperability with other standards (like UML).

2.3.1. Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) is an Object Management Group (OMG) **standard** approach for the development of software

systems through MDE [85].

In MDA, software is developed on the basis of a forward engineering process, that is, by producing code from abstract models. The general idea is to decouple software design from the technical aspects of its implementation [85].

Reciprocally to MDE, MDA is based on two main mechanisms: abstraction through **modeling** and refinement through **transformations**. The MDA standard development methodology can be represented as depicted in Figure 2.11.

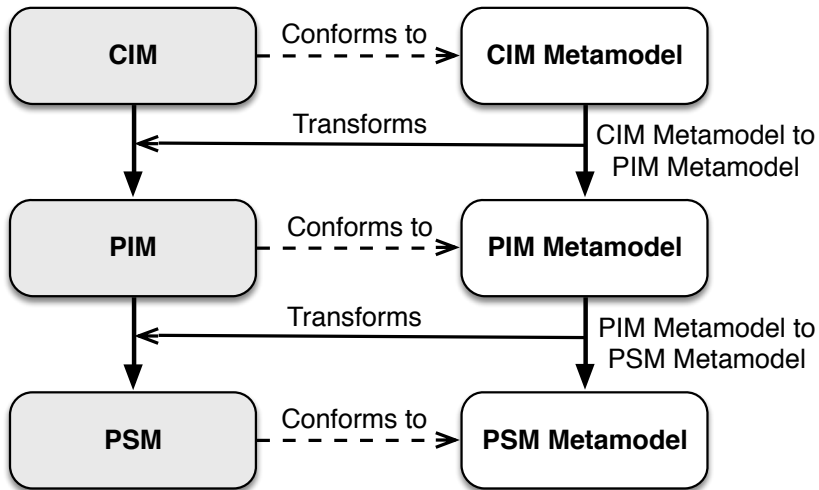


Figure 2.11: Metamodel-based transformations in Model-Driven Architecture (MDA)

Models are categorized into three different abstraction levels:

- *Computation Independent Model (CIM)*. It is often referred as a business or domain model, since it uses a vocabulary that is familiar to the subject matter experts (SMEs). It is totally independent of the technologies that are going to be used for its implementation.
- *Platform independent Model (PIM)*. It focuses on the operation of the system, but abstracts out the specific technologies to implement it.
- *Platform Specific Model (PSM)*. It combines the specifications in the PIM with the technical details of a specific platform (programming language, operating system, etc.).

Each model conforms to a metamodel, which represents the elements and relationships of any of its instances. Metamodels are also instances of meta-metamodels, which are represented in the OMG specification of the Meta-Object Facility (MOF) [92]. MOF is the metamodeling architecture of UML. Consequently, it is possible to define MDA metamodels and models in UML.

In relation to the definition of model transformations, the MDA standard uses the MOF Query/ View/ Transformation (QVT) OMG specification [90]. However, it is also very common to specify transformation rules through the ATLAS Transformation Language (ATL) (<http://www.eclipse.org/at1>),

which is supported by the Eclipse Modeling Framework (EMF, <http://www.eclipse.org/emf>).

Transformations are defined on the metamodels, so as to be able to automatize the transformation process of their respective instances. For example, by establishing the transformation rules between the CIM and the PIM metamodels, it is possible to derive a transformation between the CIM and PIM instances. One of the main benefits of this transformation approach is the reusability of the transformation rules, since they can be applied to any instance of the metamodels that they relate.

2.3.2. A Comparison between MDE and Code-Centric Developments

Many authors have extensively compare MDE and code-centric developments. MDE is generally considered as an “slow” development methodology for small-scale software projects, since it requires too much time and efforts to produce models and use them to obtain software prototypes [37]. However, MDE is commonly said to be adequate to only deal with very complex projects, since it produces high quality software that can be easily reused, maintained and extended [78].

In any case, code-centric developments can be initially

“faster” and it is easier to produce software prototypes from the beginning of the development process. Nonetheless, MDE should not be considered neither “slower” nor as a development process exclusive to large and complex projects. In MDE it is generally simpler to produce code, validate and maintain it, independently of the scale of the project to be developed [60]. As illustrated in Figure 2.12, in MDE the efforts are decreased over time, whereas, as illustrated in Figure 2.13, in code-centric developments, the efforts are constant.

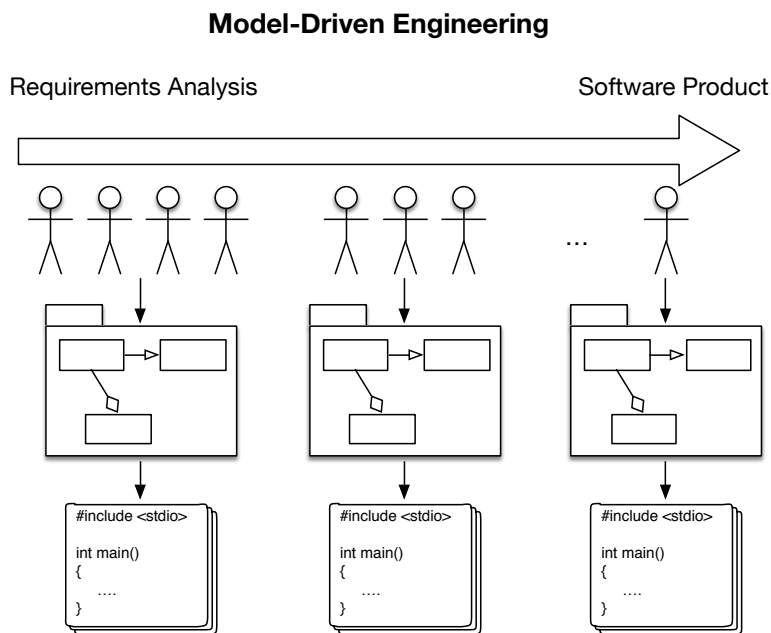


Figure 2.12: Scheme of the MDE process

Moreover, in MDE, the separation between designing and coding is clearer than in code-centric developments. Consequently,

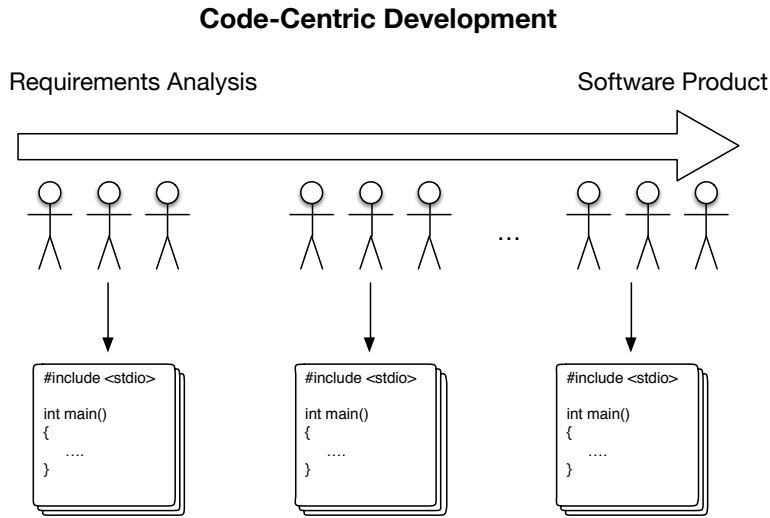


Figure 2.13: Scheme of the code-centric development process

in MDE, software engineers tend to design very detailed models that are implemented by programmers [49]. On the other hand, in code-centric developments, all the members of a development team should have similar skills, thus making it very difficult to set up a large team to manage complex software projects. Additionally, in MDE is easier to parallelize the development of different projects, since, as a project progresses, several members of a team can start to dedicate design efforts to other projects.

Furthermore, it has been experimentally tested that MDE is able to reduce development costs, even by half (approximately), in small-scale projects [60]. In fact, MDE not only decreases the development costs of a project, but it also incrementally decreases the

cost of future projects, as it promotes the reusability of the produced models. Besides, the maintenance and extension costs are also reduced, since it is usually easier to extend models to incorporate new functionalities (or to modify the existing ones), rather than modifying the code. To that respect, MDE is also optimum for reducing the cost of integrating legacy models into new software designs [4]. Additionally, in MDE is possible to simulate a system (or certain parts of it) on the basis of the models that are used to specify it [27]. Thereby, it is feasible to detect operational failures without an actual implementation of a system, which further decreases development costs in many cases.

In spite of the benefits that MDE offer, it also have several drawbacks [49]:

- **Redundancy.** In MDE is usual to provide different representation of the same artifacts, representing different perspectives or abstraction levels of the same concepts. Therefore, redundancy issues may arise, thus requiring a continuous consistency checking of the produced models. Anyhow, automatized model checking tools may overcome this problem.
- **Rampant round-trip problems.** To keep separate models as lowly interrelated as possible is difficult, since as systems grow in complexity it is increasingly arduous to clearly sepa-

rate abstraction levels. The interrelationship between models decrease their maintainability level, since a change in a model needs to be reflected in other models. Reverse engineering techniques may help to avoid that issue.

- **Moving complexity rather than reducing it.** Sometimes abstract models only reduce complexity at a certain development phase (e.g., design), but increase complexity in other phases (e.g., implementation), since some details are completely skipped. The problem is to be able to detect the degree of abstraction that models should have during the whole development cycle, so as to not move the whole complexity of a project to a certain phase only.
- **More expertise required.** Correctly defining, using and transforming each model requires a certain set of very particular skills, while all the models need to be exchanged, improved and understood by all the members of a development team.

To summarize, MDE overcomes most of the issues that code-centric developments have. However, it is still necessary to improve certain aspects of this development methodology by researching new reverse engineering techniques or proposing new model checking tools, among other things. Regardless, it is apparently certain that

future developments will move the complexity from the implementation to the other phases of the software engineering cycle. This fact further motivates the research work behind this thesis, since it reinforces the need of avoiding important tasks (like defining all the aspects related to the communications) to be relegated to the implementation, instead of being tackled during the software design.

2.3.3. Developing Communication Mechanisms on the basis of MDE

MDE has proven to be an appropriate methodology to facilitate the development of different aspects of the software communications: protocols, middleware, networking technologies, etc.

For example, in [71] it is presented an approach to develop protocols for client-server architectures on the basis of MDE. The main idea behind this work is to generate code from a set of well-defined models that represent the main features and quality properties of a certain protocol. Moreover, a communications profile for UML is introduced, so as to be able to more easily represent the structural modeling and behavior of a communication protocol through an standard graphical notation. This proposal benefits from MDE in the sense that this methodology facilitates automatic generation of high quality implementations including very complex programming structures that are hard to code even by skilled program-

mers.

A proposal to apply MDE to manage communications in the Internet of Things (IoT), that is, in resource-constrained, mobile and highly dynamic computing systems, like sensors or wearable devices, is presented in [36]. This research work highlights the difficulty of producing high quality software for these types of computing devices without the help of the appropriate supporting abstract models. Moreover, the authors emphasize the need of applying MDE-based methodologies to the development of complex communication environments, which are tough to manage from a merely technical point of view.

MDE can also be applied to model communication schemes, and to automatically analyze them to check their quality, as demonstrated in [120]. For instance, in that research work, it is described the development process of a sensor network supported by wireless technologies. The MDE-based development involves specifying a set of models, checking them to detect issues (i.e., sensors out of the range of the wireless technologies, performance issues, non-satisfied constraints, etc.) and refining them to improve the characteristics of the resulting network. The possibility of simulating the designed sensor networks using the produced models is also pointed out in that research work. Actually, system simulation is one of the most important benefits of MDE, as mentioned in previous Section.

In [60] it is detailed how middleware development can benefit from MDE too. The development costs can be considerably reduced using MDE (in that research work is mentioned that even 2.6 times), while, at least, keeping the same quality level of developing a middleware on the basis of a code-centric development methodology.

Integrating a set of heterogeneous communication technologies is a method to provide their combined benefits in a unique artifact (i.e., a software component, middleware, service, etc.). This is an important task that needs to be performed very commonly nowadays, as a means, for example, to maintain compatibility between newer and legacy technologies, or to fulfill an amalgam of quality properties that are not possible to satisfy with any individual technology. MDE can be suitable to integrate communication technologies, as described in [4], since the integration process is usually manual, very complex and prone to errors.

To conclude, all these works stress the increasing complexity (and cost) of managing communications, specially without the appropriate methodologies or by dealing with them only from a technological perspective.

2.4. Conclusions

This chapter has presented a survey about the communication of information in distributed and ubiquitous systems, and the development of communication mechanisms on the basis of MDE.

Software communication can be very complex, since a number of technical and design decisions need to be made, while taking into account several quality properties: scalability, efficiency, reliability, interoperability, etc. Communicating information in ubiquitous systems further increases complexity, since additional properties need to be fulfilled, like mobility support, transparent adaptation or context-awareness. Software engineers tend to achieve some of those quality goals at implementation level by adopting certain supporting technologies, like middleware. Moreover, since the existing supporting technologies tackle with some functionalities or quality properties, but do not take into account others, they usually integrate heterogeneous ones to fulfill the desired requirements.

MDE can be applied to approach the communication aspects of a ubiquitous system at design time. By addressing communications at design level, it is possible to reuse knowledge across software design and implementation, so as to reduce development costs and produce optimal solutions to common problems. Additionally, the combination of different communication technologies can be

part of the design process, instead of being relegated to the implementation, as traditionally occurred. A direct benefit of using MDE to develop the communication aspects of a ubiquitous system is the mitigation of the negative impact in maintainability and reusability that results from a composition of technologies, patterns, frameworks, etc. without a clear methodology.

Finally, it can be deduced from the information presented in this chapter that dealing with communications directly affects the quality properties to be pursued, the functionalities that can be provided and the final development costs of a software. Consequently, managing communications should not be exclusively considered as an implementation activity: New models, methods and techniques should be proposed to incorporate such a relevant task to the whole development process of a ubiquitous system. To this respect, in the next chapter it is proposed a conceptual model of a communication system. The idea is to clearly expose the most relevant concepts (and their relationships) that are present in any communication system, and to serve as a basis to introduce a MDA-based development methodology for ubiquitous systems in the later chapters.

Chapter 3

A Model for Communication Systems

This chapter presents a **conceptual model** for Computation-Independent Communication Systems (**CI-CS**). As it will be described along this chapter, the conceptual model, which is also presented as a **metamodel**, intends to tackle with some structural and behavioral aspects of the communications that are not currently approached in the main conceptual models for communication systems: Shannon-Weaver [115], SMCR [11] and Barnlund [6] models. In fact, the conceptual model has been devised through the analysis of the expressiveness problems that are present in those current communication models. In order to make it easier to understand the different elements that should be encompassed in order to conceptualize, analyze or optimize a communication system, and to make more explicit the relationships between those elements, **separated**

views of a communication system are offered: **structural** and **behavioral** view. Furthermore, an **ontology** has been devised to provide a formal specification of the conceptual model. To conclude, a qualitative description of the proposal is provided.

3.1. An Introduction to the Communication Theory

Currently, **three communication models** can be considered to provide appropriate and precise descriptions of the process of communicating information.

A technical-oriented approach to communications was firstly presented in the **Shannon-Weaver Mathematical Model of the Communications** [115], which is, by the way, the first publication that introduced the term **bit**. The Shannon-Weaver model (see Figure 3.1¹) establishes that communications always follow the same process: a source sends a message through a channel to a receiver. A *noise source* is an external element to the communication system that alters the transmitted messages while they are being transferred through a channel. Therefore, the contents of the message may differ between senders and receivers.

¹Note that the “channel” element is not named in the original depicting of the model, but it is mentioned in its description

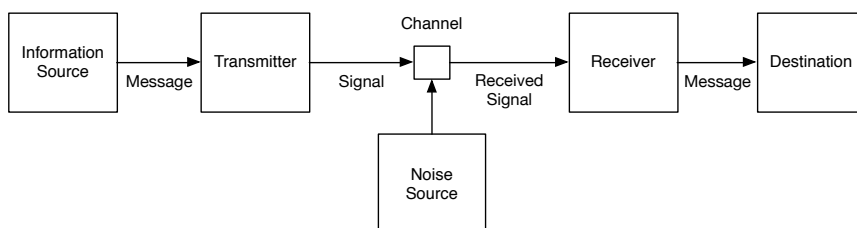


Figure 3.1: The communication model proposed by Shannon and Weaver in the *The Mathematical Theory of Communication*, 1949 [115]

An extension to the Shannon-Weaver model is presented in the widely-accepted **Source-Message-Channel-Receiver (SMCR)** model [11], whose description has been graphically represented in Figure 3.2. The main contribution is that it incorporates the possibility of making the communication process iterative, since a receiver may become a source after receiving a message, in order to provide a feedback. Additionally, the *noise source* concept is removed in order to produce a more abstract communication model.

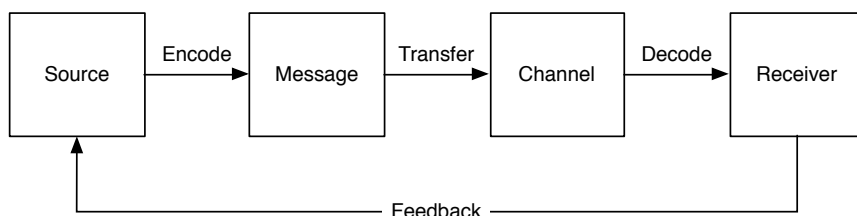


Figure 3.2: The communication model described by Berlo in the *The Process of Communication*, 1960 [11]

Both the SMCR and the Shannon-Weaver models are oriented towards *ordered* social interactions. However, Barnlund proposed

a **transactional model** in which each participant can communicate with itself and to simultaneously be both a receiver and a sender [6]. The model (see Figure 3.3) also reflects the possible presence of communication noise, which could lead to the need to re-transmitting a message.

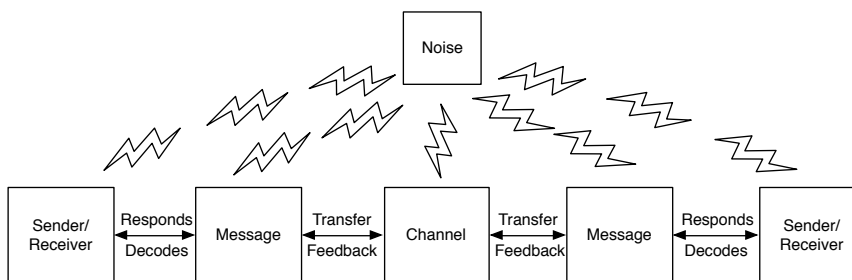


Figure 3.3: The communication model described by Barnlund in *A Transactional Model of Communication*, 1970 (re-printed in 2008 [6])

These models intend to provide a highly abstract perspective of the communications. However, they do not provide an insight of the domain-independent elements that are present in a communication system (i.e., the elements that are present in both human and computer-based communications), the relationships between them, the mechanisms to support their interaction, or how to organize them to fulfill certain requirements. For example, Barnlund's theories clearly specify that both the receiver and the sender need a protocol (a *code-book*) to be able to understand each other, but that term is neither clearly conceptualized nor incorporated into the transac-

tional model. Also, none of the presented models take into account the scenarios in which a message can be simultaneously transferred through diverse channels and according to multiple protocols.

3.2. A Model to Conceptualize a Communication System

This section presents a conceptualization of the elements and relationships that are present in any communication system. The idea is to **extend, complete and formalize** the current, most-widespread communication models, which were presented in previous section, so as to overcome the issues that may arise when characterizing certain scenarios on their foundations.

The overall notion of a communication system have been described on the basis of **two views**:

- **Structural View.** The elements that are present in a communication system, and how they interact.
- **Behavioral View.** The flow of interactions in a communication system to provide a communication-related functionality or to fulfill a quality property.

These views share certain concepts and relationships. Hence, their conjunction conforms a complete **metamodel** (or conceptual

model) that tackles with all the facets of a communication system in a holistic way. Finally, the specification of a Computation-Independent Communication System (CI-CS) has been formalized as an ontology in order to have a basis on which it is possible to describe the capabilities and quality properties of the conceptual model (see next section).

3.2.1. Structural View

The structural view of a CI-CS presents the elements of a communication system, and how they relate to each other.

To work out the general behavior of a participant in a CI-CS, the process described in the current communication models has been analyzed. The devised model is depicted in Figure 3.4 as a UML activity diagram.

As it is illustrated in Figure 3.4, in any communication system, the communication process is initiated when the sender decides to deliver some information (1). After that, the information needs to be prepared to be sent by adopting a protocol (2), the information needs to be formatted according to the specifications of the protocol to produce a certain message (3) and a protocol-compatible channel has to be selected to deliver the message itself (e.g., a software protocol might not be able to tackle with all the existing physical

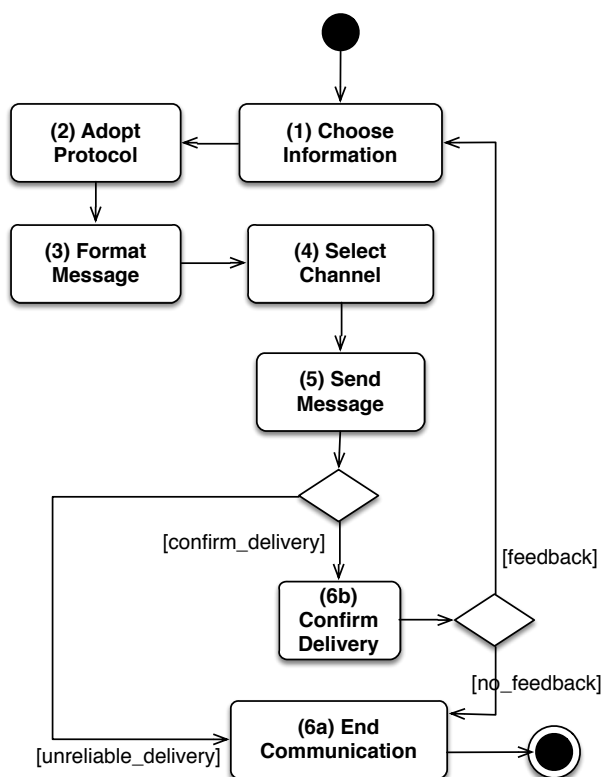


Figure 3.4: A UML activity diagram depicting the dynamic behavior of a CI-CS

channels, but only with a subset of them) (4). Later, the message is sent to a receiver (5). If the communication does not involve a delivery acknowledgement, then the communication process ends (6a). In other case, the message is delivered (6b) and the receiver may provide a feedback, thus acquiring the role of a sender (1).

Consequently, all the existing communication models include the following structural elements:

- **Sender.** The party that initiates a communication.
- **Message.** The information unit to be exchanged during a communication.
- **Channel.** The medium to transfer messages.
- **Receiver.** The destination party of a communication.

Optionally, a *noise source* can also be present in order to represent the possible situation in which an external source modifies in some ways the exchanged messages while they are transferred through a channel.

On the basis of these notions and the analysis of the previous behavioral model, the structural view of a CI-CS has been devised. The view, whose model has been depicted in Figure 3.5 as a UML class diagram, also aims to extend and refine the existing communication models, as it is described below.

In the structural view, whose elements and relationships are defined in Tables 3.1 and 3.2, a communication system is made up of one or more **participants**, **channels** and **protocols**. This way, the notion of communication system is extended in comparison to the previous models, in which the exchange of messages between senders and receivers is made through only one channel and according to a particular protocol.

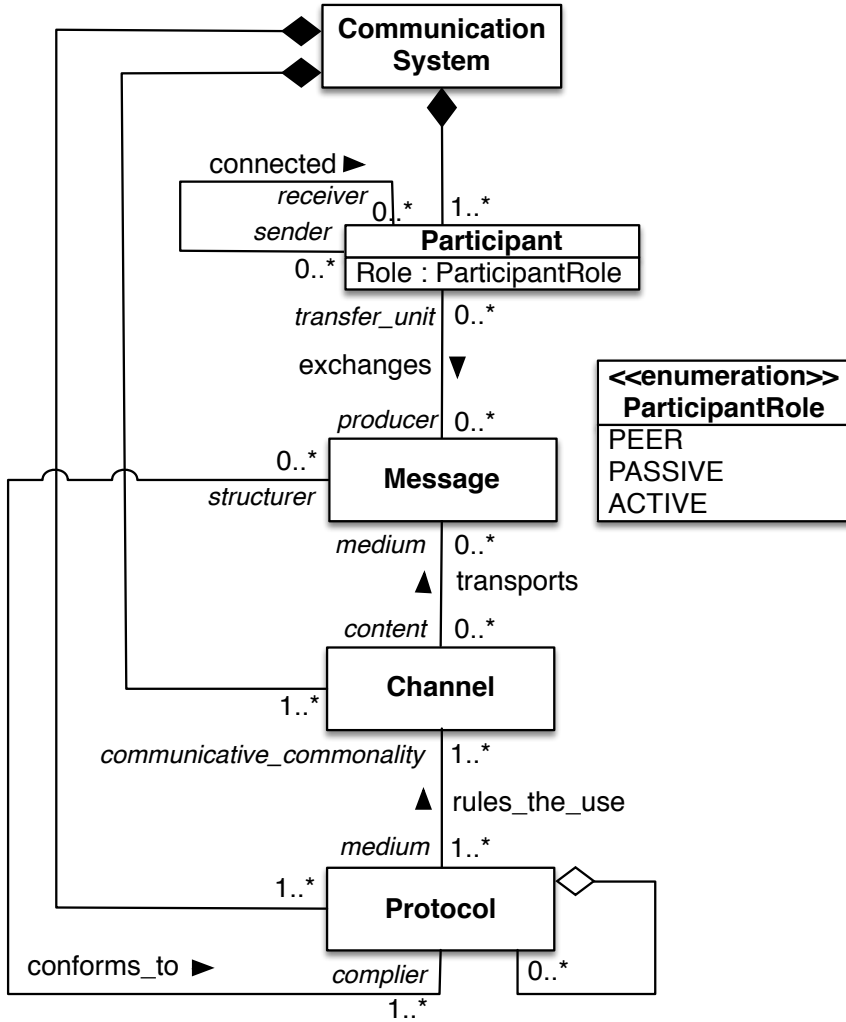


Figure 3.5: The structural view of the CI-CS metamodel, depicted as a UML class diagram

A participant may exchange information with others or with itself (i.e., a communication system may be conformed by a unique participant that self communicates). Besides, each participant may have a different role, according to its **role** attribute: it can be **pas-**

Element	Summarized description
Communication System	It encompasses a collection of parties (or participants) that exchange messages through a set of channel, and conforming to certain protocols.
Participant	A receiver and/or sender of a set messages in a communication system. In a human conversation, it can be one person. In a networking environment, it can be a computing node.
Message	The minimum information unit that is exchanged between participants in a communication system. The ordering and the format of the each message in specified through the communication protocol.
Channel	The medium to transport a message between the different participants in a communication system.
Protocol	It defines the format and ordering of the messages to be exchanged. Additionally, it defines the rules to adequately use the communication channel (i.e., when to use it, how to use it in an optimum way, etc.). Moreover, a protocol can be a composition of other simpler protocols.

Table 3.1: Concepts present in the structural view of a CI-CS

sive (it can only act as a receiver), **active** (it can only be a sender) or **peer** (it can be both a sender and a receiver). In this way, all possible communication scenarios, incorporating different types of participants, can be represented. In contrast, in the existing communication models, some special communication scenarios can not be represented.

For example, in the proposed CI-CS model, a communication

Relationship	Summarized description
<i>Connected</i>	The participants of a communication system connect to each other to exchange messages.
<i>Exchanges</i>	Each participant transfer multiple messages to the participants to which they are connected to.
<i>Conforms_to</i>	A message must conform to a set of certain protocols that are understood by the different participants engaged in a communication.
<i>Rules_the_use</i>	A protocol has a set of rules to adequately use a communication channel (e.g., to use it efficiently).
<i>Transports</i>	A channel is in charge of transferring messages from the sending parties to the receiving ones.

Table 3.2: Relationships between the concepts present in the structural view of a CI-CS

system could be composed only by **passive** participants (i.e., they only act as *receivers*). This way, it is possible to use the model to specify a service-oriented architecture of a software, by only focusing on the representation of the services themselves, and omitting the clients consuming them (i.e., the participants with an active role), but not excluding any other communication elements (i.e., channels, messages and protocols). Also, other communication scenarios could be only composed by **active** participants. For instance, in a ubiquitous system it is common to incorporate participants that constantly send messages to be discoverable by other parties. This way, it is possible to represent scenarios through the proposed model in which all the participants are expecting to be discovered and, there-

fore, they only send messages. Finally, most scenarios in which participants with a **peer** role are involved can not be represented by neither the SMCR nor the Shannon-Weaver models, since they do not take into account the possibility of a participant to concurrently receive and send messages. However, they can be represented through the transactional model proposed by Barnlund.

The CI-CS model assumes that the information unit is a **message**, which can be transported through one or more channels. In this manner, it is possible to represent communication systems that integrate heterogenous channels, so as to improve the interoperability between the parties (i.e., to use specific channels for specific participants) or to open up the possibility of concurrently exchanging messages using different channels in order to fulfill certain quality properties (i.e., performance, reliability, etc.).

A protocol, as defined in the proposal, establishes the rules to use a channel (e.g., a software network protocol, a grammar in a language, etc.). Moreover, a protocol can be the composition of other simpler protocols, which is very common in practice. For example, the CORBA IIOP is a protocol that combines the GIOP abstract messages with their implementation as TCP/IP messages [87]. Furthermore, the CI-CS model allows the specification of communication systems that integrate diverse protocols, which is an important requirement in many systems to be able to achieve a certain level of

interoperability or quality of service (QoS) [101].

Finally, it is worth to be noted that the proposed model does not include the notion of *noise source*, since it is assumed that the possibility of incorrectly transferring a message is an intrinsic problem of any channel, and should not be part of the characterization of a communication system (i.e., the noise source should be present in the characterization of a channel, but not in the characterization of the whole communication system). Thereby, CI-CS model considers that each communication process is a transaction in which the messages are either correct or need to be re-transmitted. The mechanisms to detect an erroneous message and to send it again should be present in the specification of the protocol.

3.2.2. Behavioral View

The behavioral view of a communication system exposes how its elements can be organized to provide a communication-related functionality or to meet a quality property. The idea is to interpret a communication system as **a collection of interactions intended to achieve certain goals**.

Due to the similarities between the notions present in BPMN 2.0 (see Chapter 2, Section 2.1.3) and the ones present in the structural view of a CI-CS, this standard has been taken as an inspiration

to understand the notions behind the behavioral view of a communication system. Particularly, BPMN 2.0 choreographies have been considered in this thesis work as an appropriate manner to represent the behavioral aspects of a communication system, and the collaboration between its elements. Some authors have previously identified BPMN in general, and in particular BPMN choreographies as an appropriate notation to represent communication systems [62] [18] [46]. In contrast with UML-based notations (like communication, sequence or activity diagrams), BPMN can be more expressive to propose computation-independent models of a system [70].

In consequence, the behavioral view of a CI-CS, whose model is depicted in Figure 3.6 as a UML class diagram, has been devised on the basis of the BPMN 2.0 choreography metamodel. The concepts and relationships present in the behavioral view of a CI-CS are, respectively, described in Tables 3.3² and 3.4. The naming conventions of the BPMN choreography metamodel have been kept in order to facilitate the matching between a BPMN diagram and the proposed behavioral view of a CI-CS (i.e., this matching between both models will be explored in Chapter 5, Subsection 5.3.2, as a key part of the software engineering methodology to be presented in that chapter).

²The elements that are shared with the structural view are not described again in Table 3.3

In contrast with the BPMN 2.0 metamodel, the behavioral view focuses on the concepts that are present in a CI-CS, rather than on the graphical notation to represent the concepts themselves in a diagram. Thus, the behavioral view of a CI-CS does not include the elements and relationships present in the BPMN 2.0 metamodel in relation to graphical notations. Also, the behavioral view includes some elements present in the structural view presented in previous subsection. Namely, **participant** and **message** elements. It is also important to note that the BPMN choreography metamodel includes an element called *connection*, which is equivalent to the *link* element in this proposal. The reason behind this name mismatch is to avoid misconceptions of this term, since the notion of *connection* is commonly used in the communication field to refer to the initial interaction between two participants.

In the behavioral view, a communication system *may* contain multiple **choreographies**, since it is taken into account the possibility of modeling a communication system that does not operate at all, and also the specification of a communication system with different choreographies.

The multiple choreographies are intended to organize the flows of interaction between the **participants** of a communication system. This way, each choreography consists of the sequence of **activities** to be carried out by the participants. Particularly, in each

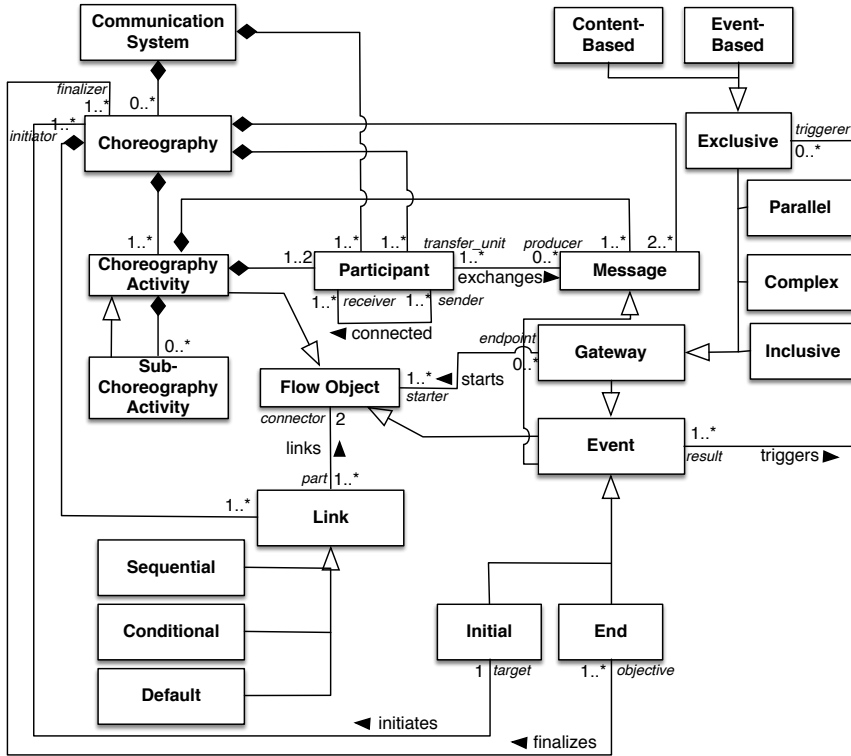


Figure 3.6: Behavioral view of the CI-CS metamodel, depicted as a UML class diagram. The model is inspired by the BPMN 2.0 Choreography Metamodel Specification [89]

activity, one or two participants exchange several **messages** (i.e., an activity is considered as a transaction in which either a participant communicates with itself or communicates with another participant to deliver messages). Additionally, an activity can be composed by several **sub-activities** representing an **atomic interaction** between two participants, involving the exchange of only one message (i.e., this constraint requires OCL to be represented in the UML class diagram, but this notation has been omitted from the figure in order

Element	Summarized description
Choreography	A possible flow of interactions between the participants in a communication system.
Choreography Activity	An interaction activity between one or two participants. An activity may be composed by multiple subactivities .
Flow Object	The different elements that are part of the execution flow of a choreography, that is choreography activities and events .
Link	A connection between two or more flow objects. There are three types of links: sequential (it directly joins two flow objects), conditional (it only joins two flow objects if a condition is satisfied) and default (the flow object that is initiated when no previous conditional interaction were satisfied).
Event	An specific type of message that represents a significant occurrence in the communication system at a given moment.
Gateway	An event that executes one or more flow objects (i.e., it initializes a choreography activity or makes an event to be delivered). There are four types: exclusive (given a received event, a flow object is chosen to be executed), inclusive (several flow objects are waited to be finished/delivered, then a specific one is executed), complex (a predicate should be satisfied before executing a flow object) and parallel (several flow objects are concurrently executed). Exclusive gateways can be event-based (i.e., any event can trigger it) or content-based (i.e., the triggerer event needs to contain certain information).

Table 3.3: Definition of the concepts present in the behavioral view of a CI-CS

Relationship	Summarized description
<i>Links</i>	A link connects two flow objects in a choreography.
<i>Starts</i>	A flow object may be initiated by multiple gateways.
<i>Triggers</i>	An event triggers an exclusive gateway.
<i>Initiates</i>	An <i>initial</i> event starts a choreography.
<i>Finalizes</i>	One or more <i>end</i> events may finish a choreography (i.e., the execution flow of a choreography may have different endings, but each of them is associated to an <i>end</i> event).

Table 3.4: Description of the relationships between the concepts present in the behavioral view of a CI-CS

to simplify it).

At the end of each activity, the execution flow of the communication system continues as specified in the associated choreography. Therefore, the execution of an activity may lead to the execution of other **activities** or the occurrence of **events**. These elements are named in the model as **flow objects**, that is, the elements that are part of the execution flow of a choreography.

An **event** is a type of message that is used by participants to notify a significant occurrence in the communication system to the other participants. For example, it could be the notification of the arrival of a new participant, the unavailability of an element in the system, the initialization of a choreography (i.e., an **initial** event), the finalization of its operation (i.e., an **end** event), etc. It is worth

to be mentioned that an event is both a type of message and a flow object. Thereby, it is possible to represent scenarios in which an activity involves the notification of an event, or the notification of an event is just part of the execution flow of a choreography of a communication system.

A **gateway** is a specialized event that is delivered to initiate a set of interactions, that is, to modify the usual execution flow of a choreography in different ways:

- **Parallel.** Several interactions are concurrently initiated.
- **Exclusive.** Given several target interactions, only one of them is executed. It allows to represent a split in the execution flow of a choreography. The selection of a specific interaction is based either on the occurrence of an event (**event-based**) or on the delivery of an event with certain contents (**content-based**).
- **Complex.** Similarly to an exclusive gateway, it splits the execution flow of a choreography. However, the selection of the interaction to initiate is based on the satisfaction of a predicate.
- **Inclusive.** Several interactions are being waited to be finished, then an interaction is initiated.

On the basis of all these notions, it is possible to define a **link** as the relationships that are pre-established in a choreography between each two flow objects (i.e., activities or occurrence of events). A link, in the real work, could be a pre-assigned turn to allow the participants to communicate in a certain order. In a Petri Net, it could be the *token* that allows to transition to certain *places*.

In the behavioral view of a CI-CS there are three types of links: **sequential**, **conditional** and **default**. Sequential links directly connect two flow objects. Conditional links are only followed when a certain condition (or predicate) is satisfied. Finally, default links are followed when any conditional link was triggered, that is, if the predicates associated to a set of conditional links could not be satisfied.

In any case it is necessary to specify at least one link that relates the initialization of a choreography and its finalization (i.e., the delivery of an *initial* event and an *end* event). Conversely, it is necessary to have at least two messages involved in a choreography (i.e., one to deliver the *initial* event and another for the *end* event).

To conclude, the behavioral view of a CI-CS is able to expose the organizational aspects of a communication system. This is a novelty in comparison to previous conceptualizations of the communication systems, which only tackled with their structure and basic

operational facets.

3.2.3. Formal Specification as an Ontology

The views of a CI-CS share certain concepts. As so, their conjunction conforms a **metamodel** of a CI-CS. The metamodel of a CI-CS, which has been semi-formally depicted using UML in previous subsections, can be **formalized** to demonstrate its semantic capabilities and quality properties (see next section). To do so, an **ontology** of a CI-CS has been defined.

In Computer Science, an ontology is defined as “a specification of a conceptualization” [45]. An ontology **formally** describes the topics in a domain, the relations between them and their constraints. Even if ontologies can be specified through different languages, **Web Ontology Language (OWL)** (<http://www.w3.org/standards/techs/owl>) is one of the most well-known ones. OWL is based on Description Logics (i.e., a formalism that is appropriate to represent knowledge) and is a W3C Recommendation. Also, **Protégé** (<http://protege.stanford.edu>) is the most used editor for ontologies, and is compatible with OWL.

Figures 3.7 and 3.8 respectively show the output that generates Protégé to graphically represent the CI-CS ontology, and its hierarchy of classes and properties. Note that in the graphical repre-

sensation of the CI-CS ontology some elements have been removed in order to avoid the figure to be overcluttered. The whole OWL implementation is provided in Appendix I.

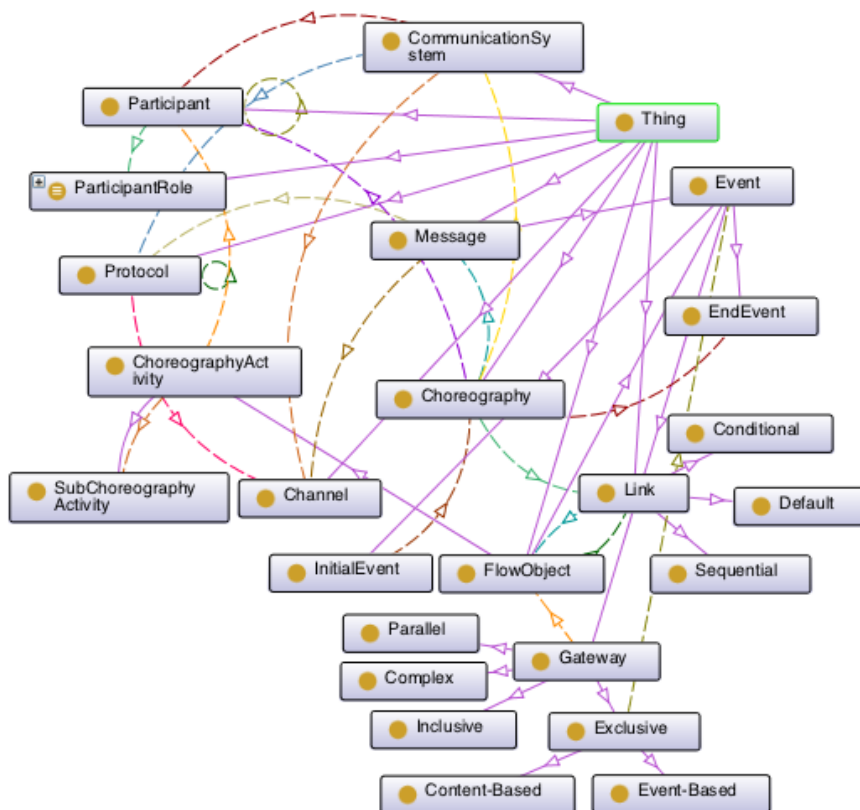


Figure 3.7: Graphical representation of an ontology of a CI-CS

A benefit of the ontological representation of a CI-CS is that it can be **shared and reused** by different groups of people or software tools to analyze a communication system in order to:

- **Check for inconsistencies** in the specification of a communication system. For example, it is possible to detect the need of



Figure 3.8: The class hierarchy and properties of an ontology of a CI-CS, as represented in Protégé

including certain participants to be able to carry out required interactions, messages to carry out these interactions, protocols that are able to codify those messages, channels compatible with those protocols, and so forth.

- **Simplify** a communication system by minimizing its elements and interactions. To do so, it could be possible to establish

equivalencies between different members (i.e., instances, individuals) of the same class, which could lead to the simplification of a communication system. For example, if two participants need to interact several times during the lifespan of a communication system, it could be considered to merge both participants into a unique one, so as to reduce the number of participants and interactions. This consideration could lead to the optimization of the communication system through the reduction of the number of messages, the complexity of the protocols, etc.

- **Integrate** several communication systems by including the members of each one to be integrated (i.e., the instances or individuals associated to each class in the ontology, that is, participants, messages, channels, protocols, etc.) into a unique communication system. Again, as mentioned above, it is possible to check for inconsistencies and simplify the resulting, integrated, communication system, which is usually considered as a complex task [101].

Besides, previous activities can be automatically done through existing **reasoners**. A reasoner is a piece of software that “infers logical consequences from a set of explicitly asserted facts or axioms and typically provides automated support for reasoning tasks such

as classification, debugging and querying” [29]. Some of the most well-known reasoners are FaCT++, RacerPro, Pellet, KAON2 and Hoolet [29].

3.3. Quality Attributes of the Communication Model

The international standard **ISO/IEC 25010:2011** [54] establishes a quality model for software products on the basis of eight attributes: **functional suitability, reliability, performance efficiency, operability, security, compatibility, maintainability and transferability**.

On the basis of the standard and the research work presented in [13], a **qualitative analysis** of the proposed metamodel have been made. In particular, the proposal presented in [13] has been adapted to the newer ISO/IEC 25010:2011, since it was published in 2010 and adopts the ISO/IEC 1926 standard as its basis, which has been deprecated since then on behalf of the ISO/IEC 25010:2011 one. The definitions of the quality attributes have been adapted from the more general software product context, which is the one presented in the standard, to the metamodeling context.

The following subset of the quality attributes presented in the ISO/IEC 25010:2011 standard, and some of their corresponding

sub-characteristics, have been taken into account:

- **Functional suitability.** The degree to which the metamodel provides functions that meet stated and implied needs, when the metamodel is used under specified conditions.
- **Reliability.** The degree to which the metamodel can maintain a specified level of performance, when used under specified conditions.
- **Performance efficiency.** The degree to which the metamodel provides appropriate performance, relative to the amount of resources used, under stated conditions.
- **Operability.** The degree to which the metamodel can be understood, learned, used and attractive to the user, when used under specified conditions.
- **Compatibility.** The ability of the metamodel to exchange information with other metamodels and/or to perform their required functions while sharing the same domain.
- **Maintainability.** The degree to which the metamodel can be modified. Modifications may include corrections, improvements or adaptation of the metamodel to changes in environment, and in requirements and functional specifications.

- **Transferability.** The degree to which the metamodel can be transferred from one environment to another.

Note that the security attribute present in the ISO/IEC 25010:2011 standard has not been taken into account, since it is not suitable to describe the quality of a metamodel [13]. Likewise, the quality attributes have several sub-characteristics that are compiled in the standard, and some of them are not suitable to describe the quality of a metamodel. For example, those related to the dynamic behavior of a software (e.g., the *time behaviour* sub-characteristic of the *performance efficiency* quality property) are not appropriate since a metamodel is an static, non-executable and conceptual model [13]. In addition, some of the sub-characteristics related to standards compliance for certain characteristics, like reliability, compatibility or operability, are defined in the standard for generic software products, but they can not be matched to the context of the metamodel description (i.e., no standards exist to asses the compliance of certain characteristics in the definition of a metamodel).

Table 3.5 summarizes the analyzed quality attributes and sub-characteristics of the CI-CS metamodel. Some of the attributes and sub-characteristics have been assessed through the ontological representation of a CI-CS. Appendix II describes how the proposed me-

the model fulfills the multiple attributes and sub-characteristics. Also, a definition of the sub-characteristics is provided in that appendix.

Attribute	Sub-Characteristic	Fulfilled
Functional suitability	Appropriateness	✓
	Accuracy	✓
	Interoperability	✓
	Compliance	✓
Reliability	-	Partially
Efficiency	-	✓
Operability	Recognizability	✓
	Learnability	✓
	Helpfulness	✓
	Attractiveness	Partially
Compatibility	-	Partially
Maintainability	Modularity	✓
	Reusability	✓
	Analyzability	✓
	Changeability	Partially
	Testability	✓
Transferability	Adaptability	✓
	Portability	Partially

Table 3.5: Quality attributes of the CI-CS metamodel

3.4. Conclusions

This chapter has presented a metamodel for Computation-Independent Communication Systems (CI-CS). The metamodel conceptualizes the notion of Computation-Independent Communication Systems (CI-CS) and it is specified through two different views: structural and behavioral view.

The structural view captures all the structural artifacts that are present in a communication system. To do so, the most well-known and accepted communication theories have been studied. Conversely, the behavioral view includes the elements that should be present in a communication system to model its organization. The idea is to model the behavior of a communication system as a choreography, that is, as an ordered sequence of interactions between the participants of the communication. Since BPMN 2.0 includes a choreography metamodel, it has been taken as a reference to define the elements and relationships that should be present in the behavioral view.

The conceptualization of a CI-CS has been formalized through an ontology. The ontology can be used to check if a model is *consistent* with the defined conceptualization of a communication system, that is, if a model contains all the necessary elements and relationships to be considered as a communication system (i.e., if a given model is consistent with the proposed conceptualization of a communication system). Moreover, the ontology could allow to optimize a communication system by establishing equivalencies between certain elements. Conversely, multiple communication systems could be integrated into a unique one in an optimum way (i.e., by minimizing its elements through the identification of equivalencies between the elements and relationships of the several commu-

nication systems to integrate).

On the basis of the international standard quality framework for software systems defined in ISO/IEC 25010:2011 some quality properties present in the proposal have been studied. The ontology assists in that study, since it allows to assess certain features of the metamodel.

Finally, the conceptual model described in this chapter can be considered as a CIM metamodel for Communication Systems (CS-CIM) that will serve as the foundation to propose an MDA-based development methodology for ubiquitous systems in subsequent chapters of this thesis work. As a necessary upcoming step to achieve that goal, the next chapter approaches the metamodeling of the mechanisms supporting the communications in a ubiquitous system through the specialization of the defined communication model.

Chapter 4

A Communication Model for Ubiquitous Systems

This chapter describes a communication model for platform-independent ubiquitous systems (PI-US), which, in turn, serves as a metamodel for the communication mechanisms related to these systems. The metamodel includes the elements and relationships that are necessary to represent the specific communication mechanisms that could be integrated in any ubiquitous system in order to support message exchanging, event distribution and dynamic discovery functionalities through software mechanisms. Similarly to the metamodel of a CI-CS, the metamodel of a PI-US is presented through two views: structural and behavioral view.

The metamodel has also been formalized as an ontology, with the objective of demonstrating certain of its capabilities. Moreover,

the relationships between a CI-CS and a PI-US have also been formalized. On one hand, the idea is to show that, conceptually, a ubiquitous system can be considered as a communication system implemented through software mechanisms, and taking into account the specific requirements and quality properties of this kind of distributed systems. On the other hand, it is possible to demonstrate that a *forward transformation* from a CI-CS to a PI-US (i.e., from a more abstract level to a more refined one) is feasible, and some elements in the resulting PI-US model can be traced back to the initial CI-CS. The benefits of these mappings between models will be explored along the chapter.

Finally, the quality properties that can be attributed to the PI-US metamodel will be analyzed and described.

4.1. Communication Functionalities of a Ubiquitous System

As previously mentioned in Chapter 2, Section 2.2, a ubiquitous system can be considered, from a technical point of view, as a distributed system with certain specific requirements: volatility in the communication process and constant mobility of the participants. In consequence, ubiquitous systems usually need to make use of several mechanisms to provide, at least, the following communi-

cation functionalities [69]:

- **Message exchanging:** One-to-one communications between participants. In practice, this type of communication is commonly used for exchanging messages between applications and services.
- **Event distribution:** Participants may send events to notify changes in their internal state to a set of other interested participants. For example, in a sensor network, whenever a sensor measures a significant value, then an event could be sent, so as to execute certain actions on its reception. Event distribution can also improve the decoupling between senders from the receivers, which contributes to reach the mobility requirements of the ubiquitous.
- **Dynamic discovery of participants:** Dynamic detection of new participants, allowing to exchange messages and/or distribute events among them. Due to the mobile nature of the participants, it is necessary to have this feature in order to detect the available participants at a given moment, like the reachable services around the physical user environment.

The conceptualization of a CI-CS presented in previous chapter can be taken as a foundation to define a **ubiquitous system** as

a a computation-based communication system that includes the mechanisms to support the above-mentioned functionalities.

Nonetheless, note that this definition of a ubiquitous system is only focused on the **communication mechanisms** of a ubiquitous system, due to the orientation of this thesis work. In fact, this definition does not take into account other important aspects like the presentation (i.e., user interface), the underlying information systems that may be required, etc. Those facets of a ubiquitous system are not the focus of this work.

Finally, the above mentioned functionalities are already supported by several standards and/or well-known middleware technologies, like ICE, CORBA, DDS, WCF, etc., which were explained in Chapter 2, Section 2.1.4.1. However, some of these proposals focus only one functionality (like DDS or SOAP, which, respectively, provide event distribution and message exchanging) and the others provide a very detailed and technical specification of each functionality (like CORBA, DDS or WCF), which makes it difficult to find many equivalencies in the specification of a functionality between different standards or middleware specifications. Therefore, it is complex to approach the integration of multiple standards and middleware technologies into a unique ubiquitous system, due to the different technical details that should be taken into consideration and to the potentially unrelated notions that may be present in their

corresponding specifications.

In contrast, the PI-US metamodel to be introduced in the next section intends to provide the concepts to support all the above mentioned functionalities, but providing a less technically oriented and more abstract perspective than the existing proposals. The idea is to be able to easily match between the concepts in the metamodel and the *core* (and shared) concepts present in the most well-known existing standards and middleware specifications, so as to be able to facilitate the integration of heterogeneous communication technologies using a platform-independent model for the conceptualization of a ubiquitous system.

4.2. General Communication Model for Ubiquitous Systems

This section presents a conceptualization of the mechanisms supporting the communication in a ubiquitous system. The idea is to specialize the proposed CI-CS conceptual model presented in previous chapter to include the mechanisms to provide the communication functionalities mentioned in previous section. The specialization of the CI-CS metamodel seeks to support the peculiarities of the ubiquitous systems by means of a metamodel for **Platform-Independent Ubiquitous Systems (PI-US)**. Similarly to the CI-CS

conceptual model, the metamodel of a PI-US will be described on the basis of **two views**:

- **Structural View.** The mechanisms and basic structural elements that are present in any ubiquitous system, independently of the specific computing platform, and how they interact with each other.
- **Behavioral View.** The flow of interactions between the elements presented in the structural view.

In consequence, the notions featured in these views have been conceptualized through two models that converge into a complete metamodel that integrates all the mechanisms related to the communications in ubiquitous systems. To conclude, the specification of the metamodel has been formalized as an ontology in order to have a basis on which it is possible to study the capabilities and quality properties of the metamodel (see next section), and to check the compliance of the metamodel with the formal conceptualization of a CI-CS. This way, it will be possible to assess that a ubiquitous system is, in an abstract way, a communication system.

4.2.1. Structural View

The structural view of the PI-US metamodel includes the concepts that are present in a ubiquitous system to support **message exchanging**, **event distribution** and **dynamic discovery**, independently of any specific computing platforms. The rationale to devise the structural view, and define the concepts and relationships that should be present in its model, is based on the study and analysis of the existing communication paradigms (see Chapter 2, Subsection 2.1.2), some existing standards and other widely accepted technology-oriented proposals.

For **message exchanging**, the **RR communication paradigm** has been taken as a reference (see Chapter 2, Subsection 2.1.2.1), since it is considered the most “generic” form of message passing [101]. Moreover, some standards related to message exchanging have been analyzed. Particularly, Hypertext Transfer Protocol (HTTP) (<http://www.w3.org/Protocols/rfc2616/rfc2616.html>), which is the standard Internet protocol, and SOAP (<http://www.w3.org/TR/soap/>), which is the W3C’s standard approach to RPC, have been specially considered.

Both in the RR paradigm and in these standards, **message exchanging** refers to the idea that **participants send messages to**

make a request to other participants, which, in turn, reply with a response message. Consequently, this thesis work adopt that description to refer to the process of exchanging messages in a ubiquitous system. This process is depicted in Figure 4.1, so as to illustrate the assumed message exchanging functional behavior in a better way.

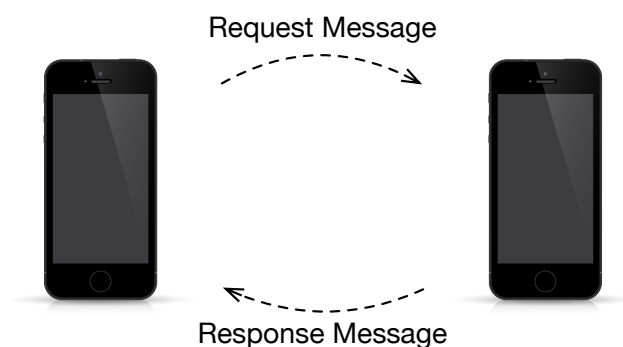


Figure 4.1: Simplified interaction process involved in the message exchanging communication functionality of a ubiquitous system

Note that the specific operation mode of the message exchanging functionality is not considered in that description. Consequently, a response could be asynchronously delivered, there could be batch requests, etc. This way, the description of the message exchanging functionality can be applied to different types of message passing (e.g., asynchronous or synchronous message passing, RPC with batch requests, etc.).

Event distribution has been assumed to conform to the

PubSub paradigm (see Chapter 2, Subsection 2.1.2.3), since this paradigm describes a widely-accepted way of delivering events among the participants of any communication system, and particularly, of any ubiquitous system. Furthermore, this paradigm promotes the decouplement between the publishers and the subscribers, which assists in the fulfillment of mobility support and volatility of the communications [5]. Thus, event distribution, as assumed to be carried out in this thesis work, should involve two different processes: **subscription** and **publication**.

Event subscription consists of a participant in the communication informing an “intermediate entity” that it needs to receive certain sets of events, thus becoming *subscribers* of these events. At the same time, other participants can publish events, which consists of delivering them to the “intermediate entity”, which, afterwards, will notify the published events to the appropriate subscribers. The “intermediate entity” is usually named as an **event handler** and it will be considered to be a piece of **replicated** software associated with each participant, and capable of managing subscriptions and delivering published events to other replicas of itself. The notion of event handler is adopted from the specifications of some middleware solutions, like ICE, WCF or DDS. The overall event distribution process is depicted in Figure 4.1, so as to illustrate the behavior of this communication functionality.

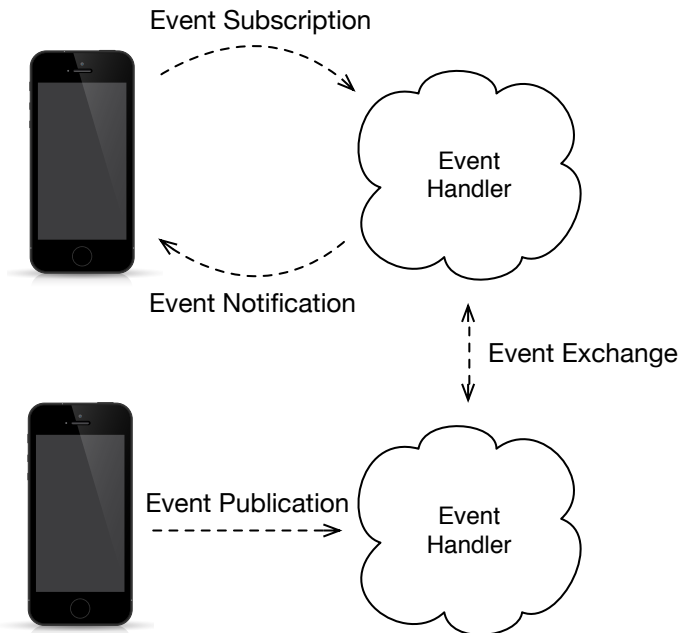


Figure 4.2: Assumed interaction process involved in the event distribution communication functionality of a ubiquitous system

The event handler has been considered to be replicated, since a centralized event handler (e.g., a centralized service) limits the mobility of the participants in the communications associated to a ubiquitous system, while the mobility is one of the most important requirements of this type of systems. The reason is that the participants could only move around the physical areas in which the centralized event handler is available, in order to be able to make use of the event distribution functionality. For example, if a centralized event handler is implemented as a service in the “cloud”, then the participants could only physically move around the areas in

which an Internet connection is available, considering that they will need to interact with that service to distribute events. Additionally, if a ubiquitous system with a replicated event handler is required to have a centralized one (e.g., to improve performance, for any technical reasons, etc.), then a centralized entity (e.g., a service) that orchestrates (i.e., coordinates and organizes) the different replicas of the event handler should be considered to be part of the ubiquitous system itself. Therefore, the assumption that a replicated event handler is more appropriate for ubiquitous systems does not limit the possibility of taking into consideration to manage the event distribution through a centralized manager of the replicas of an event handler.

Some standards have been studied to understand how the process of **dynamically discovering participants in a ubiquitous system** is usually carried out. Particularly, the IETF's Zeroconf standard (<http://www.zeroconf.org>) and the set of networking protocols associated to the computer industry initiative for universal discovery of devices (i.e., a *de facto* standard), a.k.a. Universal Plug and Play (UPnP, <http://www.upnp.org>), have been examined. Both Zeroconf and UPnP standards consider the process of discovering a participant as a sequence of multiple notifications of events. For instance, in both standards, the participants continuously notify events about their own presence. The other participants,

whenever they are notified of one of these events, consider that the delivering participants are available. If a participant does not notify about its own presence for a certain time interval, then it is considered by the others to be unavailable. Thus, dynamic discovery is assumed in this thesis work to be an **specialization of the event distribution functionality**, in conformance with the previously mentioned standards. The overall process to dynamically discover participants in a ubiquitous system have been depicted in Figure 4.3.

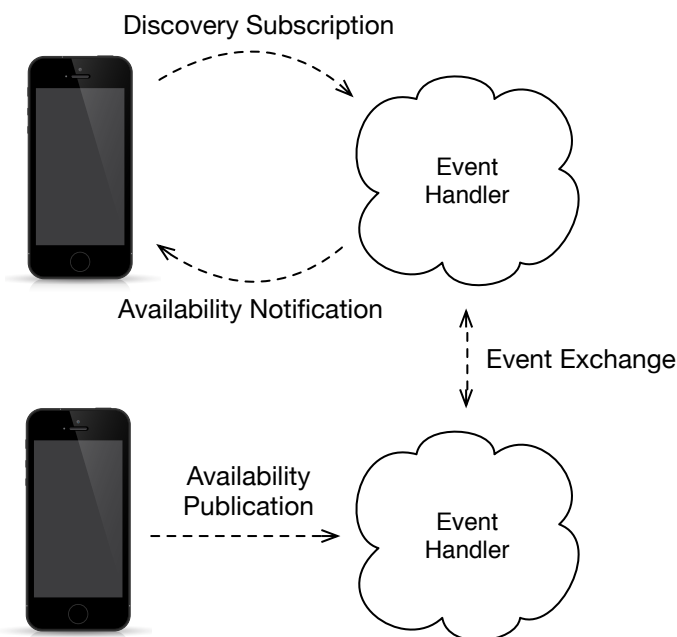


Figure 4.3: Assumed interaction process to dynamically discover participants in a ubiquitous system

In the previous figure, each discovering participant subscribes for events related to discovery, and each participant to be discov-

ered delivers an event notifying about its own availability. Hence, the discovery process could involve unsubscriptions to the discovery events, in order to stop discovering participants. Moreover, a participant could stop publishing availability events in order to prevent to be discovered by others (e.g., for privacy reasons, to save networking resources, etc.). Thereby, the assumed discovery process also contemplates possible privacy and performance requirements.

On the basis of the previous analysis and assumptions associated to the different communication functionalities that should be present in a ubiquitous system, the structural view of the PI-US metamodel has been devised. It is depicted in Figure 4.4 as a UML class diagram. A brief description of the shared elements among the different communication functionalities is present in Table 4.1. Likewise, in Tables 4.2 and 4.3, respectively, each of the elements (grouped by the communication functionalities that they support) and relationships of the structural view of the PI-US metamodel are described.

As it is depicted in Figure 4.4, **a ubiquitous system is conformed by a set of software agents that exchange messages using specific networking technologies and according to a set of software protocols.**

Before describing in detail the devised metamodel, it is neces-

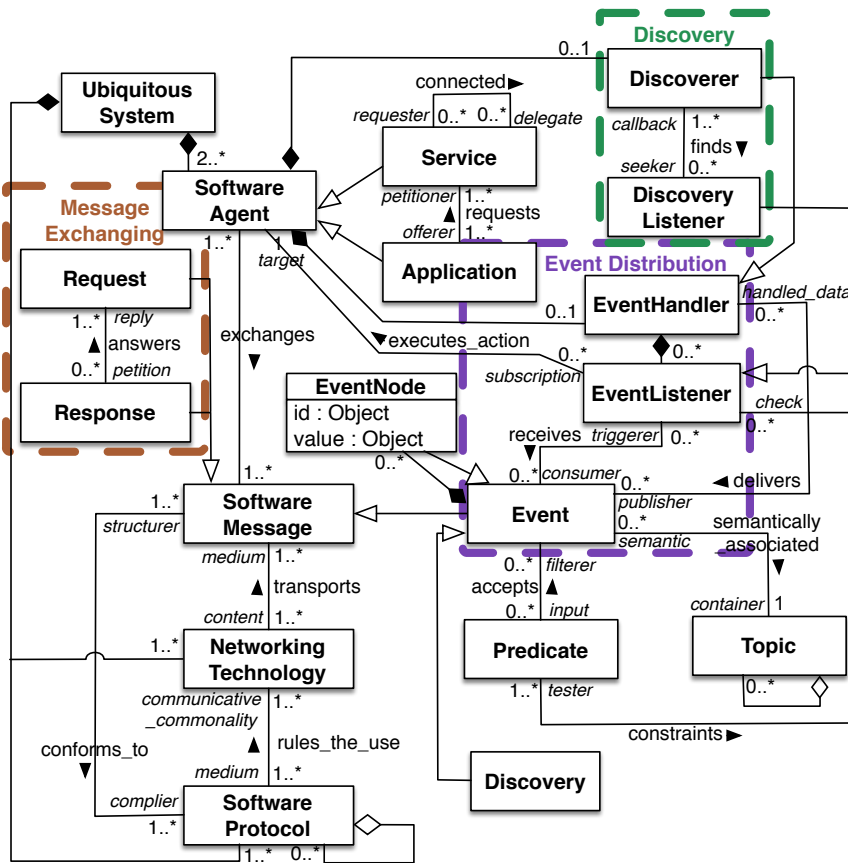


Figure 4.4: A UML class diagram depicting the structural view of the PI-US metamodel

sary to clearly explain the notion of **software agent**, which is present in the structural view. In the CI-CS conceptual model presented in previous chapter the entities that exchange messages in a communication system were referred as *participants*. The reason was to stress the computing independence of that metamodel, since a participant could refer to a software component, an electronic device, a human being, etc. However, in the PI-US metamodel, the only en-

Element	Summarized description
Ubiquitous system	A computation-based communication system that includes mechanisms to support message exchanging, event distribution and dynamic discovery.
Software agent	A participant in a ubiquitous system. Agents can be either applications or services.
Software message	The minimum information unit that is exchanged between the agents of a ubiquitous system.
Networking technology	A technique to manage a physical channel as needed by a computing environment to exchange messages.
Software protocol	A software specification about how to use a networking technology to exchange messages.

Table 4.1: Description of the elements of the PI-US metamodel that are shared between the different communication functionalities supported by a ubiquitous system

tities that communicate are software components that act on behalf of a user, another software or a device, which is the definition of software agent given by Nwana et al. in [83]. Note that in this thesis work, a software agent is merely considered to be an abstraction over any type of software that communicates in a ubiquitous system (i.e., an application or a service), not necessarily to a software with autonomy or proactivity capabilities, as it is considered in other research works. In that sense, the software agents presented herein could be considered as *basic software agents* [34].

Likewise, the notions of software protocol, software message and networking technology are conceptually similar to the notions of protocol, message and channel present in a CI-CS, but they try to

Functionality	Element	Summarized description
Message exchanging	Request	A message to retrieve information from certain agents.
	Response	The message that replies a request.
Event Distribution	Event	A notification of a change in the state of an agent. It is composed by event nodes and have an associated topic . Subsection 4.2.1.1 describes the event model in detail.
	Event Handler	A mechanism to notify events to agents.
	Event Listener	An artifact that receives the events notified by event handlers to do certain actions. A predicate is used to filter the events to be received.
Dynamic Discovery	Discoverer	An event handler that delivers events to discover agents.
	Discovery Listener	An event listener that receives events from discoverers and notifies them about their own availability.

Table 4.2: Description of the elements of the PI-US metamodel, grouped by the communication functionalities that they support

stress the computational orientation of a PI-US, in contrast with the computing independence of a CI-CS.

Software agents can represent an **application** or a **service**. An application is considered to be an **active** software that is controlled by a user to carry out certain tasks involving the interaction between the different elements of a ubiquitous system. Services are mostly considered to be **passive** (or reactive) pieces of software that wait for an interaction from an application to do certain activities, like

Relationship	Summarized description
<i>Exchanges</i>	Software agents (i.e., applications and services) transfer multiple messages between them.
<i>Conforms_to</i>	A message must conform to certain software protocols to be understood by the applications and services present in a ubiquitous system.
<i>Rules_the_use</i>	A software protocol has a set of rules to adequately use a networking technology (e.g., to use it efficiently).
<i>Transports</i>	A channel is in charge of transferring messages from the sending parties to the receiving ones.
<i>Answers</i>	A response message is the reply of a previous request message.
<i>Delivers</i>	Event handlers deliver events to the participants.
<i>Receives</i>	Event listeners receive the events from the event handlers.
<i>Semantically associated</i>	An event is semantically associated to a topic.
<i>Accepts</i>	A predicate is only satisfied by certain events.
<i>Constraints</i>	A predicate constraints the events to be received by an event listener.
<i>Connected</i>	A service can delegate certain actions to other services.
<i>Finds</i>	The discoverer associated to an agent tries to find discovery listeners associated to other agents.

Table 4.3: Description of the relationships between the elements present in the structural view of the PI-US metamodel

providing a piece of information. Services can interact with other services in which they delegate the execution of complex activities. For example, a meteorological service could delegate the measurement of the temperature in a certain physical location to another

service specialized on performing that task. This way, the PI-US metamodel supports the representation of a Service-Oriented Architecture (SOA) in which *compound* services (i.e., the services that need to interact with other services to carry out their actions) offer a common interface to multiple *simple* or *compound* services (i.e., a simple service is able to carry out its actions without interacting with other services). In addition, it is possible to use the PI-US metamodel to specify a client-server architecture, in which applications are clients that consume information from the available services.

Note that, in contrast with the conceptual model presented in previous chapter, in which a unique communication participant self-communicating may conform a communication system, in the specification of a ubiquitous system it is assumed that at least two software agents must be part of the communications, since the presence of less than two software agents is not appropriate to fulfill the interaction level that is expected in this type of systems.

The communication functionality that more usually supports the interaction between software agents in a distributed system is **message exchanging**, which is supported in the proposed metamodel through **request** and **response messages**. Thereby, applications demand certain information through a request message, and a service provides the requested information inside the contents of a response message. In the same way, a compound service could send

a request message to other (simple or compound) services in order to retrieve certain information (i.e., a service could delegate tasks to other services by requesting information to them).

As previously mentioned, event distribution has been assumed to be based on the PubSub paradigm and involves the association of an **event handler** with each software agent in order to *publish* events. Consequently, in the proposed metamodel, there is an optional association between **event handlers** and software agents. The association is optional to allow the metamodel to represent software agents that do not distribute events in any case, without “overloading” them with unnecessary communication functionalities. Thus, the metamodel is flexible enough to represent scenarios in which certain communication functionalities are not present in every software agent (e.g., to improve performance in some ways, for technical reasons, etc.).

Each event handler (associated to each software agent) contains a collection of **event listeners**. An event listener is a software mechanism that includes a **predicate** that is only satisfied by certain events, as a manner to filter the events accepted by each listener (i.e., an event ε is accepted by an event listener λ if ε satisfies the predicate related to λ). Also, an event listener, upon the reception of an event that satisfies its predicate, executes a certain action in a software agent. Hence, the execution of an action is the usual con-

sequence of the reception of an event in which a software agent is interested in. Therefore, event listeners can be considered to be the elements in the proposed PI-US metamodel that represent the **subscriptions** of each software agent, and they provide a mechanism to *bind* the interest of a software agent in certain events to the execution of specific actions. The concept of event listener is also present in some middleware technologies, like ICE, WCF and DDS.

Note that in a CI-CS the notion of predicate was not explicitly presented, but could be implicitly associated to *conditional links* (i.e., in the behavioral view of a CI-CS, presented in Chapter 3, Section 3.2.2, the elements that link two activities upon the satisfaction of a condition or *predicate*). The reason was to keep the CI-CS conceptual model as abstract as possible, without taking into account a concept that could be more related to the actual software (or event mathematical) representation of a condition. However, at the PI-US abstraction level it is appropriate to take into account this notion, since it needs to be taken into account in the design of a ubiquitous system to adequately model constraints or filters associated events.

Events are composed by a set of **event nodes**, which, in turn, are also events. Each event node has a unique identifier and a value in order to represent the contents of an event. Events (and event nodes) are also related to a topic, that is, the semantic of the notification within a certain context (e.g., temperature, humidity, location,

etc.). An insight of the proposed event model and the mechanisms to manage their subscription can be found in the following subsections.

Finally, in the metamodel, the dynamic discovery functionality is a specialization of event distribution, as it was previously assumed when describing the rationale behind the proposal of the PI-US metamodel. For each software agent, there is a specialization of an event handler, known as a **discoverer**, that distributes events each time a new party changes its reachability within a given environment. Specialized listeners (**discovery listeners**) allow software agents to subscribe to discovery events and to execute certain actions whenever other software agents are discovered or become unavailable. Note that a discoverer can only contain discovery listeners, and discovery listeners and discoverers can only manage **discovery events**, even if those constraints are not specified in Figure 4.4 (i.e., in UML, those constraints should be specified through OCL, but this notation has been omitted from the figure in order to simplify it). To conclude, the benefit of reusing the event distribution mechanisms to discover software agents is that the metamodel is simpler to understand and use, in comparison with the possibility of including separated mechanisms to support event distribution and dynamic discovery.

4.2.1.1. A Detailed Explanation of the Proposed Event Model

In the proposed PI-US metamodel, **events** encapsulate information about a change in the state of a software agent. Thus, in this thesis work, an event is defined as a **communication unit that is composed of a set of pieces of information that are related to some topics and can be produced by a software agent as a result of a change in its state**. The adopted event model was present in the PI-US metamodel depicted in Figure 4.4, but an extract of that figure is depicted in Figure 4.5 as a UML class diagram, so as to more clearly illustrate the description of the proposed event model to be presented herein.

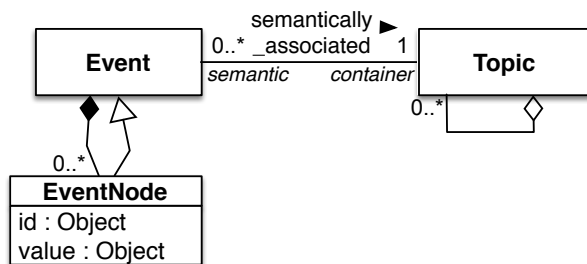


Figure 4.5: An extract of the UML class diagram representing the PI-US metamodel, with a focus on the adopted event model

The pieces of information that are contained into an event are called **event nodes**. An event node contains an identifier (i.e., a unique name) and an associated value. Each event may have an unlimited number of event nodes to represent any piece of informa-

tion. Also, an event node is also an event. This way, hierarchical structures (i.e., trees) can be contained in an event. For example, an event may have a collection of nodes, each of them containing another collection of nodes, and so on.

Each event is related with one topic, which represents its semantics. A topic may be associated with a collection of other topics, so as to represent that a topic is (semantically) the combination of other topics. To avoid confusions, from now on, a topic with no other topics associated will be called a *simple* topic, whereas a topic with other topics associated will be referred as a *compound* topic. The relationships between events and topics are similarly defined in the specifications of the DDS middleware.

If an event has a compound topic, then its event nodes necessarily will have to be related to the topics associated to the compound one. Hence, it is represented that the semantics of each event may be the result of the combination of the topics of its nodes. For example, if a compound *comfort* topic contains *temperature* and *noise* simple topics, then there may exist an event whose topic is *comfort* and whose event nodes are associated to *temperature* and *noise* topics. Conversely, the topic of an event can be inferred as follows:

- If an event is composed of one event node only, its topic will

be the topic of such event node.

- Otherwise, if it is composed of more than one event node, its topic will be one that combines the topics of each event node.

The following subsection offer an insight of the subscriptions mechanisms that are supported in the proposed PI-US meta-model regarding this type of events.

4.2.1.2. Supported Event Subscription Techniques

A subscription σ to an event ε_i is defined as a “filter over a portion of the event content (or the whole of it), expressed through a set of constraints” [28]. Subscriptions are specified through event listeners in the proposed PI-US metamodel. Event listeners support two subscription variants that were previously mentioned in [33]: topic-based and content-based. In this thesis work, these subscription techniques are defined as follows:

- *Topic-Based*. Software agents show interest in a topic and, from that moment, they receive events that are semantically related to that topic. For example, if a software agent subscribes to a *comfort* topic, then it will receive events, not only related to that topic, but also to the *temperature* and *noise* ones (provided that comfort is a compound event linked to temper-

ature and noise ones). Formally, the set of events that are received by a software agent with a topic-based subscription σ_τ is defined as follows:

Let *subscribe* be a function that filters all the events that are received by a software agent on the basis of their associated topic, let T be the set of all semantically formalized topics and τ be a topic, then:

$$\sigma_\tau = \text{subscribe}(\tau), \tau \in T$$

Let E be the set of all possible events and let R be the product set (or the cartesian product) $E \times T$ in which all the relations (ε_i, τ) represent that an event ε_i is semantically related to the topic τ , then \rightsquigarrow is the binary operation “*semantically related to*”, which is defined as follows:

$$\varepsilon_i \rightsquigarrow \tau \iff (\varepsilon_i, \tau) \in R, R = E \times T, \varepsilon_i \in E, \tau \in T$$

Let S_{σ_τ} be the set of all the received events by a software agent whose topic-based subscription is σ_τ , let $\varepsilon_i(k)$ be a function that retrieves the k -th event node of the event ε_i , which is also an event (see previous subsection), and let n_i be the num-

ber of event nodes of ε_i , then:

$$S_{\sigma_\tau} = \{\varepsilon_i | \varepsilon_i \rightsquigarrow \tau\} \cup \{\varepsilon_i | \forall k \in \{1, \dots, n_i\}, \varepsilon_i(k) \rightsquigarrow \tau\}$$

- *Content-Based.* Software agents show interest in receiving events that are semantically associated with a topic when a set of conditions over the event nodes are accomplished. For example, it can be specified a subscription to the *temperature* topic and receive events only when this temperature is over 45°C. The set of events that are received by a software agent with a content-based subscription $\sigma_{(\tau, P_\tau)}$ is formally defined as follows:

Let $\varepsilon_i(k)$ be the k-th event node of the event ε_i , let $t_{i,k}$ be the primitive type of the event node $\varepsilon_i(k)$, let a be a constant of any primitive type and t_a its primitive type, then \dagger is the binary operation “*is comparable to*”, which is defined as:

$$\varepsilon_i(k) \dagger a \iff t_{i,k} = t_a$$

If $\varepsilon_i(k) \dagger a$, five comparing operators can be defined:

$$\varepsilon_i(k) = a; \varepsilon_i(k) < a; \varepsilon_i(k) \leq a; \varepsilon_i(k) > a; \varepsilon_i(k) \geq a$$

These operators always follow a lexicographical order. For

example, $4 < 5$, "aaa" < "aab", $(4, 5, 6, 9) < (4, 6, 3, 1)$, etc. Let s_j be a comparing operation between any $\varepsilon_i(k)$ and a constant a_i , where $\varepsilon_i(k) \dagger a_i$, let τ be a topic and $\varepsilon_i(k) \rightsquigarrow \tau$, then, the predicate P_τ is defined as:

$$P_\tau = s_1 \wedge \dots \wedge s_m$$

Let *contentSubscribe* be a function that filters all the events that are received by a software agent based on a topic τ and a set of constraints described by the predicate P_τ , then:

$$\sigma_{(\tau, P_\tau)} = \text{contentSubscribe}(\tau, P_\tau)$$

Let $S_{\sigma_{(\tau, P_\tau)}}$ be the set of all the received events by a software agent whose content-based subscription is $\sigma_{(\tau, P_\tau)}$, whose constraints are specified by the predicate P_τ , and let $v_{i,k}$ be the value associated with the event node $\varepsilon_i(k)$, then:

$$S_{\sigma_{(\tau, P_\tau)}} = \{\varepsilon_i : \varepsilon_i \in S_{\sigma_\tau}, \{P_\tau \wedge (\varepsilon_i(k) = v_{i,k})\} \vdash \neg \emptyset\}$$

In this formula, $\{P_\tau \wedge (\varepsilon_i(k) = v_{i,k})\} \vdash \neg \emptyset$ means that the predicate P_τ , when in conjunction with the equality $\varepsilon_i(k) = v_{i,k}$ has to be consistent (i.e., it not contains any logical contradictions). For example, if the predicate $P_{\text{temperature}} = \{\varepsilon_1(1) <$

45} and the event ε_1 is published with $\varepsilon_1(1) = 46$, then the subscriber will not receive the event, as $\{\varepsilon_1(1) < 45 \wedge \varepsilon_1(1) = 46\}$ is not a consistent set.

Note that in [33] one additional subscription technique is mentioned: the type-based one. In these subscriptions, each kind of event is directly matched with a type and, therefore, software agents can implement some static checks to ensure that they have received the appropriate information. In the proposed PI-US metamodel type checking subscriptions can be achieved through a topic associated to a unique event. That way, a topic-based subscription can represent a type-based one.

The event handler is the element in the PI-US metamodel that stores every σ subscription, since it has a collection of associated event listeners (i.e., each event listener models a subscription, as it was mentioned before). Moreover, the event handler should provide both *subscribe* and *contentSubscribe* functions in order to be able to dynamically integrate or remove event listeners (i.e., subscriptions) on demand (i.e., to allow the software agents to subscribe or unsubscribe to events during the execution flow of a ubiquitous system). Additionally, an event handler could also provide the function *checkSubs*(ε_i). This function would check whether a software agent is subscribed to the event or not, in order to decide if an event should

be published, which could contribute to improve the performance of the event distribution functionality present in a ubiquitous system. Similar mechanisms are already present in ICE and DDS middleware solutions.

4.2.2. Behavioral View

The behavioral view of the PI-US metamodel includes the concepts and relationships to support the specification of the flow of interactions between the software agents present in a ubiquitous system. Consequently, the foundations presented in the behavioral view of a CI-CS (see Chapter 3, Subsection 3.2.2) have been taken as a basis.

However, ubiquitous systems are computing systems and, therefore, some “abstract” notions present in the behavioral view of the CI-CS metamodel need to be adapted to the software context. For example, the notion of *link*, which represents the connection between two activities, needs to be matched to software-related concepts.

Moreover, in a ubiquitous system, in terms of communications, the only possible activities that can be carried out by the software agents are to exchange messages, to distribute events and to discover other software agents. Nonetheless, event distribution and

message exchanging activities should take into consideration that the software agents are constantly moving, and their availability is not permanent. As so, software agents must discover the other available agent before carrying out these activities.

With all those considerations related to the specific properties that ubiquitous systems have, the behavioral view of the PI-US metamodel has been defined. Figure 4.6 illustrates the behavioral view through a UML class diagram. Additionally, Tables 4.4 and 4.5 respectively provide a brief descriptions of the elements and relationships depicted in the UML class diagram.

Similarly to the behavioral view of the CI-CS metamodel, in the behavioral view of the PI-US metamodel there are some **software agent choreographies**. Each of them represents a possible flow of interactions between the software agents present in a ubiquitous system.

The choreographies are organized in **communication activities**, which involve an exchange of information between one or two agents (either a self interaction or a “remote” interaction). A communication activity comprises a set of **elemental communication activities** between the agent/s present in the activity, and representing a unique information exchange. The idea is to represent that a communication activity involves several *transactions*. Accordingly,

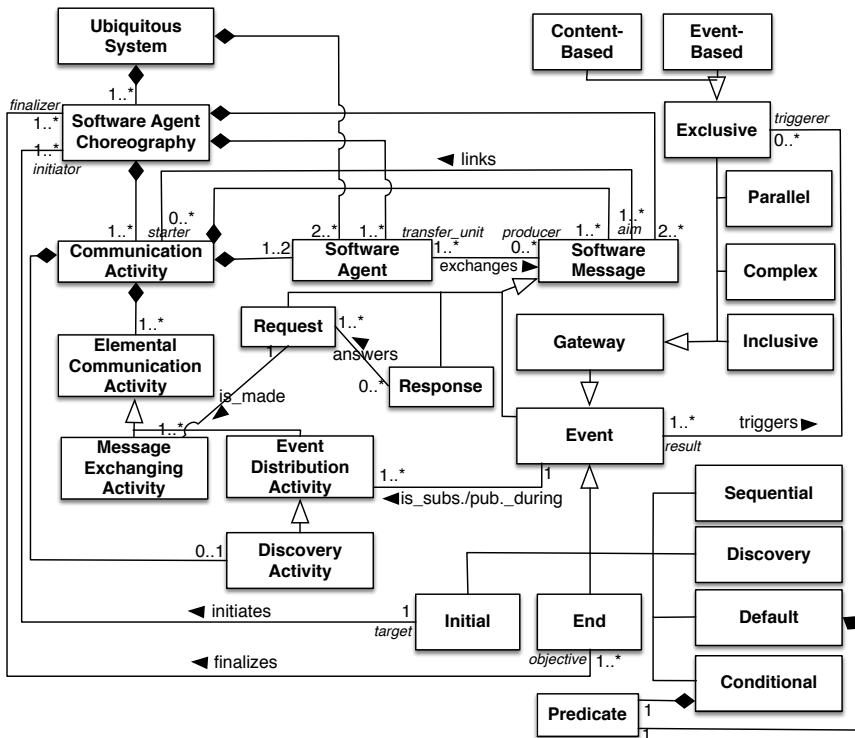


Figure 4.6: A UML class diagram depicting the behavioral view of the PI-US metamodel

these elemental activities can be a **message exchanging** (i.e., to send a request message and to receive its corresponding response), an **event distribution** (i.e., to publish and/or subscribe to one event) or a **dynamic discovery** (i.e., to publish and/or subscribe to a discovery event).

Even if a communication activity may involve different elemental ones, in case that two agents carry out the same activity (i.e., an activity may involve a unique agent exchanging information with

Element	Summarized description
Software Agent Choreography	A flow of interactions between the set of software agents present in a ubiquitous system.
Communication Activity	A set of interactions between one or two software agents. An activity may be composed by multiple elemental activities : exchange two messages (i.e., a request and its corresponding response), distribute an event or discover an entity (i.e., to look for the availability of a certain software agent).
Software Message	The minimum information unit that is exchanged between the agents of a ubiquitous system. It also allows to connect two activities of a choreography (the end of one with the start of another).
Event	An specific type of message that represents a significant occurrence in the communication system at a given moment. An initial event initializes the ubiquitous system and an end event finishes its execution
Gateway	An event that executes or stops one or more communication activities.

Table 4.4: Definition of the elements that are present in the behavioral view of the PI-US metamodel. Some of the elements shared with the structural view are not described again

itself), then at least it is necessary to do an elemental dynamic discovery of software agents consisting of checking the availability of one agent towards the other, and viceversa. The reason is to be able to represent activities that can be only carried out when two specific agents are available in the same context simultaneously. Hence, it is possible to specify a choreography in which the availability of the agents is constantly changing (i.e., they are physically moving,

Relationship	Summarized description
<i>Links</i>	A message connects two activities in a choreography of software agents.
<i>Is_subs./pub._during</i>	The subscription or publication of an event occurs during an event distribution activity.
<i>Is_made</i>	A request is made during a message exchanging activity. Associated to the request, there is a response.
<i>Triggers</i>	An event triggers an exclusive gateway.
<i>Initiates</i>	An <i>initial</i> event starts one or more choreographies in a ubiquitous system.
<i>Finalizes</i>	One or more <i>end</i> events may finalize a choreography (i.e., the execution flow of a ubiquitous system may have different endings, but each of them is associated to an <i>end</i> event).

Table 4.5: Description of the relationships between the elements of the behavioral view of the PI-US metamodel

the networking technologies have restrictions to connect a certain amount of agents, etc.). This aspect of the metamodel contributes to support the volatility of the interactions and the mobility of the agents, which are two key properties of the ubiquitous systems that were highlighted in previous section.

Figure 4.7 depicts a UML sequence diagram that illustrates how two software agents interact during a communication activity, according to the previous PI-US metamodel. The functions outlined in the UML sequence diagram represent the elemental communication activities that they carry out, involving the exchange of messages (requests, responses, events or discovery events).

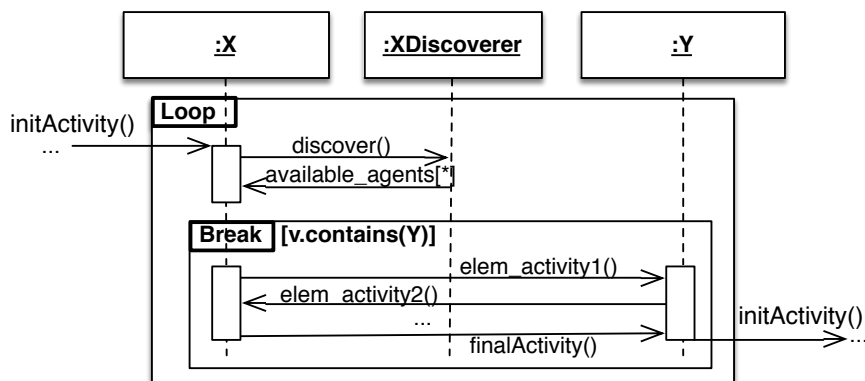


Figure 4.7: A UML sequence diagram representing how software agents interact during a communication activity, as it is assumed in the proposed PI-US metamodel

Communication activities are linked by **messages**. This way, an activity is started by a message that is transferred from a software agent that carries out the initial activity to a software agent present in the target activity. This way, the interactions between activities are homogeneously represented using the same concept that the interactions between agents, that is, the *message*.

Note that in the CI-CS metamodel, a *link* element was part of the behavioral view of the metamodel to represent the connection between activities. The link element was defined to be *something external* to the communication system, but linking activities between them (i.e., a predefined turn to carry out a human conversation, the established flow of a token in a Petri Net, etc.). However, at the PI-US metamodel abstraction level, it is necessary to express the

connection between activities using software-related concepts. In particular, messages are used to represent such connections since the initialization of an activity should involve an interaction between software agents and, conceptually, the message is the information unit that is exchanged during interactions.

Similarly to the CI-CS metamodel proposed in previous chapter, the initialization of a choreography is indicated by the notification of an **initial** event, and its finalization by one or more **end** events. An event could initiate an activity too, since an event is considered in the PI-US metamodel as a specialization of a message (see previous subsection). In the proposed behavioral view of the PI-US metamodel there are three types of events that are specialized into connecting activities: **sequential**, **conditional** and **default**. Sequential events are the ones that are notified to directly relate two activities. Conditional events are only notified when a certain **predicate** is satisfied. Finally, default events are notified when any conditional events were triggered (i.e., any predicates associated to a set of conditional events could not be satisfied).

To conclude, a **gateway** is a specialized event that is delivered to initiate multiple communication activities or to wait for the conclusions of several previous activities to initialize other one. This way, it is possible to modify the usually linear execution flow of a choreography in different ways:

- **Parallel.** Several activities are concurrently initiated.
- **Exclusive.** Given several target activities, only one of them is executed. It allows to represent a split in the execution flow of a choreography. The selection of a specific interaction is based either on the occurrence of an event (**event-based**) or on the delivery of an event with certain contents (**content-based**).
- **Complex.** Similarly to an exclusive gateway, it splits the execution flow of a choreography. However, the selection of the activity to initiate is based on the satisfaction of a predicate.
- **Inclusive.** Several activities are waited to be finished, then another activity is initiated.

4.2.3. Formal Specification as an Ontology

The combination of the different views of a PI-US through the shared concepts results in a conceptualization of the communication mechanisms supporting a ubiquitous system. The conceptualization includes all the notions that are necessary to model a PI-US, and the relationships between those notions. The metamodel has been **formally** defined through an **ontology**, since in previous subsections it has been semi-formally depicted using UML. The formal specification of the metamodel allows to analyze some of its capabilities and quality properties (see next section).

Figure 4.8 shows the graphical representation that Protégé generates for the PI-US ontology. Note that in the graphical representation some elements have been removed in order to avoid the figure to be overcluttered.

Figure 4.9 lists the hierarchy of classes and properties present in the ontology. The whole OWL implementation is provided in Appendix III.

Similarly to the CI-CS ontology proposed in Chapter 3, Section 3.2.3, the ontological representation allows to **share and reuse** the specification of a PI-US between different groups of people or software tools, so as to systematically (and automatically, through the use of **reasoners**) analyze different aspects of a ubiquitous system:

- **Check for inconsistencies** in the specification of a ubiquitous system. For example, it is possible to detect the need of including certain software agents to be able to carry out required interactions, messages to carry out these interactions, software protocols that are able to codify those messages, networking technologies compatible with those protocols, and so on.
- **Optimize** the communications between software agents in a ubiquitous system. To achieve that goal, it could be studied

the use of the different networking technologies and protocols by the different software agents, and to associate the most efficient ones to the software agents that carry out more interactions.

- **Integrate** several ubiquitous systems by including the members of each one to be integrated (i.e., the instances or individuals associated to each class in the ontology, that is, software agents, messages, networking technologies, software protocols, etc.) into a unique ubiquitous system. The resulting system could be optimized and checked for possible inconsistencies, as previously mentioned.

Finally, in the graphical representation of the ontology it is particularly interesting to highlight how the notions related to event distribution appear interlinked with most of the other elements. This aspect of the ontology reflects the great influence that event distribution has in the communication between the software agents of a ubiquitous system. Several authors have previously mentioned that influence [5][28][33], but the ontology of a PI-US makes explicit and formally demonstrates that assessment.

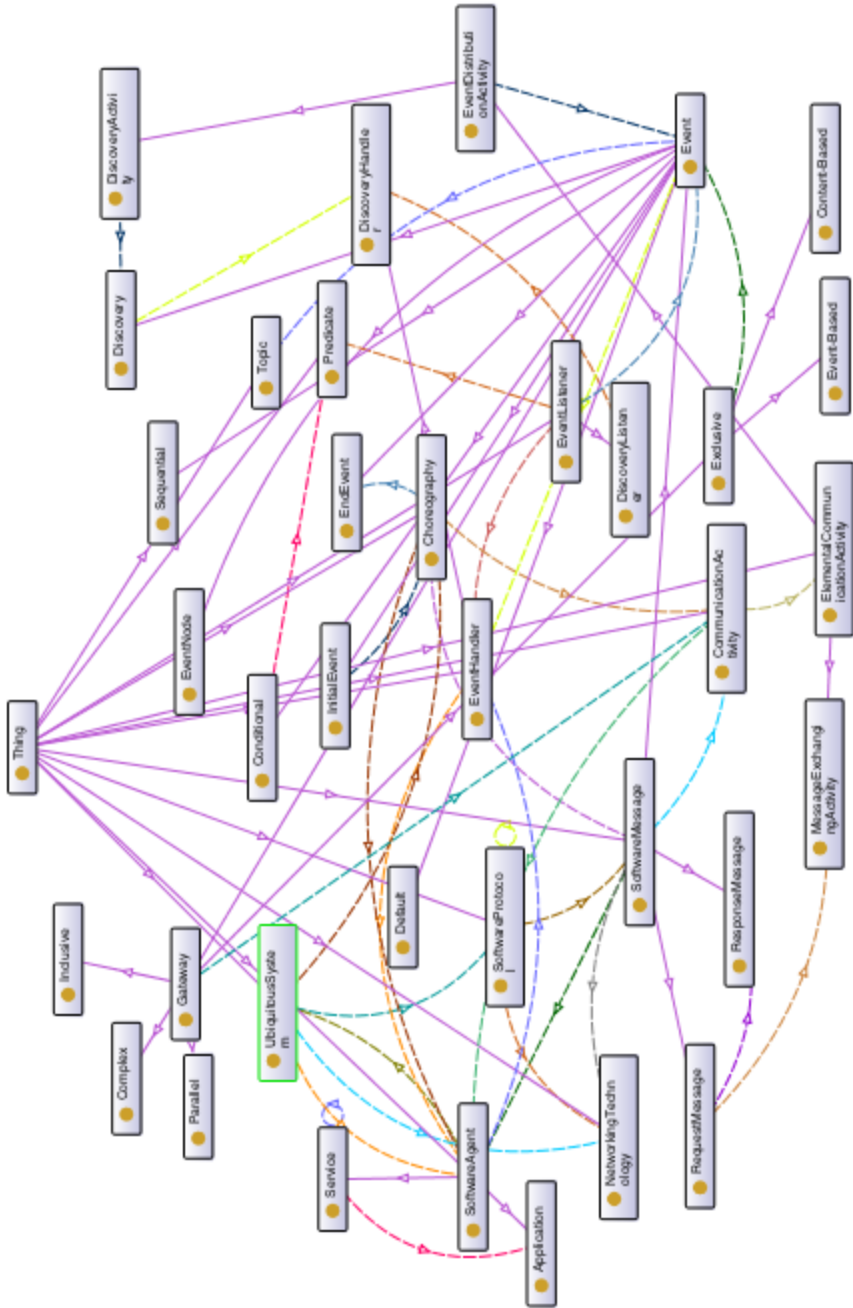


Figure 4.8: An ontology of the communication mechanisms supporting a PI-US



Figure 4.9: The class hierarchy and properties of an ontology of the communication mechanisms supporting a PI-US, as represented in Protégé

4.2.4. Ontological Representation of a Ubiquitous System as a Communication System

As it can be deduced from the description presented all along this section, there is an strong relationship between some of the notions associated to the CI-CS metamodel and some of the notions associated to the PI-US one, even by taking into account their different abstraction levels. The reason is that the proposed metamodel of a PI-US is focused in the software mechanisms, whereas the proposed CI-CS metamodel includes all the concepts that should be present in any communication system. Even so, at the communication system abstraction level, all notions are **computing-independent**, whereas at the ubiquitous system abstraction level, they have to be considered as software mechanisms, but could be treated as **platform-independent**.

Particularly, there are **strong semantic relationships** between Participant - Software Agent, Protocol - Software Protocol, Channel - Networking Technology, Message - Software Message, Choreography - Software Agent Choreography, Choreography Activity - Communication Activity, Sub-Choreography Activity - Elemental Communication Activity and Event (CI-CS level) - Event (PI-US level). In fact, the elements in the PI-US metamodel can be considered as a semantic specialization of the related elements in the

CI-CS metamodel.

This semantic relationship between both abstraction levels has been **formalized** using the ontology of a CI-CS introduced in Chapter 3, Section 3.2.3. The reason is twofold: (1) to formally demonstrate that a ubiquitous system *is a* communication system, and (2) to formally demonstrate that it is possible to trace back the notions present in a CI-CS from the notions present in a PI-US. This is of great importance to demonstrate that it is possible to make a **semantically consistent transformation** from a CI-CS into a PI-US, that is, the elements in the target model keep the semantics of the elements in the initial, more abstract, model. In addition, it demonstrates that there is a possible reverse transformation between a PI-US to a CI-CS (i.e., a transformation from a more specific model to a more general or abstract one), which is relevant to propose possible future **model-based reverse engineering methodologies**, like OMG standard **Architecture-Driven Modernization (ADM)** (<http://adm.omg.org>), to help into analyzing, validating and improving existing ubiquitous systems.

Figure 4.10 illustrates the hierarchy of ontological classes that have been proposed on the basis of the ontology of a CI-CS, and the **automatically inferred** hierarchy using a reasoner. Note that the initial hierarchy of classes specifies that the notion of ubiquitous system is **unrelated** to the notion of communication system.

Also, the previously identified semantic relationships between a CI-CS and a PI-US have been represented in the ontology of a CI-CS. For example, a *SoftwareAgent* has been represented as a subclass of a *Participant*, a *NetworkingTechnology* as a subclass of a *Channel*, and so forth. Moreover, these subclasses have been constrained to be associated to the other software-related subclasses. For instance, a *NetworkingTechnology* has been declared as a subclass of a *Channel* constrained to be *a medium of a SoftwareProtocol*. To conclude, a ubiquitous system has been defined as a semantic class associated to a non-empty set of networking technologies, software protocols and, at least, two software agents, as specified in the PI-US metamodel.

On the basis of all those constraints and semantic classes added to the CI-CS ontology, **the reasoner is able to automatically deduce that a ubiquitous system is a communication system** (i.e., the *UbiquitousSystem* class is a subclass of the *CommunicationSystem* one). However, it is important to clear up that a communication system can not be considered as a ubiquitous system *per se*, but a ubiquitous system can be considered as a software-based communication system that fulfills the specific requirements and quality properties that were defined in previous section.

The reason of this “mismatch” is that the CI-CS metamodel has less constraints related to the cardinality of some elements due

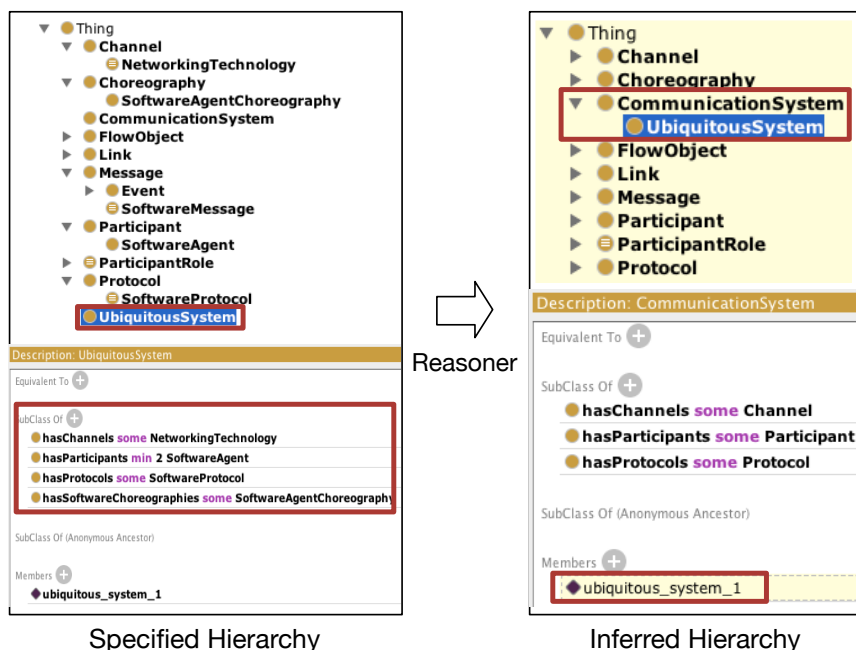


Figure 4.10: Some screenshots of Protégé that show how a reasoner can automatically infer that a ubiquitous system is a communication system

to its more abstract level (e.g., there can be zero participants in a CI-CS, but there must be at least two software agents in a PI-US). This is very positive, since it allows to check for inconsistent ubiquitous systems' design when transforming from CI-CS (see next chapter), that is, to check if a CI-CS can be implemented by software means as a platform-independent ubiquitous system or not (i.e., if the design of a CI-CS should be modified to be able to implement it as a platform-independent ubiquitous system).

4.3. Quality Attributes of the Communication Model for Ubiquitous Systems

As it was explained in Chapter 3, Section 3.3, an adaptation of the international standard **ISO/IEC 25010:2011** [54] to the meta-modeling field has been considered as the framework to describe some quality attributes of the CI-CS metamodel.

Likewise, that framework has been used to analyze the **quality attributes** of the PI-US metamodel. Table 4.6 summarizes the analyzed quality attributes and sub-characteristics of the PI-US metamodel. Some of the attributes and sub-characteristics have been assessed through the ontological representation of a PI-US. Appendix IV describes how the proposed metamodel fulfills the multiple attributes and sub-characteristics. Also, a definition of the sub-characteristics is provided in that appendix.

Each quality attribute and sub-characteristic is defined again (they were previously provided in Chapter 3, Section 3.3, and in Appendix II) in order to improve the legibility and understanding of each given qualitative description.

Attribute	Sub-Characteristic	Fulfilled
Functional suitability	Appropriateness	✓
	Accuracy	✓
	Interoperability	✓
	Compliance	✓
Reliability	-	Partially
Efficiency	-	✓
Operability	Recognizability	✓
	Learnability	Partially
	Helpfulness	✓
	Attractiveness	✗
Compatibility	-	✓
Maintainability	Modularity	✓
	Reusability	✓
	Analyzability	✓
	Changeability	✗
	Testability	Partially
Transferability	Adaptability	Partially
	Portability	Partially

Table 4.6: Quality attributes of the PI-US metamodel

4.4. Conclusions

This chapter has described a generic communication model for Platform-Independent Ubiquitous Systems (PI-US), which has been presented as a metamodel. This metamodel has two different views: structural and behavioral view.

The structural view captures all the structural and basic operational software mechanisms that should be present in a ubiquitous system to support message exchanging, event distribution and dynamic discovery. To devise the metamodel, some communication

standards and well-known middleware technologies have been analyzed.

The behavioral view includes elements to represent the organization of the communication in a ubiquitous system through a choreography of communication activities involving previous communication functionalities and software mechanisms. The behavioral view of the PI-US metamodel is based on the behavioral view of a CI-CS presented in previous chapter. Nonetheless, all the elements in the behavioral view are related to specific software mechanisms, as it is necessary in order to represent a ubiquitous system.

Note that the metamodel only includes the key notions that are shared between different communication standards and well-known middleware specifications, which makes it possible to offer a more abstract and less technically oriented perspective of the communication mechanisms present in the ubiquitous system. This aspect of the metamodel also contributes to re-affirm its platform independence, since it does not include specific elements to support any specific platforms.

The metamodel has been represented as an ontology, which formally conceptualizes the communication aspects of a ubiquitous system. The ontology can be applied to automatically check through a reasoner if a model of a ubiquitous system is able to accomplish

with the expected communication capabilities associated to these systems (i.e., if a ubiquitous system design is consistent with the proposed conceptualization of the communication mechanisms associated to a ubiquitous system). Additionally, it could be used to integrate different ubiquitous systems into a unique one, by including all their respective structural and operational elements into a common choreography of communication activities. The consistency of the integration could be semantically checked through a reasoner, in order to validate that the resulting design conforms to the PI-US metamodel proposed herein, and to minimize or optimize this integration by establishing equivalencies between the elements present in the different ubiquitous systems to be integrated.

Furthermore, the ontology of a CI-CS has been extended by incorporating some key notions present in the PI-US metamodel. This semantic extension has allowed to formally demonstrate that a ubiquitous system can be considered as a software communication system with specific requirements and supporting concrete communication functionalities. Moreover, it has been formally demonstrated that it is possible to trace back certain notions present in a PI-US to the more abstract level of a CI-CS. This is of great relevance for the future proposal of reverse engineering methods for ubiquitous systems intended to help into analyzing, validating and improving existing ubiquitous systems.

Similarly to the analysis of the quality properties of the CI-CS metamodel, the ISO/IEC 25010:2011 standard has been applied to the study of the quality of the proposed PI-US metamodel with the assistance of its ontological representation.

Finally, as it was mentioned in Section 4.1, the metamodel presented in this chapter is focused on the **communication aspects** of these systems, due to the orientation of this thesis work. The conceptualization of the other aspects of a ubiquitous system (user interfaces, information systems, etc.) could contribute to a complete formalized conception of a ubiquitous system, which could help to tackle with the design of this type of systems in a holistic way. However, this thesis is not focused on these other facets of a ubiquitous system.

Chapter 5

MUSYC: An MDA-based Methodology to Develop Ubiquitous Systems on the Basis of the Communications

With the foundation of the metamodels presented in previous chapters, an **MDA-based Methodology to Develop Ubiquitous Systems on the Basis of the Communications (MUSYC)** is proposed. The objective is to approach the development of the ubiquitous systems on the basis of the specification of a CI-CS, its systematic transformation to the design of a PI-US and, ultimately, the code generation to support the implementation of a ubiquitous system.

The direct consequence is that **the use and adoption of specific technologies is delayed until the final stages of the development**, rather than guiding the whole development process from the

beginning, as it commonly occurs in code-centric developments¹. Consequently, the methodology pursues a development process that is **more focused on fulfilling the needs of the user**, rather than on making an appropriate use of concrete technologies. Additionally, MUSYC allows the specification of **more perduring designs** that do not tackle with today's technologies only, but that can also be reused when future technologies become available.

Finally, to assist in the development of ubiquitous systems through MUSYC, a set of **CASE tools** have been implemented. This implementations also demonstrate the feasibility of **automatizing and validating** the transformation from the conceptual specification of a computation-independent communication system to programming code that can be executed on top of target computing platforms (operating systems, middleware, etc.).

5.1. Overview

Previous chapters have dealt with the conceptual modeling of a CI-CS and a PI-US. The metamodels presented in both chapters serve as a basis to carry out an MDA-based development of communication schemes for ubiquitous systems (i.e., the structural and operation elements, and their organization). According to the pro-

¹The disadvantages of code-centric developments in comparison with model-driven ones were explored in Chapter 2, Subsection 2.3.2

cess depicted in Figure 5.1, the MDA approach should be based on the **transformation** between the following models:

- **Communication System CIM (CS-CIM).** A computation independent model of the concepts and relations that conceptualize a communication system, without taking into account any technical details related to its software implementation. The metamodel is equivalent to the conceptual model of a CI-CS presented in Chapter 3.
- **Communication System for Ubiquitous Computing PIM (US-PIM).** A communication model for ubiquitous systems that considers the software mechanisms that should be part of the final implementation of the ubiquitous system, but it does not take into consideration any specific target computing platforms (e.g., programming languages, operating systems, middleware solutions, etc.). The PI-US metamodel was presented in Chapter 4.
- **Communication System for Ubiquitous Computing PSM (US-PSM).** A communication model of a ubiquitous system supported by specific computing platforms. At this abstraction level, it is possible to define the specific technologies that the communications should use, and to establish how they can be technically integrated.

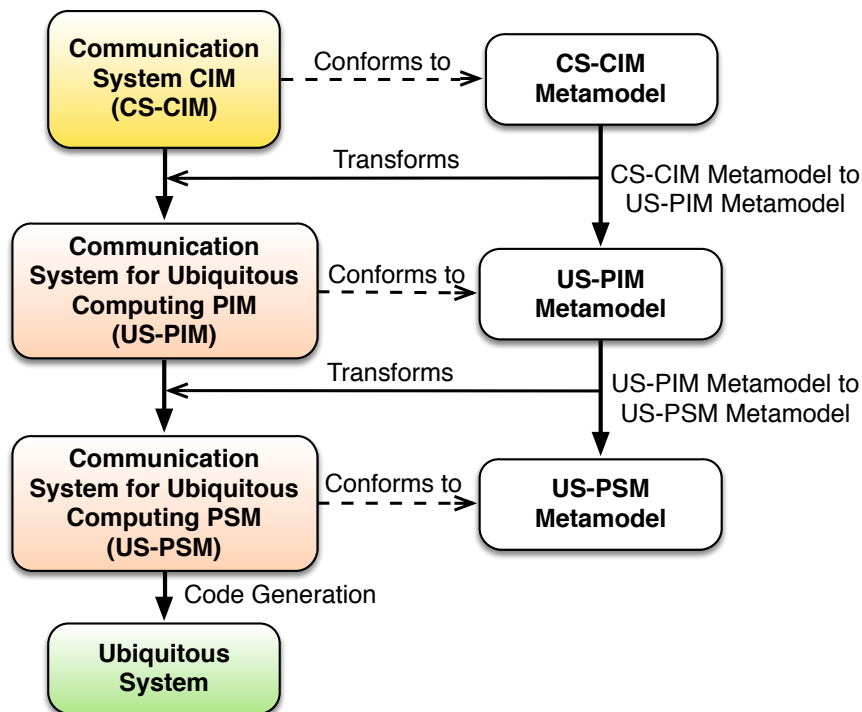


Figure 5.1: MDA approach to the development of ubiquitous systems

As specified in the MDA standard, the models can be transformed from higher to lower abstraction levels (i.e., from CIM to PIM and from PIM to PSM) on the basis of **transformation rules applied to their underlying metamodels**, like it is also illustrated in Figure 5.1.

MUSYC systematically approaches the metamodel-based model transformation defined in MDA through a set of **proposed transformation rules** to be described in the following sections. Furthermore, it also includes additional stages to be able to correctly

specify the CS-CIM, US-PIM and US-PSM models. The rationale is that the development of a ubiquitous system does not only involve model transformation, but **to make appropriate model definitions** too (CS-CIM, US-PIM and US-PSM, that is, instances of the corresponding metamodels) in order to fulfill the requirements that need to be accomplished in the ubiquitous system to be developed.

Consequently, the general idea pursued by MUSYC is to **allow the design of complex ubiquitous systems without dealing since the initial development stages with the specific technical issues that need to be usually taken into consideration** (e.g., choosing specific programming languages, networking technologies, software protocols, middleware solutions, etc.). From an abstract CS-CIM, by applying a set of proposed transformation rules, it is possible to systematically derive a US-PIM that incorporates some notions oriented towards supporting the specific communication requirements of a ubiquitous system (e.g., mobility, volatility, etc.). Finally, the US-PIM can also be transformed into a US-PSM, which includes specific technical details that are directly related to the implementation of the **communication components** that would be part of the ubiquitous system itself (i.e., the communication components are considered to be all the software mechanisms that are involved in the information communication in a ubiquitous system).

A simplified description of the development process defined by MUSYC is illustrated in Figure 5.2 using a Software & Systems Process Engineering Meta-Model 2.0 (SPEM 2.0) diagram. SPEM (<http://www.omg.org/spec/SPEM/2.0/>) is a standard adopted by OMG for the definition of software development processes. The detailed development process is depicted in Appendix V, also as an SPEM 2.0 diagram.

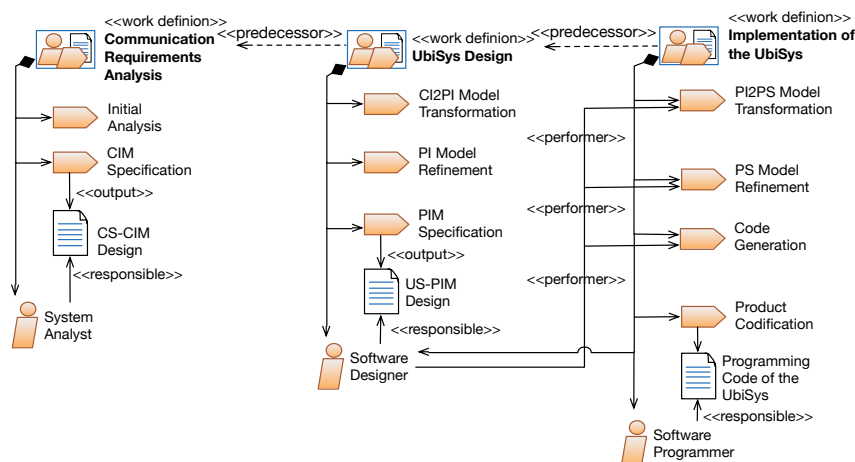


Figure 5.2: A simplified description of the development process proposed in MUSYC, depicted as an SPEM 2.0 diagram

In order to improve the legibility of this chapter, a diagram about the overall development process specified in MUSYC is also depicted in Figure 5.3.

As it is depicted in those figures, MUSYC includes three main stages:

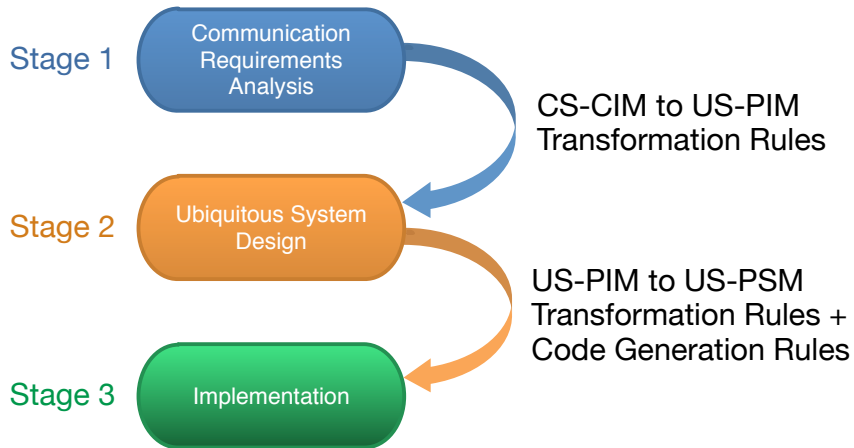


Figure 5.3: Overall development process specified in MUSYC

- **Communication Requirements Analysis:** On the basis of the needs of the stakeholders, a system analyst makes an initial analysis of the communication requirements using the CS-CIM metamodel, also specifying a brief use case model and a choreography model represented through a BPMN 2.0 choreography. The use case and choreography models are later used to propose the design of a CS-CIM.
- **Ubiquitous System Design:** In this stage, a software designer incorporates to the design the specific communication requirements that should be taken into account in a ubiquitous system. To do so, the CS-CIM is transformed into an initial US-PIM, represented through UML class and sequence diagrams. The transformation consists on applying a set of transformation rules to the CS-CIM and US-PIM metamodels, as it is

specified in MDA. The resulting US-PIM is refined to produce a final one.

- **Implementation of the Ubiquitous System:** A software designer transforms the US-PIM into a US-PSM by adopting several concrete computing platforms (target operating systems, programming languages, specific communication technologies, etc.), and applying a set of transformation rules to the US-PIM metamodel. The resulting US-PSM is refined, and a code template is generated on its basis. To conclude, the code template is completed by a software programmer in order to produce the final implementation of the communication components of a ubiquitous system.

It is particularly important to mention that most of the development tasks specified in the MUSYC methodology are carried out by **software analysts and designers**, as it is possible to observe in previous figure and summarized descriptions of the different development stages. Specifically, software programmers are only present in the development process to produce the final implementation of the communication components for a ubiquitous system. This is distinct from the usual way of approaching the development of a ubiquitous system, in which their communication aspects are commonly considered to be technical issues that should be approached

during the implementation stage. Therefore, **MUSYC shifts the focus of the specification of the communication components of a ubiquitous system from the implementation to the design stages.**

The main benefit is that the communication aspects of a ubiquitous system can be approached during the design of a ubiquitous system, taking into account the links that would probably exist between communication components and user presentation, data model, etc. Moreover, MUSYC encourages a more close association between the **requirements of a stakeholder** and the final communication functionalities to be supported in the ubiquitous system to be developed.

The following sections provide a description of the different stages that conform the development of the communication components of a ubiquitous system, as specified in the MUSYC methodology. In order to ease the understanding of the different development stages associated to MUSYC, an example of a Ubiquitous Medical Environment (UME) is used along the rest of this chapter.

In the UME sample example, the doctors monitor the biometric signals of the patients through several everyday objects in which multiple sensors are placed (e.g., a couch controls the patients' weight, the cookware controls their blood pressure or sugar level, etc.). If the level of any of the monitored biometric signals

may be critical for a patient (e.g., the blood pressure is too high), then an alert is sent to the doctors. Afterwards, the doctors may request additional information to the patient in order to make a medical decision related to his/her status (e.g., to send an ambulance, go to visit the patient, detect a false alarm, etc.). If the doctors decide that the alert is really related to a health problem, then they can relate a new medical record to the patient. Finally, if the doctors consider the UME to be dangerous or unlikeable for the patient (e.g., the biometric sensors do not work well, the patient is stressed by his/her constant monitoring, etc.) they can remotely stop its execution.

5.2. Stage 1: Communication Requirements Analysis

At this stage of MUSYC, a system analyst makes an initial analysis of the communication requirements of the system to be developed, on the basis of the requirements of the stakeholders. To do so, firstly, the analyst has to define a brief use case model and a choreography model. The use case model will represent the main functionalities and participants involved in the system. The choreography model will define an organized set of interactions in which those participants exchange some messages to carry out the expected functionalities. Moreover, some additional participants can be included in the use case and choreography models to take into account

external systems or hardware devices that need to be present in the ubiquitous system to be developed (e.g., sensors, actuators, external storage services, etc.).

On the basis of the use case, choreography models and the CS-CIM metamodel (see Chapter 3), a CS-CIM can be specified. The CS-CIM will be able to represent the key structural, operational and organizational elements that should be present in the design of a communication environment that fulfills the requirements of the system to be developed.

Figure 5.4 depicts the overall description of the first development stage of MUSYC. The following subsections will provide a description of the initial analysis (i.e., the specification of the use cases and choreography models) and the CS-CIM specification tasks defined in MUSYC.

5.2.1. Initial Analysis through Use Cases and Choreography Models

The initial analysis of the communication requirements of a system involves the definition of a **use case model**. This model should be able to capture the main actors to be taken into consideration in a system, the interactions between them and the most general tasks that they carry out, which can be directly traced to the re-

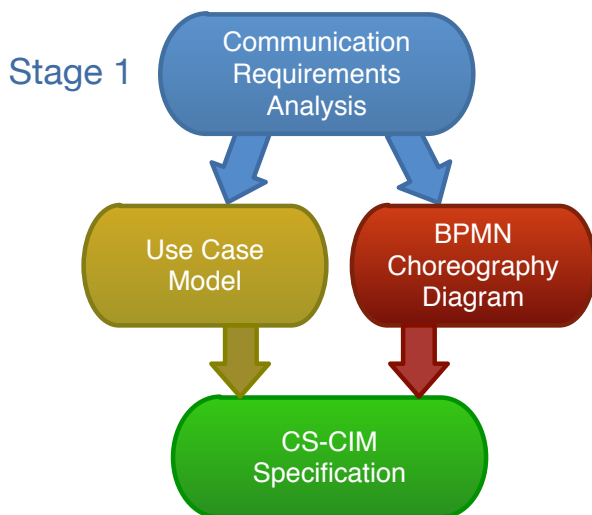


Figure 5.4: First development stage specified in MUSYC

quirements provided by the stakeholders. For example, to develop a UME, it should be defined a use case model including the most general tasks to be accomplished by doctors and patients, taking into account the specific requirements of the ubiquitous system to be developed, like the use of sensors to monitor the biometric signals or the need to interact with an external medical information system to store the medical records of the patients. The sample use case model is depicted in Figure 5.5 using UML.

In MUSYC, a **choreography model** represents the organization of the interaction processes that are carried out by the different actors during the tasks specified in the use case model. The choreography model can be presented as a set of **choreographies**, and

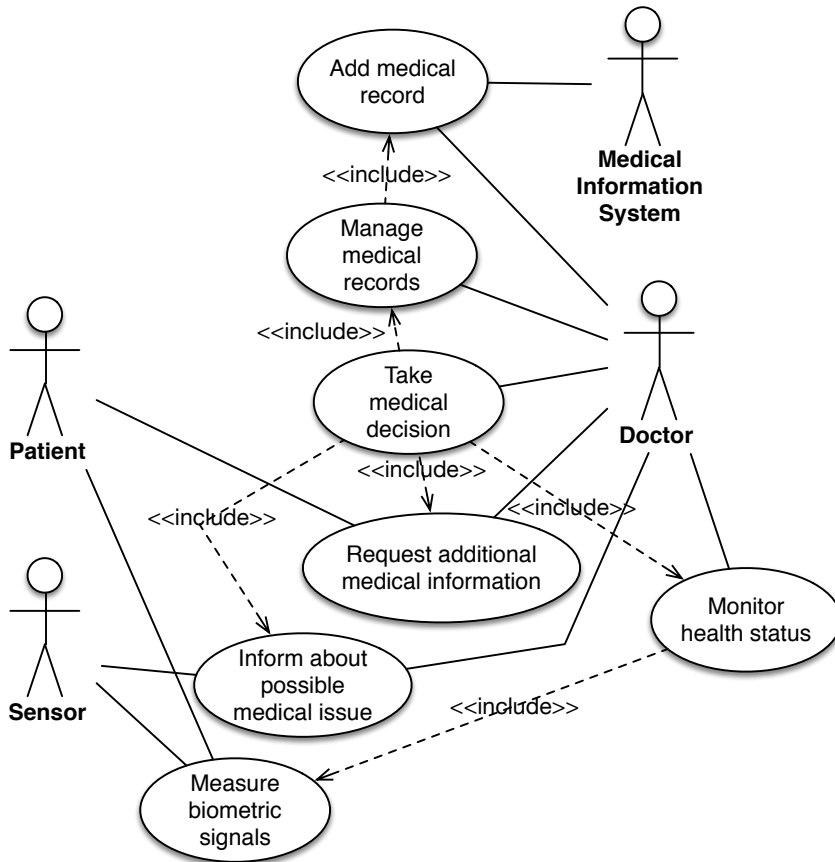


Figure 5.5: A sample UML use case model that could be specified during the initial analysis of a Ubiquitous Medical Environment

depicted through BPMN 2.0 Choreography diagrams.

For example, on the basis of the use case model defined for the UME, a choreography model for the UME example has been defined. It has been depicted using a BPMN 2.0 Choreography diagram in Figure 5.6, as it is defined in MUSYC.

To sum up, it is important to highlight the benefits of us-

ing both use case and choreography models, like it is promoted in MUSYC. The use case model provides a helpful representation of *what* is needed by the stakeholders. Therefore, it closely represents the key functional requirements that the system should accomplish. Consequently, it is a model that should always be present during the whole development process, and constantly reviewed in order to check that a provided system design really accomplishes with all the stakeholders' needs. On the other hand, the choreography model represents the interactions that need to be carried out by the different actors (or participants) during the tasks that were defined in the use case model. Hence, it is an important abstraction that gives a general and initial idea about *how* the requirements of the stakeholders can be fulfilled with a concrete software design.

5.2.2. CS-CIM Specification

On the basis of the defined choreography it is possible to design a model that conforms with the CS-CIM metamodel (i.e., an *instance* of the metamodel) and supports the functional requirements represented in the use cases. The separated instantiation of the elements present in the structural and behavioral views of the CS-CIM metamodel assists in that instantiation task, as it is described in the following subsections.

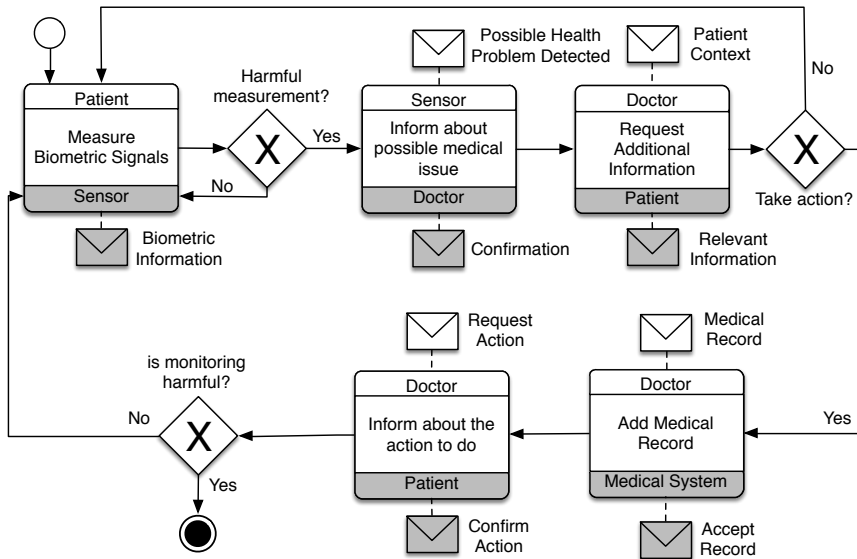


Figure 5.6: A sample choreography specified during the initial analysis of a Ubiquitous Medical Environment, depicted as a BPMN 2.0 Choreography diagram

5.2.2.1. A Method to Instantiate the Elements of the Structural View

The **structural view of the CS-CIM metamodel** (see Chapter 3, Section 3.2.1) assists into focusing on understanding and designing the structural and basic operational elements that should be present in the system to be developed. To instantiate the elements of the structural view, the required structural elements need to be identified with the assistance of the use case and choreography models as follows (and in the order that has been depicted in Figure 5.7 through a UML activity diagram):

- **Participants:** The participants are equivalent to the reflected ones in the choreography model.
- **Channels:** At least, a common channel must be established for the participants. If it is needed to interact with external systems, then additional channels should be specified to allow those interactions. If some participants can not share the same channel, then it is necessary to define additional participants that can make use of some of those heterogenous channels at the same time. Those participants will be referred as *mappers*.
- **Messages:** The interactions that have been specified in the choreography model make it possible to identify the messages to exchange between the participants of the system to be developed. If a channel, for any reason, does not allow to transfer a message, then another channel needs to be established.
- **Protocols:** For each channel, a protocol to manage it according to the requirements of the system has to be adopted. If a suitable protocol for a particular channel can not be designed or adopted for any technical reasons, then it will be necessary to replace the channel. Additionally, the messages to be transferred need to be formatted according to the specifications of each adopted protocol. If a protocol does not support the codification of a message, then another protocol needs to

be chosen.

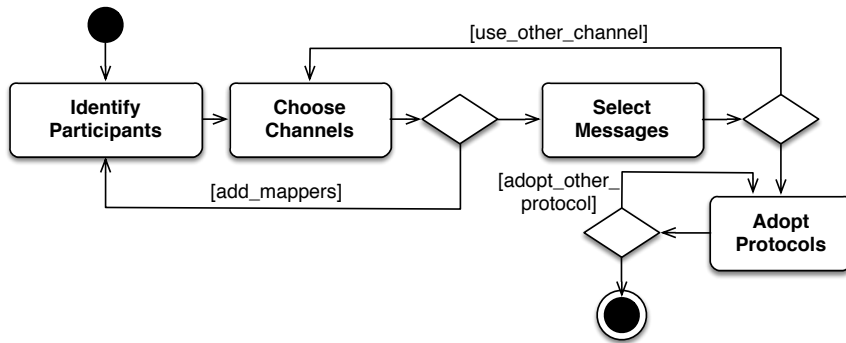


Figure 5.7: A UML activity diagram that specifies the process that should be followed to identify the elements present in the structural view of a CS-CIM, as it has been defined in MUSYC

As it is possible to observe in the previous activity diagram and descriptions of the different identification processes, the adoption of the appropriate structural elements of a CS-CIM involves several **iterations** in order to obtain a design that can be both feasible and accomplishes the requirements established in the initial analysis of the system to be developed.

As an example of the result of the identification process and the subsequent instantiation of the elements present in the structural view of the CS-CIM metamodel, Figure 5.8 depicts a model of a UME that conforms with the structural view of the CS-CIM metamodel.

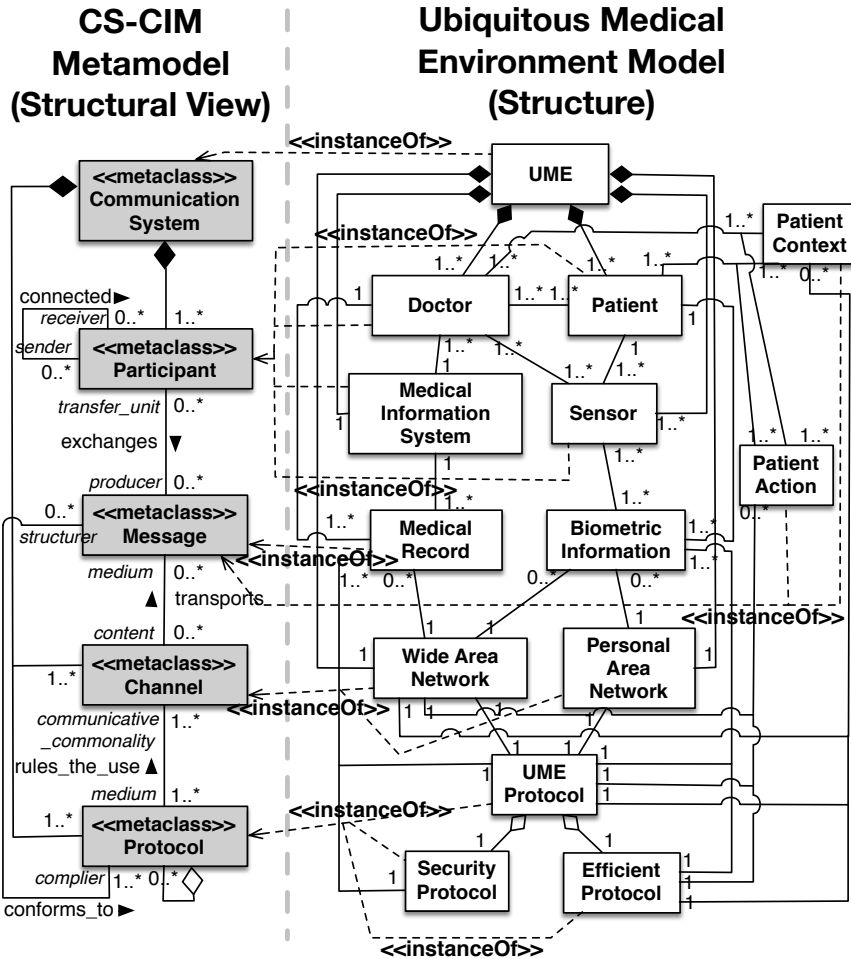


Figure 5.8: A UML class diagram depicting the structural elements of the sample UME as an instance of the elements present in the structural view of the CS-CIM metamodel

To devise the depicted model, the previously described identification process has been applied. Firstly, the participants have been directly identified using the choreography model. Then, specific channels have been adopted to be able to connect the sensors

with the patients, the patients with the doctors and the doctors with the external medical information system.

Since the sensors can be physically attached to the patient, a personal area network (e.g., BlueTooth, Infrared, etc.) has been supposed to be an appropriate communication channel, because they should be more suitable for low-range connections (e.g., lower power consumption, they do not need a fixed infrastructure, etc.). However, to connect the patient with the doctor, and the doctor with the external medical information system, it could be more appropriate to use a wide area network, like Internet (e.g., to have long-range connections, more bandwidth, etc.).

The messages to be transferred by the participants can be obtained from the choreography model (see Figure 5.6). In this case, four types messages have been identified: *Biometric information* (measure biometric signals and inform about possible medical issue activities), *medical record* (add medical record activity), *patient context* (request additional information activity) and *patient action* (inform about the action to do activity). The *biometric information* messages are sent by the sensors to the patients. The patients send these messages to the doctors in order to notify them about a possible health problem. Medical records are transferred by the doctors to the medical information system. The *patient context* messages are exchanged to inform about the circumstances in which the patients

are when a health issue is detected. Finally, the *patient action* messages contain information about the different actions that a patient may carry out when a health problem is diagnosed by a doctor.

It is worth to be mentioned that the sample instantiation of the CS-CIM metamodel to design a UME has been simplified. For instance, in the choreography model more messages are represented, but they all refer to confirmations or responses to the previously mentioned messages. In consequence, they have been removed from the devised model. In real situations they should be taken into consideration to instantiate the structural view of the CS-CIM metamodel. Furthermore, it has been assumed that the medical information system is able to communicate through a wide area network with the doctors. As it was previously noted, in practice, it could be possible to establish concrete channels to communicate with any external systems. Moreover, it is assumed that a sensor is able to communicate using both a wide area network and a personal area one. As mentioned earlier, if this was not the case, then it should be necessary to add a *mapper* participant, which, in the UME example, it would be in charge of collecting the messages transferred by the sensor through the personal area network and transferring them through the wide area network.

Additionally, it needs to be highlighted that, in MUSYC, at these initial development stages, no specific technologies or plat-

forms should be presented in the instantiated CS-CIM metamodel. For example, the UME model refers to an ad-hoc network, but it does not refer to BlueTooth, infrared, etc. The reason is to keep the devised model as abstract as possible by avoiding computation or platform dependent mechanisms as much as possible.

5.2.2.2. Behavioural View Model by Choreography Model Instantiation

The behavioral view of a CS-CIM can be directly instantiated using the choreography model, since the behavioral view shares concepts with the metamodel of the choreographies in BPMN 2.0 (see Chapter 3, Subsection 3.2.2). The direct benefit is that a BPMN 2.0 choreography can be directly used to design the behavioral view of a CS-CIM. Moreover, it is possible to match existing BPMN choreography diagrams to the concepts present in the behavioral view, as a manner to obtain a conceptualization of an previously existing communication system from its graphical representation as BPMN choreographies.

However, as it was previously mentioned in Chapter 3, Subsection 3.2.2, the behavioral view of the CS-CIM metamodel is not exactly as the BPMN 2.0 choreography metamodel, since some elements related to the graphical notion have been removed, and the name of some others has been changed to avoid confusions. For

example, the term *connection* in the BPMN metamodel has been replaced by the term *link* in the CS-CIM metamodel, since the notion of *connection* is commonly used in the communication field to refer to the initial interaction between two participants, whereas, in this case, it refers to the relationship between two activities in a choreography.

To carry out the direct matching between a BPMN 2.0 choreography and the CS-CIM metamodel, Tables 5.1, 5.2 and 5.3 are provided. In those tables, it is related each possible graphical element in a BPMN 2.0 choreography with an element in the CS-CIM metamodel.

Note that the matching of the BPMN graphical notation of an event is simplified in previous tables. In BPMN there are multiple types of events: errors, message arrival notifications, timeouts, etc. Those events need to be matched to different instances of the event metaclass, that is, to define specialized events in the CS-CIM for each concrete event defined in the BPMN choreography. Nevertheless, the initial and end events can be directly matched with elements in the metamodel, due to their significant importance when defining any choreography (i.e., they start and finish the choreography itself).

Also, the BPMN graphical notation allows choreographies that reference other choreographies to be represented. In this

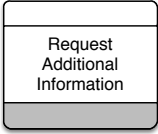
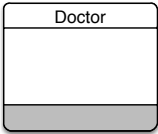



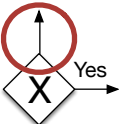
BPMN Notation	Behavioral View Element
	Choreography Activity
	Participant
	Message
	Sequential Link
	Conditional Link
	Default Link

Table 5.1: Description of the matchings between a BPMN 2.0 Choreography and the concepts in the behavioral view of the CS-CIM metamodel

case, the direct matching involves the definition of these kinds of choreographies as a whole, that is, by including all the elements of the choreography into the same diagram.

5.3. Stage 2: Ubiquitous System Design

The second development stage that has been established in MUSYC, which is depicted in Figure 5.9, is to propose a platform-






BPMN Notation	Behavioral View Element
	Parallel Gateway
	Inclusive Gateway
	Complex Gateway
	Content-based Exclusive Gateway
	Event-based Exclusive Gateway

Table 5.2: Description of the matchings between the gateways of a BPMN 2.0 Choreography and the concepts of the behavioral view of the CS-CIM metamodel




BPMN Notation	Behavioral View Element
	Initial Event
	End Event
	Event

Table 5.3: Description of the matchings between the events of a BPMN 2.0 Choreography and the concepts of the behavioral view of the CS-CIM metamodel

independent design of a ubiquitous system that fulfills the requirements that were detected and analyzed in the previous stage. To achieve that goal in a methodological way, a **transformation from a CS-CIM into a US-PIM** is proposed.

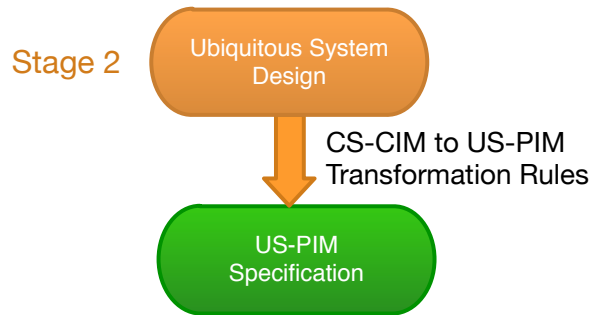


Figure 5.9: Second development stage specified in MUSYC

As it will be explained in the following subsections, the transformation process can be systematically done through the proposal of a set of transformation rules applied to the corresponding meta-models, as it is defined in the MDA standard. To improve the comprehension of the transformation process, the proposed rules will be separately described in the following subsections for the structural and behavioral views of the CS-CIM and US-PIM metamodels. The transformation rules, which have been implemented in QVT as it is specified in the MDA standard, are fully listed in Appendix VI.

5.3.1. CS-CIM to US-PIM Transformation: Structural View

The systematic application of a set of QVT transformation rules to the elements in the structural view of the CS-CIM metamodel can assist software designers during the specification of the structural elements of a US-PIM. The idea is to **systematically ob-**

tain a design of the structural parts of a ubiquitous system from the specification of a CS-CIM. As it is specified in MDA, the transformation rules are applied to the CS-CIM metamodel, as a way to avoid the need of proposing ad-hoc transformation rules on a *per-model* basis.

A set of transformation rules instantiate each participant represented in a CS-CIM as software agents in a US-PIM, that is, into an application, a service or both an application and a service. An excerpt of the QVT rules to apply this transformation between the CS-CIM and the US-PIM metamodels is shown in Figure 5.10.

```

mapping Participant::toAgent() : PIM::SoftwareAgent
when {self.type <> CIM::ParticipantRole::PEER} {
  init {
    if (self.type = CIM::ParticipantRole::PASSIVE) then {
      result := object Service{name := self.name+'Service'};
    }else {
      result := object Application{name := self.name+'App'};
    }endif;
  }
  exchanges := self.exchanges->map toSoftwareMessage()->flatten();
  result->map populateSoftwareAgent(self);
}
mapping Participant::toPeerAgents() : app:PIM::Application, service:PIM::Service
when {self.type = CIM::ParticipantRole::PEER} {
  app.name := self.name+'App';
  service.name := self.name+'Service';
  app.exchanges := self.exchanges->map toSoftwareMessage()->flatten();
  service.exchanges := self.exchanges->map toSoftwareMessage()->flatten();
  app->map populateSoftwareAgent(self);
  service->map populateSoftwareAgent(self);
}
    
```

Figure 5.10: An excerpt of the QVT rules to transform a participant of a CS-CIM into a software agent in a US-PIM

As it is possible to observe in these rules, the selection of the target US-PIM element/s is based on the *role* of the origin partici-

pant. If the role is *passive*, then the participant is transformed into a service, since it will be considered to only be in execution when another element of the system requests it some information. On the contrary, if the role is *active*, then the participant is transformed into an application, and it will be meant to be in constant execution, requesting information to services or delivering information to other applications or services. Finally, if the participant has a *peer* role, then it will be transformed into both an application and a service, in order to act in an active way (as an application), but to also provide information to other elements of the system in a more passive manner (as a service).

The *connected* relationship between the participants, which is defined in the structural view of the CS-CIM metamodel (see Chapter 3, Section 3.2.1), is also kept in the transformation process. However, in the US-PIM metamodel, applications can not relate to each other, since an application should not directly request information to another application: applications should only directly request information to a service, and services should request information to other services, as it is defined in the US-PIM metamodel. The reason is two-fold: technically, applications can not provide a public interface (i.e., at software level, to provide a public interface, it is necessary to behave as a service); conceptually, from a functional point of view, applications are not information providers, but infor-

mation consumers and presenters (i.e., they provide the information that they retrieve to the user).

Anyhow, applications can exchange information between them in an **indirect way**, that is, by distributing events. This way, they can interoperate to achieve certain goals, like providing collaboration tools for the users, but with a **low cohesion** between them, and working around the conceptual and technical limitations that were previously mentioned. To do so, an *event handler* is associated to each transformed software agent in order to be able to distribute events to others. Consequently, both applications and services can distribute events. Additionally, a *discoverer* and a *discovery listener* is also associated to each software agent, so as to check for the availability of each other software agents. Moreover, for each software agent, a different instance of a discovery event is produced through the transformation rules, in order to make it possible for each software agent to distribute its own univocally identifiable discovery event.

The association of previous elements to each software agent during the transformation process **fulfills the communication requirements of a ubiquitous system** (see Chapter 4, Section 4.1), by allowing each software to exchange messages with others (i.e., as it was previously explained, only if the messages are exchanged between applications and services, or between services, not between

applications), to be dynamically discovered, and to distribute events to notify changes in their status. The QVT transformation rules to associate the corresponding communication functionalities to each transformed software agent are presented in Figure 5.11.

```

mapping Participant::softwareAgentConnections() {
  self.resolve(PIM::SoftwareAgent)->forEach(agent){
    if (agent.ocIsTypeOf(PIM::Service)) then {
      var s:PIM::Service := agent.ocAsType(PIM::Service);
      self.sendsTo.resolve(PIM::SoftwareAgent)->forEach(delegate) {
        if (delegate.ocIsTypeOf(PIM::Service)) then {
          s.connectsTo += delegate.ocAsType(PIM::Service);
        }else{
          delegate.ocAsType(PIM::Application).requestsServices += s;
        }endif;
      }
    }else {
      var app:PIM::Application := agent.ocAsType(PIM::Application);
      self.sendsTo.resolve(PIM::SoftwareAgent)->forEach(delegate) {
        if (delegate.ocIsTypeOf(PIM::Service)) then {
          app.requestsServices += delegate.ocAsType(PIM::Service);
        }endif;
      }endif;
    }
  };
}
mapping inout PIM::SoftwareAgent::populateSoftwareAgent(p:CIM::Participant){
  var discoveryEvent := object Discovery{
    name := self.name+'DiscoveryEvent';
    medium := p.exchanges.medium.late resolve(PIM::NetworkingTechnology);
    conforms := p.exchanges.conforms.late resolve(PIM::SoftwareProtocol);
    topic := object Topic{name := self.name+'DiscoveryTopic'};
  };
  self.discoverer := object Discoverer{
    name := self.name+'Discoverer';
    callback := object DiscoveryListener {
      name := self.name+'DiscoveryListener';
      discoveredBy := discoveryEvent;
      constrainedBy := object Predicate{name := self.name+'DiscoveryPredicate'};
    };
    listeners += callback;
  };
  self.eventhandler := object EventHandler {name := self.name+'EventHandler'};

  self.exchanges->select(evt | evt.ocIsTypeOf(PIM::Event)).ocAsType(PIM::Event)->forEach(evt){
    self.eventhandler.listeners += object EventListener{
      name := self.name+evt.name+'Listener';
      listens += evt;
      constrainedBy := evt.acceptedBy;
    };
    self.eventhandler.handles += evt;
  };
}

```

Figure 5.11: An excerpt of the QVT rules to include the communication functionalities of a ubiquitous system to each transformed software agent

As it is specified in the QVT rules presented in Figure 5.12, the

messages defined in the CS-CIM are transformed into software messages, that is, *requests*, *responses* and *events*. For each event, there is a topic related to it and a predicate that accepts that topic. Additionally, the generated event listeners accept the generated events. The reason to transform each single message into an event is to allow software agents to notify events each time a request or a response is received. In this way, software agents could monitor the behavior of the others.

```

mapping Message::toSoftwareMessage() : Sequence(SoftwareMessage) {
  init {
    if (not self.oclIsTypeOf(CIM::Initial) and not self.oclIsTypeOf(CIM::End)) then {
      var eventTopic := object PIM::Topic{name := self.name+'Topic'};
      var event := object PIM::Event{
        name := self.name+'Event';
        topic := eventTopic;
        medium := self.medium.late resolve(PIM::NetworkingTechnology);
        conforms := self.conforms.late resolve(PIM::SoftwareProtocol);
      };
      var predicate := object PIM::Predicate{
        name := self.name+'Predicate';
        accepts := event;
      };
      var request := object PIM::Request{
        name := self.name+'Request';
        medium := self.medium.late resolve(PIM::NetworkingTechnology);
        conforms := self.conforms.late resolve(PIM::SoftwareProtocol);
      };
      var response := object PIM::Response{
        name := self.name+'Response';
        medium := self.medium.late resolve(PIM::NetworkingTechnology);
        conforms := self.conforms.late resolve(PIM::SoftwareProtocol);
      };
      request.reply := response;
      response.petition := request;

      result += event;
      result += request;
      result += response;
    }endif;
  }
}

```

Figure 5.12: An excerpt of the QVT rules to transform a message of a CS-CIM into the corresponding elements in a US-PIM

Finally, channels and protocols specified in the CS-CIM are

directly transformed into networking technologies and software protocols, respectively. The idea is to implement these elements through specific technologies in the PSM. Therefore, at the PIM level, they merely play a supportive role for the rest of the elements in the ubiquitous system.

As an example of the result of applying the transformation rules to the structural elements of a CS-CIM, they have been applied to the CS-CIM of a UME. Due to its size in UML graphical notation, an excerpt of the resulting model is shown in Figure 5.13 in standard XMI notation (i.e., an OMG standard XML-based textual notation for UML models, <http://www.omg.org/spec/XMI/>).

It is possible to observe through this sample that the transformation rules produce a design that systematically takes into account many technical details associated to the ubiquitous system to be developed. For example, the instantiation of a service for the patient allows to manage the information about the patient itself. The application for the patient can be used to show biometric information to the end user, or to provide communication tools that can interoperate with the service instantiated for the doctor. This service could manage any information related to the doctor, or to provide some health-related information to the patient.

Moreover, the different requests, responses and events that

CHAPTER 5. MUSYC: AN MDA-BASED METHODOLOGY TO DEVELOP UBIQUITOUS SYSTEMS ON THE BASIS OF THE COMMUNICATIONS

```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" ...>
  <pim:UbiquitousSystem name="UME" channels="WAN PAN" protocols="UMEProtocol
SecureProtocol EfficientProtocol">
    <agents xsi:type="pim:Service" name="MedicalInformationSystemService" ...
isRequestedBy="DoctorApp" connectedBy="DoctorService"/>
    <agents xsi:type="pim:Application" name="SensorApp"
eventhandler="SensorAppEventHandler" exchanges="BiometricInformationEvent
BiometricInformationRequest BiometricInformationResponse"
discoverer="SensorAppDiscoverer" requestsServices="DoctorService PatientService"/>
    <agents xsi:type="pim:Application" name="DoctorApp" .../>
    <agents xsi:type="pim:Application" name="PatientApp" .../>
    <agents xsi:type="pim:Service" name="DoctorService" ... isRequestedBy="PatientApp
SensorApp" connectsTo="PatientService MedicalInformationSystemService"
connectedBy="PatientService"/>
    <agents xsi:type="pim:Service" name="PatientService" ... isRequestedBy="DoctorApp
SensorApp" connectsTo="DoctorService" connectedBy="DoctorService"/>
    ...
  </pim:UbiquitousSystem>
  ...
  <pim:Topic name="BiometricInformationTopic" isTopicOf="BiometricInformationEvent"/>
  <pim:Event name="BiometricInformationEvent" exchangedFrom="SensorApp DoctorApp
DoctorService PatientApp PatientService" medium="PAN" conforms="UMEProtocol
EfficientProtocol" topic="BiometricInformationTopic" ...
acceptedBy="BiometricInformationPredicate"/>
  <pim:Predicate name="BiometricInformationPredicate"
accepts="BiometricInformationEvent" .../>
  <pim:Request name="BiometricInformationRequest" exchangedFrom="SensorApp DoctorApp
DoctorService PatientApp PatientService" ... reply="BiometricInformationResponse"
targets="PatientService"/>
  <pim:Response name="BiometricInformationResponse" exchangedFrom="SensorApp
DoctorApp DoctorService PatientApp PatientService" ...
petition="BiometricInformationRequest"/>
  <pim:Discovery name="SensorAppDiscoveryEvent" medium="PAN" conforms="UMEProtocol
EfficientProtocol" topic="SensorAppDiscoveryTopic"
discovers="SensorAppDiscoveryListener"/>
  <pim:Topic name="SensorAppDiscoveryTopic" isTopicOf="SensorAppDiscoveryEvent"/>
  <pim:Discoverer name="SensorAppDiscoverer" listeners="SensorAppDiscoveryListener"
callback="SensorAppDiscoveryListener"/>
  <pim:DiscoveryListener name="SensorAppDiscoveryListener" ...
discoveredBy="SensorAppDiscoveryEvent"/>
  ...
  <pim:EventHandler name="SensorAppEventHandler"
handles="BiometricInformationEvent" .../>
  ...
  <pim:SoftwareProtocol name="UMEProtocol" system="UME" rules="WAN PAN"
compiles="MedicalRecordEvent MedicalRecordRequest MedicalRecordResponse
MedicalInformationSystemServiceDiscoveryEvent BiometricInformationEvent
BiometricInformationRequest BiometricInformationResponse SensorAppDiscoveryEvent
DoctorAppDiscoveryEvent DoctorServiceDiscoveryEvent PatientAppDiscoveryEvent
PatientServiceDiscoveryEvent InitialEvent EndingEvent" composedBy="SecureProtocol
EfficientProtocol"/>
  <pim:SoftwareProtocol name="SecureProtocol" system="UME" rules="PAN WAN"
compiles="MedicalRecordEvent MedicalRecordRequest MedicalRecordResponse
MedicalInformationSystemServiceDiscoveryEvent DoctorAppDiscoveryEvent
DoctorServiceDiscoveryEvent" composes="UMEProtocol"/>
  <pim:SoftwareProtocol name="EfficientProtocol" system="UME" rules="PAN WAN"
compiles="BiometricInformationEvent BiometricInformationRequest
BiometricInformationResponse SensorAppDiscoveryEvent DoctorAppDiscoveryEvent
DoctorServiceDiscoveryEvent PatientAppDiscoveryEvent PatientServiceDiscoveryEvent"
composes="UMEProtocol"/>
  <pim:NetworkingTechnology name="WAN" system="UME" transports="MedicalRecordEvent
MedicalRecordRequest MedicalRecordResponse
MedicalInformationSystemServiceDiscoveryEvent DoctorAppDiscoveryEvent
DoctorServiceDiscoveryEvent PatientAppDiscoveryEvent
PatientServiceDiscoveryEvent" .../>
  <pim:NetworkingTechnology name="PAN" .../>
  ...
</xmi:XMI>

```

Figure 5.13: An excerpt of the result of applying the QVT transformation rules to the sample CS-CIM of a UME

need to be exchanged between these elements are instantiated, and they are linked to the corresponding communication protocols and networking technologies. Furthermore, the event handlers and discoverers related to each software agent are also instantiated. Their corresponding listeners to subscribe to each possible event that could be received in a software agent are instantiated too.

Additionally, it is interesting to highlight that the transformation rules instantiate the sensor in the CS-CIM specification as an *application* in the US-PIM. The reason is that the role of the sensor, according to the CS-CIM, is *active*, which is conceptually correct, since in the UME it constantly provides information to the doctor and the patient, even without their previous request of that information. However, technically, there could be a bit of confusion about the *nature* of a sensor, that is, if the software associated to the sensor should be considered as a service or as an application. The transformation rules avoid this confusion, and make explicit that the sensor can not operate as a service, since it is an autonomous, pro-active part of the ubiquitous system (and, in practice, it could be endowed by a certain level of “intelligence” to be able to automatically detect a health issue on the patient). This is one example about how **a correct conceptualization of a system as a CS-CIM can have direct benefits in the software design.**

5.3.2. CS-CIM to US-PIM Transformation: Behavioral View

To obtain the behavioral elements from the choreography associated to the behavioral view of a CS-CIM, the transformation rules have to produce the corresponding *elemental communication activities* (message exchanging, event distribution or dynamic discovery) related to each *choreography activity*. Also, it needs to be taken into consideration that, as it was explained in Chapter 4, Subsection 4.2.2, the idea in the US-PIM is that each *choreography activity* that involves two software agents (i.e., there can be activities in which a unique software agent communicates with itself), entails a previous discovery activity that checks for the availability of both of the software agents (i.e., some of them may be unreachable, since they can be moving, and their networking capabilities can change).

An example transformation is depicted in Figure 5.14, so as to illustrate the overall rationale behind the QVT transformation rules to be presented below. The resulting model is represented as a UML sequence diagram in order to more precisely outline the expected behavior of each choreography activity in a US-PIM.

Each choreography specified in the CS-CIM is transformed into a **software agent choreography** in the target US-PIM, as it is defined in the QVT rules depicted in Figure 5.15.

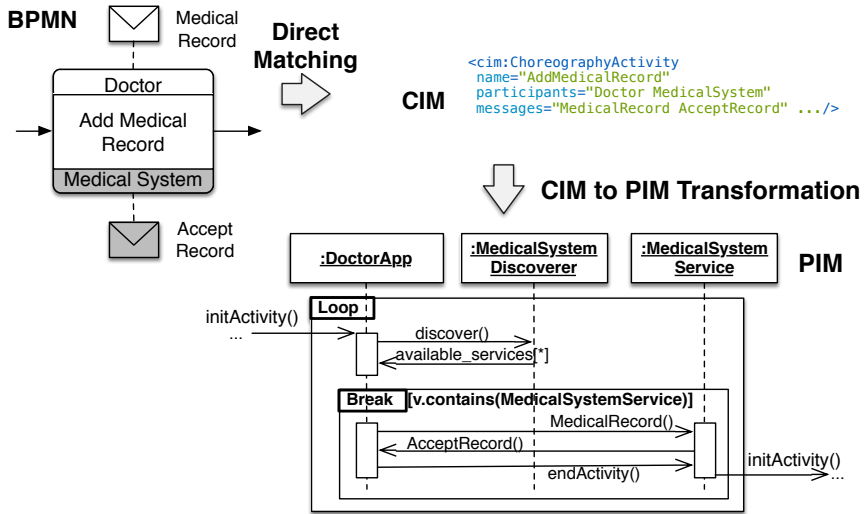


Figure 5.14: An example of a transformation from a BPMN choreography diagram into a sequence diagram

```

mapping Choreography::toSoftwareAgentChoreography() : PIM::SoftwareAgentChoreography {
    name := self.name;
    system := self.system.late resolveOne(PIM::UbiquitousSystem);
    participants := self.participants.resolve(PIM::SoftwareAgent);
    messages := self.messages.resolve(Sequence(PIM::SoftwareMessage)->flatten());
    activities := self.activities->map toChoreographyActivity()->flatten();
    activities->forEach(act){
        messages += act.messages;
    };

    startingEvent := self.startingEvent->map toStartingEvent(self)->asSequence()->at(1);
    endingEvents := self.endingEvents->map toEndingEvents();
    activities += self.links->map toActivities(result)->flatten();
}

```

Figure 5.15: An excerpt of the QVT rules to transform a choreography of a CS-CIM into a choreography in a US-PIM

Since a participant can be transformed into multiple software agents (i.e., if the participant has a *peer* role), then there can not be a 1-to-1 transformation between the choreography activities in the CS-CIM and the US-PIM. As it is possible to observe in Figure

5.16, the proposed QVT transformation rules may instantiate several activities in the US-PIM from a unique activity in the CS-CIM. Those derived activities involve the possible combinations between the different software agents that are derived from the corresponding participants in the CS-CIM. Also, for each sub-activity in the CS-CIM it is created an elemental communication activity (i.e., a transaction) in the US-PIM.

```

mapping CIM::ChoreographyActivity::toChoreographyActivity() :
Sequence(PIM::ChoreographyActivity) {
  init{
    var counter := 0;
    var msgs := self.messages.resolve(Sequence(PIM::SoftwareMessage))->flatten();

    self.sourceParticipant.resolve(PIM::SoftwareAgent)->forEach(sAgent){
      self.targetParticipant.resolve(PIM::SoftwareAgent)->forEach(tAgent){
        var activity := object PIM::ChoreographyActivity{
          name := self.name+counter.toString();

          participants += sAgent;
          participants += tAgent;

          counter := counter+1;
        };

        activity.transactions += object DiscoveryActivity{
          name := sAgent.name+'2'+tAgent.name+'Discovery';
          sourceParticipant := sAgent;
          targetParticipant := tAgent;
        };
        activity.transactions += self.subactivities->map
toElementalCommunicationActivities(activity, msgs)->flatten();
        activity.messages += activity.transactions->select(tl
t.ocIsTypeOf(PIM::EventDistributionActivity)).oclAsType(PIM::EventDistributionActivity).
relatedEvent;

        result += activity;
      };
    };
  }
}

```

Figure 5.16: An excerpt of the QVT rules to transform a choreography activity of a CS-CIM into a set of choreography activities in a US-PIM

The *links* that connect a source and a target activity in the CS-

CIM do not have a direct matching with any element in the US-PIM. Consequently, they are transformed to *sequential events*, as it can be observed in Figure 5.17. These events are delivered from a software agent in the source activity, and it is marked as an *starting message* in the target activity. The idea is that, a software agent delivers an event after the completion of a choreography activity, which will start another activity. This way, there is no coupling between activities, nor any direct association between them. The benefit is that, in later development stages, the activities can be separately implemented, without taking into account any details of the implementation of the others.

Also it is taken into account that a link in the CS-CIM refers to the association between an activity and the delivery of an event. In the US-PIM, this situation derives in the instantiation of a choreography activity for each event to be delivered, including an event distribution activity, as it is shown in the rules presented in Figure 5.17.

Figure 5.18 illustrates the US-PIM (in XMI standard notation) that results from applying the described transformation rules to the behavioral elements of a CS-CIM of a UME. As it can be observed, each choreography activity includes the discovery of the target software agent from the source software agent. Since the patient plays a *peer* role in the CS-CIM, and it is transformed into both a service

```

mapping CIM::Link::toActivities(inout choreography:PIM::SoftwareAgentChoreography) :
Sequence(PIM::ChoreographyActivity){
  init{
    var src:CIM::FlowObject := self.source;
    var tgt:CIM::FlowObject := self.target;

    var activity1:Bag(Sequence(PIM::ChoreographyActivity));
    var activity2:Bag(Sequence(PIM::ChoreographyActivity));

    if (src.oclIsKindOf(CIM::Event)) then{
      activity1 := src.oclAsType(CIM::Event)->map
toEventDistributionActivity();
      result += activity1->flatten();
    }else {
      activity1 :=
src.oclAsType(CIM::ChoreographyActivity).resolve(Sequence(PIM::ChoreographyActivity))
->asBag();
    }endif;

    if (tgt.oclIsKindOf(CIM::Event)) then{
      activity2 := tgt.oclAsType(CIM::Event)->map
toEventDistributionActivity();
      result += activity2->flatten();
    }else {
      activity2 :=
tgt.oclAsType(CIM::ChoreographyActivity).resolve(Sequence(PIM::ChoreographyActivity))
->asBag();
    }endif;

    activity1->flatten()->forEach(act1){
      act1.startingMessages += src.starter->map toGateway();
      activity2->flatten()->forEach(act2){
        var startingMessage := object PIM::Sequential{
          name := act2.name+'StartingEvent';
        };
        act1.messages += startingMessage;
        act1.transactions += object PIM::EventDistributionActivity{
          name := startingMessage.name+'DistributionActivity';
          relatedEvent := startingMessage;
          sourceParticipant := act1.participants->at(1);
        };
        act2.startingMessages += tgt.starter->map toGateway();
        act2.startingMessages += startingMessage;
      };
    };

    choreography.messages += activity1->flatten().startingMessages;
    choreography.messages += activity2->flatten().startingMessages;
  }
}

```

Figure 5.17: An excerpt of the QVT rules to transform a link of a CS-CIM into choreography activities in a US-PIM

and an application, the activities in which the corresponding software agents are involved are split into multiple ones. Particularly,

in the sample figure, the *MeasureBiometricSignal* activity is transformed into the following activities:

- An activity that involves the interaction between the sensor application and the patient application. Since applications can only interact through events, then this activity consists of the distribution of an event related to the notification of the biometric signals measured by the sensor to the patient application. For example, this activity may allow the sensor to notify the patient about any changes in his/her health.
- Another activity that involves the interaction between the sensor application and the patient service. For instance, this activity could enable the sensor to store/retrieve historical information about the measured biometric signals into/from the patient service. In this case, since there is an interaction between an application and a service, both event distribution and message exchanging are allowed, which, for example, makes it possible for the sensor to store biometric signals as the values change (an event is notified) or to request historical information about the patient (through request messages).

Additionally, the split of an activity involves that the previous activities activate them in a non-deterministic order. In this case, in the CS-CIM, the event that initializes the choreography is delivered

```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>
  <pim:UbiquitousSystem name="UME" ...>
    ...
    <choreographies ... name="MainChoreography" startingEvent="InitialEvent"
endingEvents="EndingEvent">
      <activities name="MeasureBiometricSignals0"
messages="BiometricInformationEvent" participants="PatientApp SensorApp"
startingMessages="MeasureBiometricSignals0StartingEvent">
        <transactions xsi:type="pim:DiscoveryActivity"
name="PatientApp2SensorAppDiscovery" sourceParticipant="PatientApp"
targetParticipant="SensorApp"/>
        <transactions xsi:type="pim:EventDistributionActivity"
name="SensorApp2PatientAppEventDistribution" sourceParticipant="SensorApp"
targetParticipant="PatientApp" relatedEvent="BiometricInformationEvent"/>
      </activities>
      <activities name="MeasureBiometricSignals1"
messages="BiometricInformationEvent" participants="PatientService SensorApp"
startingMessages="MeasureBiometricSignals1StartingEvent">
        <transactions xsi:type="pim:DiscoveryActivity"
name="PatientService2SensorAppDiscovery" sourceParticipant="PatientService"
targetParticipant="SensorApp"/>
        <transactions xsi:type="pim:EventDistributionActivity"
name="SensorApp2PatientServiceEventDistribution" sourceParticipant="SensorApp"
targetParticipant="PatientService" relatedEvent="BiometricInformationEvent"/>
        <transactions xsi:type="pim:MessageExchangingActivity"
name="SensorApp2PatientServiceMessageExchanging" sourceParticipant="SensorApp"
targetParticipant="PatientService" request="BiometricInformationRequest"/>
      </activities>
      <activities name="InitialEventDistributionActivity"
messages="MeasureBiometricSignals0StartingEvent
MeasureBiometricSignals1StartingEvent">
        <transactions xsi:type="pim:EventDistributionActivity"
name="InitialEventDistributionTransactionActivity"
sourceParticipant="PatientApp" relatedEvent="InitialEvent"/>
        <transactions xsi:type="pim:EventDistributionActivity"
name="MeasureBiometricSignals0StartingEventDistributionActivity"
relatedEvent="MeasureBiometricSignals0StartingEvent"/>
        <transactions xsi:type="pim:EventDistributionActivity"
name="MeasureBiometricSignals1StartingEventDistributionActivity"
relatedEvent="MeasureBiometricSignals1StartingEvent"/>
      </activities>
    </choreographies>
  </pim:UbiquitousSystem>
  ...
</xmi:XMI>

```

Figure 5.18: An excerpt of the result of applying the QVT transformation rules to the sample CS-CIM of a UME

before the *MeasureBiometricSignal* activity is started. Therefore, in the US-PIM, the distribution activity in which the initial event is delivered also involves the activation of the two activities that result from the transformation of the *MeasureBiometricSignal* activity

from a CS-CIM to a US-PIM.

Finally, as a side note, a set of transformation rules have also been defined to produce a **UML sequence diagram from a CS-CIM**, by applying them to the **metamodels of UML and a CS-CIM**. Due to their complexity in QVT, these rules have been implemented in ATL, and can be consulted in Appendix VII. This additional contribution provides a manner to obtain a diagram in a standard notation that illustrates the sequence of interactions between the different elements of the system to be designed. Moreover, the diagram could be used to **refine the US-PIM** that is obtained by applying the defined transformation rules, just in case that some aspects of the automatically generated design could be improved.

5.4. Stage 3: Implementation of the Ubiquitous System

The implementation of a ubiquitous system is carried out in MUSYC through the specification of a US-PSM, and the code generation from that model. The overall description of the third stage is depicted in Figure 5.19.

The following subsection describes how the transformation from a US-PIM to a US-PSM could be carried out, independently of the target platforms. Afterwards, the succeeding subsection explains

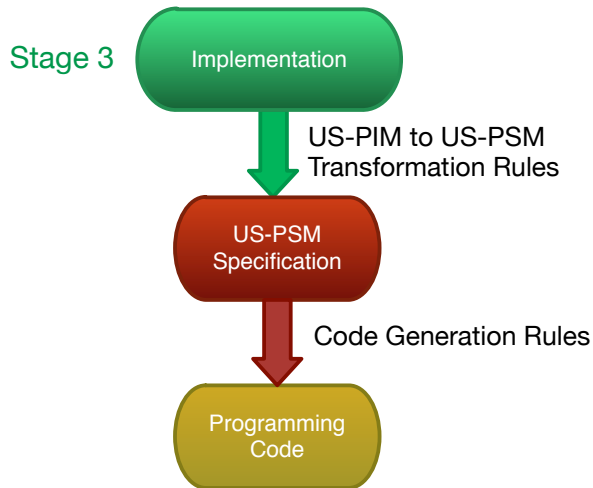


Figure 5.19: Third development stage specified in MUSYC

how some model-to-text transformation rules could be applied to a US-PSM to generate programming code.

5.4.1. Transformation from a US-PIM to a US-PSM

To produce a more specific design of a ubiquitous system it is possible to **transform a US-PIM into a US-PSM**. However, the specification of the US-PSM depends on the particular technological platforms to be used. Therefore, it is not possible to define a unique US-PSM metamodel or a set of transformation rules. Nonetheless, it is described a general way of approaching the transformation from the US-PIM metamodel into any US-PSM metamodel.

The proposal is that the transformation rules should take into

account that, in the US-PSM, it could be appropriate to separate the elements that are directly derived from the US-PIM from the implementations of those abstractions by means of specific technologies. The rationale behind this recommended separation is to produce a model that differentiates the elements that are part of the design of the ubiquitous system from the elements that will be supported by specific technologies and whose programming code will be completed during the final programming stages of MUSYC (see next subsection).

To do so, for each element in the US-PIM metamodel, there should be an *abstract* element in the US-PSM metamodel and an *implementation* element that is a specialization of the abstract one. Figure 5.20 exemplifies how to make that transformation through QVT rules. That figure particularly depicts the transformation between a software agent in a US-PIM to the corresponding software components in a target US-PSM.

This transformation approach contributes to improve the following quality properties in the ubiquitous system to be developed:

- *Maintainability*: The *implementation* elements encapsule the technical parts of the ubiquitous system. Therefore, if an error needs to be solved or an improvement has to be made, these elements are the only ones that need to be modified. More-

over, the design of the ubiquitous system is kept, since the “abstract” elements (i.e., the elements that are directly derived from the US-PIM) are not changed during the maintenance processes.

- *Reusability*: The *abstract* elements can be reused in different implementations of the same ubiquitous system. Conversely, the *implementation* elements can be reused for developing other ubiquitous systems.

5.4.2. Code Generation from a US-PSM

The US-PSM allows to **systematically generate code** for the specific adopted platforms (programming languages, operating systems, middleware, etc.). To do so, the OMG’s MOF Model to Text Transformation Language standard (MOFM2T, <http://www.omg.org/spec/MOFM2T/1.0/>) can be used to specify transformation rules from a model conforming to MOF (or UML, since MOF is its metamodel) into a textual notation representing a programming code. However, any specific set of MOFM2T transformation rules can be provided in MUSYC methodology, since they directly depend on the target platforms. Nonetheless, this subsection provides some general approaches for generating code.

```

mapping PIM::SoftwareAgent::toPSAgents() : Sequence(PSM::SoftwareAgent){
  init{
    if (self.oclIsTypeOf(PIM::Application)) then{
      var abstractApp := object PSM::Application{
        name:='Abstract'+self.name;
        isAbstract:=true;
      };
      result += abstractApp;
      result += object PSM::Application{
        name:=self.name+'Impl';
        isAbstract:=false;
        extendsFrom := abstractApp;
      };
    }else{
      var abstractServ := object PSM::Service{
        name:='Abstract'+self.name;
        isAbstract:=true;
      };
      result += abstractServ;
      result += object PSM::Service{
        name:=self.name+'Impl';
        isAbstract:=false;
        extendsFrom := abstractServ;
      };
    }endif;
  }
}

```

Figure 5.20: Some example QVT rules to transform from software agents in a US-PIM to the corresponding elements in an unspecified target US-PSM

For example, the elements associated to the structural view of the US-PSM can be used to **produce classes and objects** using the adopted programming languages and according to the target middleware solutions. For example, if Java is the adopted programming language, then the services could be converted into Java classes that inherit from the classes supporting service programming according to the adopted middleware (e.g., the *ServantBase* class in CORBA specification).

The *choreography activities* could be used to produce the codification of some methods to be provided by the generated classes. These methods will consist on carrying out the multiple *elemental communication activities* that conform a choreography activity, probably using the mechanisms provided by the adopted middleware solutions. In consequence, the elements related to the behavioral view of a US-PIM can be analyzed to produce a part of the **public interface** of each class, and a **preliminar implementation** of the corresponding methods.

It is worth to be mentioned that some existing middleware technologies already provide mechanisms to produce code from abstract specifications of the distributed system (or ubiquitous system) to be developed. For example, in CORBA, the interfaces of the multiple services to be developed are specified in an Interface Definition Language (IDL). If it is necessary, some MOFM2T transformation rules can be specified to transform part of the US-PSM into these abstract specifications, and to use the mechanisms provided by the middleware to generate the code. Consequently, it is possible to support code generation with existing mechanisms, ensuring a certain level of interoperability and compatibility between the ubiquitous systems to be developed using MUSYC and some existing target platforms.

Finally, it can be remarked that, even if the definition of the

MOFM2T transformation rules require additional development efforts, then their specification can be reused across different development projects sharing the same target computing platforms. Therefore, the development costs can be, ultimately, reduced.

5.5. CASE Tools Supporting MUSYC

The proposed metamodels and QVT transformation rules have been implemented as Computer-Aided Software Engineering (CASE) tools using the Eclipse Modeling Framework (EMF, <http://www.eclipse.org/emf>), which incorporates modeling mechanisms that facilitate model specification and code generation.

Firstly, an Eclipse Plug-in has been developed to assist software analysts during the **specification and checking** of a CS-CIM. To do so, the CS-CIM metamodel has been represented in ECORE format and an Eclipse GenModel (http://wiki.eclipse.org/Graphical_Modeling_Framework/Models/GMFGen) schema has been defined. This schema allows EMF to generate a CS-CIM model editor that produces models using the XMI standard notation. The conformance of the CS-CIM definition with the metamodel specification can be checked through a validation tool that is also integrated into the editor. The sample definition of a UME through the implemented Eclipse CS-CIM editor is depicted in Figure 5.21.

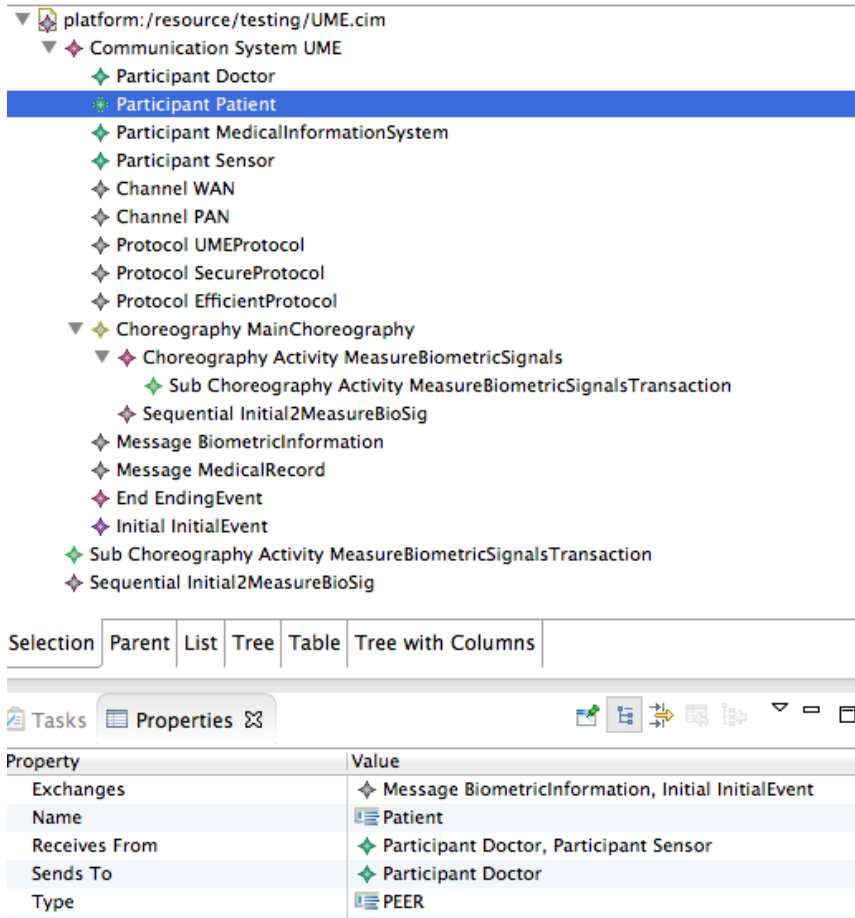


Figure 5.21: A sample UME defined using the implementation of an Eclipse Plug-in to define and check CS-CIMs

The QVT rules to transform a CS-CIM into a US-PIM have been defined in Eclipse, using the Eclipse QVT plug-in (<http://projects.eclipse.org/projects/modeling.mmt.qvt-oml>). This plug-in **checks the validity** of the transformation rules and can automatically apply them on a CS-CIM (created with the previously described CS-CIM editor) to produce a US-PIM.

A model editor for US-PIMs has also been implemented to **assist in the refinement** of the model that is automatically transformed from the CS-CIM. Consequently, the US-PIM metamodel has been implemented in ECORE format and an Eclipse GenModel schema has been defined. Similarly to the CS-CIM editor, the US-PIM one can **check for the validity of a refined model** (i.e., its conformance with the US-PIM metamodel specification). Figure 5.22 depicts the US-PIM that results from applying the QVT rules to the CS-CIM of a UME using the implemented Eclipse US-PIM editor.

A **sample Java code generator** has been implemented to show the feasibility of building CASE tools to automatically generate code from a US-PSM. The code generator can be easily modified to produce code in other programming languages. The code generation process is implemented using the Acceleo Eclipse plug-in (<http://www.eclipse.org/acceleo>), which facilitates generating textual outputs from ECORE models. However, the transformation rules have been defined in MOFM2T standard notation. As an example, the MOFM2T rules to produce a Java event listener are depicted in Figure 5.23.

Also, another sample code generator produces Web Services Description Language (WSDL), which is an standard W3C notation to define public interfaces for web services. This generator extracts the public interface from the choreography activities defined in the

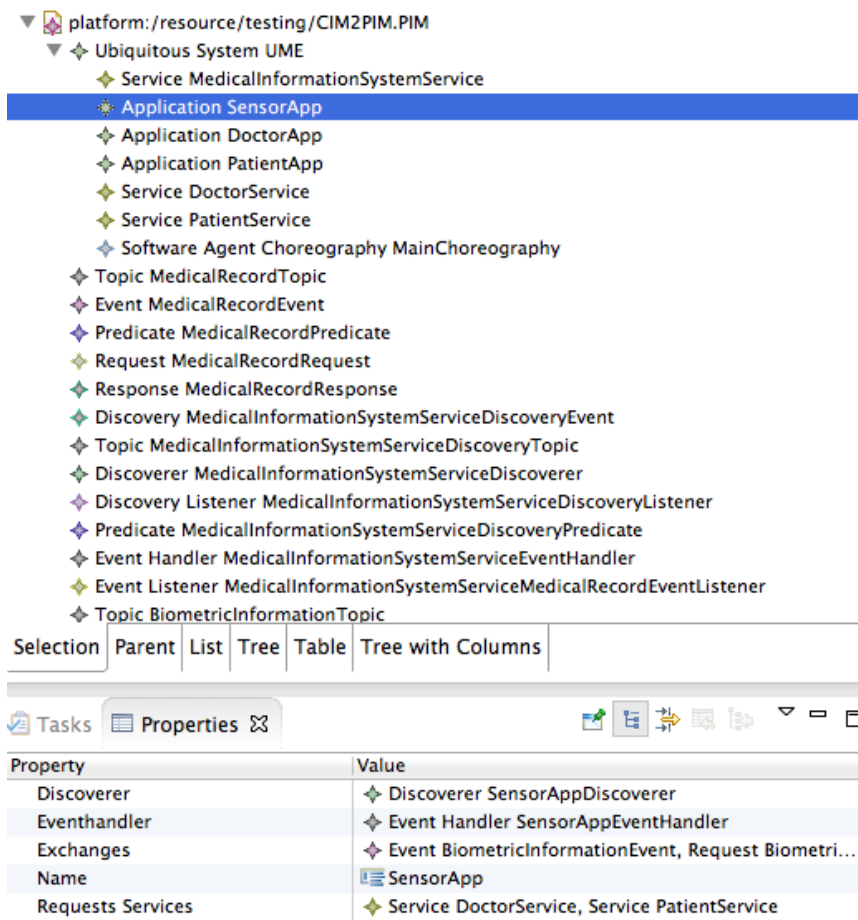


Figure 5.22: The result of transforming the CS-CIM of a UME into a US-PIM using QVT rules, depicted using the implementation of an Eclipse Plug-in to define and check US-PIMs

US-PSM and produce the corresponding WSDL specifications. Figure 5.24 represents an excerpt of the WSDL interfaces automatically generated for a *patient service* in a UME.

```
[comment encoding = UTF-8 /]
[module EventListener2Code(...)]

[template public generateEventListener(listener : EventListener)]
[file('listener/' + listener.name + '.java', false, 'UTF-8')]

package es.ugr.listener;
[for(event : Event | listener.listens)]
[if(event.isAbstract)]
import es.ugr.event.[event.name/];
[/if]
[/for]
[for(predicate : Predicate | listener.constrainedBy)]
[if(predicate.isAbstract)]
import es.ugr.predicate.[predicate.name/];
[/if]
[/for]
import java.util.List;

public[if(listener.isAbstract)] abstract[/if] class [listener.name/]
    [if(not listener.isAbstract)] extends [listener.extendsFrom.name/][[/if]]{

    [if(listener.isAbstract)]
    [for(predicate : Predicate | listener.constrainedBy)]
    [if(predicate.isAbstract)]
    protected List<[predicate.name/]> [predicate.name.toLowerCaseFirst()/]List = null;
    [/if]
    [/for]
    [/if]

    [for(event : Event | listener.listens)]
    [if(event.isAbstract)]
    [if(listener.isAbstract)]
    public abstract void check([event.name/] anEvent);
    public abstract void action([event.name/] anEvent);
    [else]
    public void check([event.name/] anEvent){
        ...
    }

    public void action([event.name/] anEvent){
        // TODO Auto-generated method stub
    }
    [/if]
    [/if]
    [/for]
}
[/file]
[/template]
```

Figure 5.23: A sample transformation from an event listener defined in an undefined US-PSM to Java code, implemented in MOFM2T standard notation


```
<definitions name="PatientService" ...>
  <xsd:element="ObtainMedicine"/>
  ...
  <wsdl:portType name="PatientServicePortType">
    <wsdl:operation
name="BiometricInformationRequest">
      ...
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="PatientServiceBinding" ...>
    <wsdl:operation
name="BiometricInformationRequest">
      <soap:operation
        soapAction=".../
BiometricInformationRequest"/>
    </wsdl:operation>
  ...
  <service name="PatientService">
    <documentation>Documentation</documentation>
    <port
name="BiometricInformationRequestPort" ...>
      <soap:address
        location=".../PatientService"/>
    </port>
  </service>
  ...
</definitions>
...
<definitions name="PatientDiscoverer" ...>
  <wsdl:portType name="PatientDiscovererPortType"/>
  <wsdl:binding name="PatientDiscovererBinding".../>
  <service name="PatientDiscoverer">
    ...
  </service>
  ...
...

```

Figure 5.24: An excerpt of a sample WSDL service interface auto-
matically derived from a US-PSM

5.6. Conclusions

This chapter has presented an **MDA-based Methodology to Develop Ubiquitous Systems on the Basis of the Communications (MUSYC)**. The conceptual models presented in previous chapters serve as metamodels for CS-CIMs and US-PIMs, and as a **conceptual framework** on which the whole methodology is founded.

Particularly, the whole development process is based on the design of a CS-CIM, and its systematic transformation to a US-PIM, a US-PSM and, finally, to programming code. Therefore, MUSYC avoids the adoption and use of specific technologies until the specification of the US-PSM. In consequence, if more advanced technologies become available during the development of a ubiquitous system, or even later, the CS-CIM and US-PIM designs could be completely reused.

Moreover, the **technologies do not guide the whole development process** from the initial stages, as it commonly occurs in many current developments, in special in ubiquitous systems. This way, it is encouraged to **focus most development efforts in the specification of a design that fulfills with the user requirements**, rather than on the adoption or use of concrete technologies. In fact, in MUSYC, there is direct match between the requirements speci-

fied through a use case model and BPMN choreographies, and the CS-CIM specification.

To assist in the development of ubiquitous systems through MUSYC, and to demonstrate the feasibility of **automatizing and validating** the transformation from the specification of a CS-CIM to executable code for specific target computing platforms (operating systems, middleware, etc.), a set of **CASE tools** have been implemented. However, it is worth to be mentioned that the ontologies that were specified for a CI-CS and a PI-US can be used to model specific CS-CIMs or US-PIMs. Also, a reasoner could help into checking and simplifying (i.e., finding equivalencies between different elements of the design) the specified models.

Finally, MUSYC demonstrates that **it could be possible to approach the whole development of a ubiquitous system by focusing on the design of the communications**. For example, a MUSYC-based development process produces an initial design of the applications and services to be developed (i.e., MUSYC does not tackle with the design of the user interface, data model, etc.), and the interactions between them. Therefore, it is possible to establish that **the communications should be considered as a central part of any ubiquitous system development**: the specification of the mechanisms supporting them should be approached before the implementation stage, since they are a fundamental part of the sys-

tem that even constrains the functionalities and quality properties that can be provided to the end user.

Chapter 6

Validation of MUSYC through the Development of a Middleware and a Software Framework for Ubiquitous Systems: BlueRose

This chapter describes how MUSYC can be applied to the development of middleware for ubiquitous systems. This way, it is shown that MUSYC, in addition to assist in the development of ubiquitous systems, can also be specifically applied to develop middleware technologies that can be **reused across different developments of multiple ubiquitous systems**.

Firstly, it is described the MUSYC-based development of a middleware called BlueRose. Afterwards, a **software framework** (see Chapter 2, subsection 2.1.4.2) comprising a set of hot and

frozen spots is identified in BlueRose, which makes explicit the different supporting mechanisms that it provides to assist during the implementation stages of a ubiquitous system. Furthermore, the **quality properties** of BlueRose are also analyzed, and compared with the quality properties of CORBA and ICE, which are two of the most well-known and used middleware technologies.

To conclude, several **real projects** in which BlueRose has been used are briefly described. These projects validate both the practical feasibility of MUSYC and its applicability to the development of supporting technologies (like middleware) that facilitate the implementation of ubiquitous systems.

6.1. Applying MUSYC to the Development of Middleware Solutions for Ubiquitous Systems

MUSYC has been applied to the development of a middleware for ubiquitous systems called BlueRose. The following subsections describe the CS-CIM, US-PIM and US-PSM models associated with this middleware. The QVT rules and CASE tools proposed in previous chapter assist in the transformation between the CS-CIM and the US-PIM. The US-PSM is obtained through the adoption of specific target computing platforms to be used by BlueRose middleware.

6.1.1. Communication Requirements Analysis

As a first step to develop BlueRose, the more general requirements of a middleware have been analyzed, without taking into consideration the specific requirements of a middleware for ubiquitous systems. In consequence, it has been considered that a middleware, at a high abstraction level, is a software whose main objective is to support the exchange of messages between different participants in a communication system. To understand how this support is fundamentally achieved, the concepts related to **distributed objects** [87] have been studied, since they are present in many traditional middleware solutions, like CORBA or RMI.

Distributed objects are separated into an interface (*Object Proxies*), which resides in a local machine, and an implementation (*Object Servants*), which is executed in a remote machine. Thus, at CS-CIM level, it has been devised a **“conversation” between proxies and servants**, involving the exchange of multiple messages. Therefore, at CS-CIM level, the communication is based on message passing. Conceptually, this association between message passing and a high abstraction perspective of the software-based communications is consistent with some previous research works, like [33], which also points-out that, at software level, any implementation of a communication paradigm is fundamentally based on message passing semantics.

The study of several middleware communication protocols, associated with the traditional middleware technologies mentioned in Chapter 2, Section 2.1.4.1, has allowed to identify the fundamental messages to be exchanged between proxies and servants:

- *Information Petition*: The messages that are transferred to make a request of information.
- *Information Supply*: The messages that are transferred to provide the requested information.

To figure out the instantiation of the elements in the behavioral view of the CS-CIM metamodel, it has been assumed that **proxies constantly request information to servants**. If a servant requires additional information from another servant, then it makes an **additional petition**. In parallel, **proxies may request information to other proxies**, in order to, for example, coordinate their activities, or to obtain information that was previously retrieved from any servants by the target proxies. The middleware execution finalizes when there are no other requests to make.

As it is specified in MUSYC, this behavior has been depicted through a use case model in Figure 6.1, and through a BPMN 2.0 Choreography model in Figure 6.2. As it is explained in Chapter 5, Subsubsection 5.2.2.2, there is a direct matching between a BPMN

choreography model and the elements in the behavioral view of the CS-CIM metamodel, which allows to directly instantiate the behavioral elements of the CS-CIM on the basis of a BPMN choreography model.

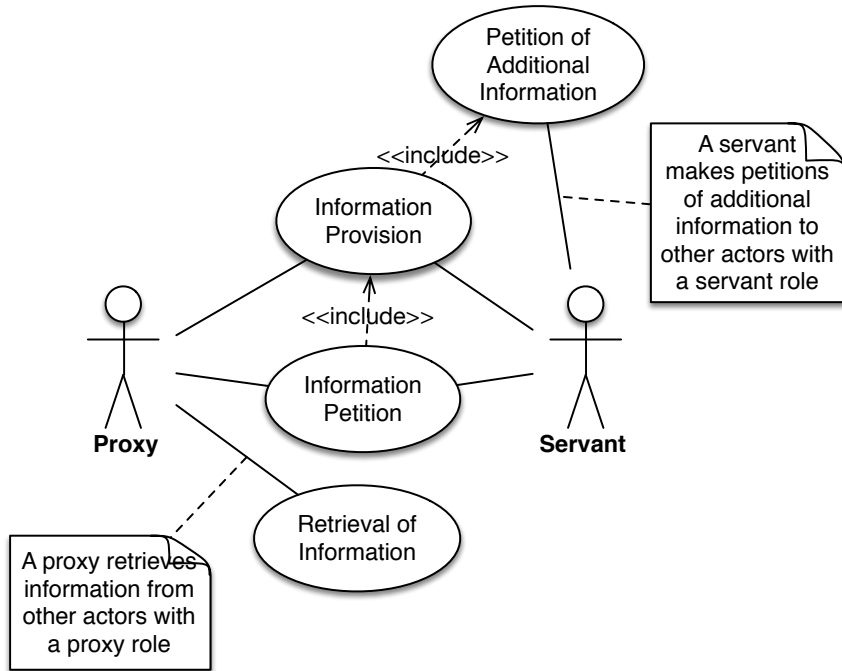


Figure 6.1: A use case model representing the functionalities that are carried out by proxies and servants in BlueRose middleware

To instantiate the structural elements of the CS-CIM, it has been taken into consideration that a middleware protocol should, at least, support *information petition* and *information supply* messages. Also, these messages need to be transported through a channel that is able to connect any proxies with any servants. Since, at CS-CIM

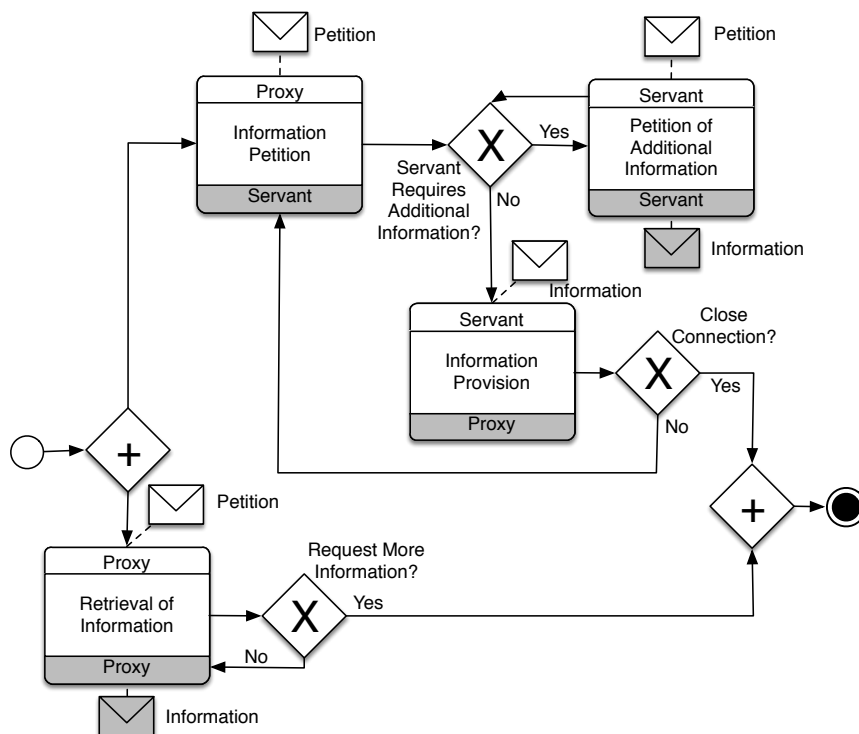


Figure 6.2: The elements in the behavioral view of a CS-CIM supporting the design of the BlueRose middleware, represented as a BPMN 2.0 Choreography

level it is not appropriate to adopt any specific technology, it is assumed that the channel is a *broker*, which is an abstraction that was adopted in the CORBA specification to refer to any channel allowing the distribution of information through a network [87].

Finally, the proxies should play a *peer* role, since they request information, but they also wait for information petitions from other proxies. However, servants play a *passive* role, since they only wait for petitions from proxies in order to initiate their activities.

The instantiation of the concepts in the structural view of the CS-CIM metamodel is represented in Figure 6.3 as a UML class diagram.

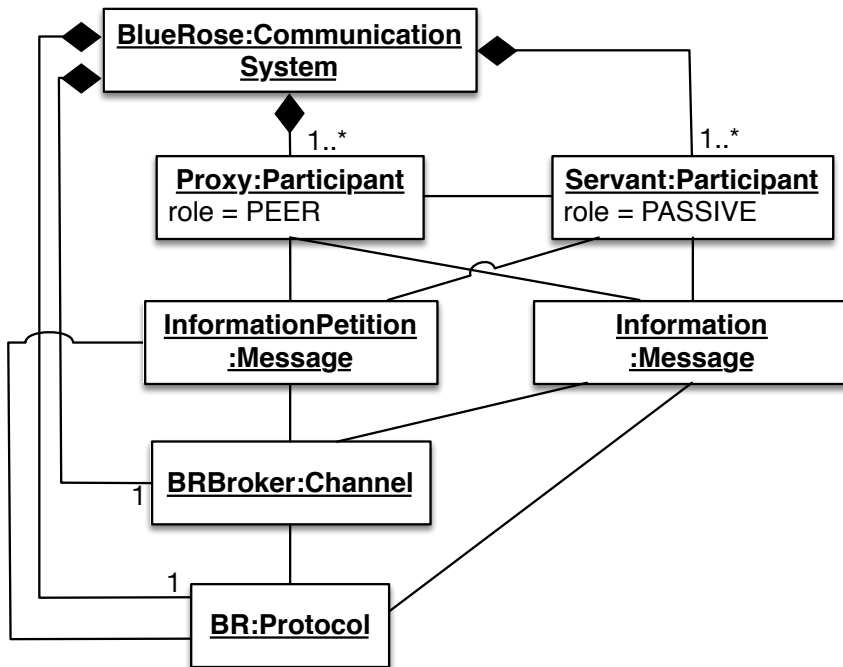


Figure 6.3: The elements in the structural view of a CS-CIM supporting the design of the BlueRose middleware, represented as a UML class diagram

The whole devised CS-CIM for BlueRose is provided in Appendix VIII using the XMI standard notation.

6.1.2. Ubiquitous System Requirements Analysis

From the CS-CIM, it is possible to **systematically obtain a US-PIM**, after applying the QVT transformation rules that were described in Chapter 5, Section 5.3. The whole US-PIM for BlueRose is provided in Appendix IX, in standard XMI notation (due to its size in graphical notation). However, a set of excerpts from the US-PIM are described below, in order to describe some of the most compelling aspects of the transformation.

As it is shown in Figure 6.4, at US-PIM level, the BRProtocol is transformed into a corresponding software protocol, and the BRBroker is also transformed into a networking technology. The servants are transformed into services (at CS-CIM level, they play a *passive* role), and the proxies into applications and services (at CS-CIM level, they have a *peer* role). A proxy service will be considered as a software artifact that is able to provide internal information about a proxy application. Conversely, a proxy application will be considered to be a software artifact that requests information to servants or delivers information to other proxies (applications or services).

The messages for information petition and supply are transformed into *requests* and *responses*. Note that this transformation involves that a petition is transformed into a *petition request* and a

petition response, and, similarly, a supply into an *information request* and an *information response*. This way, **for each message there is a confirmation of its reception**. Consequently, for requesting information and delivering a response, BlueRose exchanges four messages:

- *Petition request*: the request of a piece of information.
- *Petition response*: a confirmation of the reception of the request.
- *Information request*: the information that was requested.
- *Information response*: the confirmation of the correct delivery of the information that was previously requested.

This behavior is similar to the operation mode of many communication protocols, since it ensures the **reliability** of the communications. For example, in TCP/IP, each message are separately confirmed.

Note that the naming of the different requests and responses that are automatically produced by the QVT transformation rules may lead to some misunderstandings of the semantics. For instance, the supply of information is called *information request*, and its delivery confirmation is called *information response*. This is a good

example of a model that **should be refined** after been systematically produced by the transformation rules.

The QVT transformation rules also instantiate **an event for each information petition and supply**. This way, it is possible to deliver events from proxies or servant to notify that an information was requested or delivered, or to notify any changes in their internal status. To be able to deliver these events, there is an event handler related to each proxy application, proxy service or servant. Furthermore, each of these software agents are associated with a discoverer in order to be able to dynamically discover other software agents. In consequence, BlueRose supports the communication functionalities that should be present in a ubiquitous system: message exchanging, event distribution and dynamic discovery.

Figure 6.5 depicts the transactions that are involved in each activity that is supported in BlueRose, as it is specified in the US-PIM. For example, a petition of information from a proxy application to a servant involves the following transactions:

- A discovery of the servant, in order to ensure the availability of the service from the proxy application.
- A notification of an event from the proxy application about its intention to request some information.

```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ...>
  <pim:UbiquitousSystem name="BlueRose" channels="BRBroker"
protocols="BRProtocol">
    <agents xsi:type="pim:Service" name="ServantService"
eventhandler="ServantServiceEventHandler" exchanges="InformationEvent
InformationRequest InformationResponse InformationPetitionEvent
InformationPetitionRequest InformationPetitionResponse"
discoverer="ServantServiceDiscoverer" isRequestedBy="ProxyApp"
connectsTo="ProxyService ServantService" connectedBy="ProxyService
ServantService"/>
    <agents xsi:type="pim:Application" name="ProxyApp"
eventhandler="ProxyAppEventHandler" exchanges="InformationEvent
InformationRequest InformationResponse InformationPetitionEvent
InformationPetitionRequest InformationPetitionResponse"
discoverer="ProxyAppDiscoverer" requestsServices="ServantService
ProxyService"/>
    <agents xsi:type="pim:Service" name="ProxyService"
eventhandler="ProxyServiceEventHandler" exchanges="InformationEvent
InformationRequest InformationResponse InformationPetitionEvent
InformationPetitionRequest InformationPetitionResponse"
discoverer="ProxyServiceDiscoverer" isRequestedBy="ProxyApp"
connectsTo="ServantService ProxyService" connectedBy="ProxyService
ServantService"/>
    ...
  </pim:UbiquitousSystem>
  ...
  <pim:Event name="InformationEvent" exchangedFrom="ServantService
ProxyApp ProxyService" .../>
  ...
  <pim:Event name="InformationPetitionEvent" .../>
  ...
  <pim:Request name="InformationPetitionRequest"
exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker"
conforms="BRProtocol" .../>
  ...
  <pim:Response name="InformationResponse" exchangedFrom="ServantService
ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" .../>
  <pim:SoftwareProtocol name="BRProtocol" system="BlueRose"
rules="BRBroker" .../>
  <pim:NetworkingTechnology name="BRBroker" system="BlueRose" ...
conforms="BRProtocol"/>
  ...
</xmi:XMI>

```

Figure 6.4: Some structural elements, represented in XMI notation, of the US-PIM that results from the transformation of the BlueRose CS-CIM with the proposed QVT rules

- The petition of information from the proxy application to the servant. Note that the confirmation of the petition is not depicted in the sample figure. The reason is that in the US-PIM model, each request has an associated response, and its deliv-

ery is implicitly considered as part of the information transaction.

- A notification from the proxy application that activates the following activity to be carried out, as specified in the choreography.

The reason of not providing a direct information provision to the proxy application is that the servant **may require to access other servants** to obtain the required information, as it was illustrated in Figure 6.2. Also, a discovery of the servant is necessary, since the proxy application may be executing in different physical environments, and it might be possible that in some of them the target servant could not be available. The multiple notifications provide information to different software agents about the information requests or supplies that are carried out by the other software agents. This way, BlueRose middleware supports the monitoring of the different activities that are executed in the ubiquitous system.

Other two closely related examples are the **exchange of information between proxy applications, and from a proxy application to a proxy service**. In the first case, to exchange information between proxy applications, an event distribution is only considered, since, as it was specified in Chapter 5, Section 5.3, this is the only way of exchanging information between applications in the pro-

posed US-PIM. However, in the exchange of information between a proxy application and a proxy service, two message exchanging activities are considered: one for requesting the information and another one to provide it. In contrast with the request from a proxy application to a servant, the information supply can be directly carried out by the target proxy service, since it is meant to contain all the information related to its proxy application counterpart.

Finally, the different gateways are treated as events that need to be notified under certain conditions to activate the corresponding activities of the choreography. Figure 6.6 illustrates some of the events supported by BlueRose, and directly related to the gateways of the choreography depicted in Figure 6.2.

```

...
    <choreographies participants="ProxyApp ProxyService ServantService" ...
startingEvent="InitialEvent" endingEvents="EndEvent">
    <activities name="Proxy2ServantRequest0" ...
startingMessages="Proxy2ServantRequest0DefaultStartingEvent
Proxy2ServantRequest0StartingEvent">
    <transactions xsi:type="pim:DiscoveryActivity" ...
sourceParticipant="ProxyApp" targetParticipant="ServantService"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyApp" relatedEvent="InformationPetitionEvent"/>
    <transactions xsi:type="pim:MessageExchangingActivity" ...
sourceParticipant="ProxyApp" targetParticipant="ServantService"
message="InformationPetitionRequest"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyApp"
relatedEvent="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
    </activities>
...
    <activities name="Proxy2ProxyRequest0" ...
startingMessages="Proxy2ProxyRequest0DefaultStartingEvent
Proxy2ProxyRequest0StartingEvent">
    <transactions xsi:type="pim:DiscoveryActivity" ...
sourceParticipant="ProxyApp" targetParticipant="ProxyApp"/>
    <transactions xsi:type="pim:EventDistributionActivity"
sourceParticipant="ProxyApp" relatedEvent="InformationPetitionEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity"
relatedEvent="InformationEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyApp"
relatedEvent="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
    </activities>
    <activities name="Proxy2ProxyRequest1" ...
startingMessages="Proxy2ProxyRequest1DefaultStartingEvent
Proxy2ProxyRequest1StartingEvent">
    <transactions xsi:type="pim:DiscoveryActivity" ...
sourceParticipant="ProxyApp" targetParticipant="ProxyService"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyApp" relatedEvent="InformationPetitionEvent"/>
    <transactions xsi:type="pim:MessageExchangingActivity" ...
sourceParticipant="ProxyApp" targetParticipant="ProxyService"
message="InformationPetitionRequest"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyService" relatedEvent="InformationEvent"/>
    <transactions xsi:type="pim:MessageExchangingActivity" ...
sourceParticipant="ProxyService" targetParticipant="ProxyApp"
message="InformationResponse"/>
    <transactions xsi:type="pim:EventDistributionActivity" ...
sourceParticipant="ProxyApp"
relatedEvent="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
    </activities>
...

```

Figure 6.5: Some behavioral elements, represented in XMI notation, of the US-PIM that results from the transformation of the BlueRose CS-CIM with the proposed QVT rules

```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ...>
  ...
  <pim:Conditional
name="ParallelEndGatewayDistributionActivityConditionalStartingEvent" ...
/>
  <pim:Conditional
name="ParallelEndGatewayDistributionActivityConditionalStartingEvent" ...
/>
  <pim:Conditional
name="Servant2ServantRequest0ConditionalStartingEvent" .../>
  <pim:Default name="Proxy2ServantRequest1DefaultStartingEvent" .../>
  <pim:Default name="Proxy2ServantRequest0DefaultStartingEvent" .../>
  <pim:Default name="Proxy2ProxyRequest2DefaultStartingEvent" .../>
  <pim:Default name="Proxy2ProxyRequest1DefaultStartingEvent" .../>
  <pim:Default name="Proxy2ProxyRequest3DefaultStartingEvent" .../>
  <pim:Default name="Proxy2ProxyRequest0DefaultStartingEvent" .../>
  <pim:Default name="Servant2ProxyProvision0DefaultStartingEvent" .../>
  <pim:Default name="Servant2ProxyProvision1DefaultStartingEvent" .../>
  <pim:Sequential name="EndEventDistributionActivityStartingEvent"/>
  <pim:Sequential
name="ParallelStartGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential name="Proxy2ProxyRequest2StartingEvent"/>
  <pim:Sequential name="Proxy2ProxyRequest1StartingEvent"/>
  <pim:Sequential name="Proxy2ProxyRequest3StartingEvent"/>
  <pim:Sequential name="Proxy2ProxyRequest0StartingEvent"/>
  <pim:Sequential name="Proxy2ServantRequest1StartingEvent"/>
  <pim:Sequential name="Proxy2ServantRequest0StartingEvent"/>
  <pim:Sequential
name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="CloseConnectionGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="CloseConnectionGatewayDistributionActivityStartingEvent"/>
  <pim:Sequential
name="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
  ...
</xmi:XMI>

```

Figure 6.6: An excerpt of the US-PIM in XMI standard notation representing the different conditional events that are delivered to activate the corresponding activities in BlueRose

6.1.3. Implementation

Transformation rules can be applied again to derive US-PSMs from the BlueRose US-PIM. Specifically, several US-PSM for BlueRose have been proposed, in order to support Java, C#, Python and Objective-C programming languages. For these languages the elements in the metamodel will be, respectively, instances of *java.lang.Class*, *System.Type*, *type* and *Class*. UML profiles could be used to map the BlueRose US-PIM to other platforms.

The resulting elements from the transformation to the US-PSM are nearly equivalent to those in the US-PIM. However, **for each class in the US-PIM, there are two classes in the US-PSM.** One class is abstract and the other one represents its implementation (*Impl* classes). This separation clearly differentiates the abstract elements that are directly derived from the PIM from the elements that need to be implemented by means of specific technologies (other middleware, communication protocols, etc.). The result of the transformation from the proxy application, proxy service and servant elements in the US-PIM is represented in Figure 6.7, in standard XMI format.

The adopted target software protocol has been IceP, which is the protocol of ICE middleware (see Chapter 2, Subsection 2.1.4.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" ...>
  ...
  <agents xsi:type="pim:Application"
isAbstract="true" name="AbstractProxyApp"/>
  <agents xsi:type="pim:Application"
extendsFrom="AbstractProxyApp"
name="ProxyAppImpl"/>
  <agents xsi:type="pim:Service"
isAbstract="true"
name="AbstractServantService"/>
  <agents xsi:type="pim:Service"
extendsFrom="AbstractServantService"
name="ServantServiceImpl"/>
  ...
</xmi:XMI>
```

Figure 6.7: The XMI result of transforming the applications and services in the US-PIM to the corresponding elements in the US-PSM

The reason is due to the efficiency of the protocol and the availability of open-source implementations for all the target programming languages [130]. For the BRBroker, a specific networking technology supporting Wi-Fi and BlueTooth standards has been implemented. A detailed description of this broker will be provided in the next section. Finally, the discoverers are assumed to operate according to the IETF's Zeroconf standard (<http://www.zeroconf.org>), whose open-source implementation is also available for the target programming languages.

Additionally, for each programming language, a **code generator** has been used to produce template programming code. In consequence, as a result of applying MUSYC and adopting a set of

specific target computing platforms, an operative middleware technology is obtained, implemented in specific programming languages and with specific communication technologies that are combined in an integrated manner.

Moreover, the BlueRose middleware can support the development of ubiquitous systems using specific technologies. Therefore, it serves as a complement for the final implementation stages of MUSYC. This way, it is shown that **MUSYC is flexible enough to assist in the development of technologies that are able to help in the engineering processes associated with the MUSYC methodology itself**. Thus, it is possible to make proposals to feed back or complement MUSYC, by applying MUSYC itself.

6.2. BlueRose as a Software Framework for the Development of Ubiquitous Systems

As it was mentioned in Chapter 2, Subsection 2.1.4.2, middleware technologies incorporate software frameworks to facilitate implementation tasks. This section makes explicit the software framework that is intrinsic to BlueRose, which may **assist in the integration of different communication-related technologies, and help during the implementation of the structure and behavior of a**

ubiquitous system designed through MUSYC.

The framework is described in the following subsections through the set of *frozen* and *hot spots* (see Chapter 2, 2.1.4.2) that have been identified in BlueRose. Moreover, the core implementation of BlueRose, which was presented in previous section, is enriched and completed with new elements to support the “inversion of control”¹, or to even support the integration of new communication-related technologies to the existing middleware.

6.2.1. Structural Elements

The following subsections describe the elements of the software framework that have been identified to support message exchanging, event distribution and dynamic discovery.

6.2.1.1. Message Exchanging

Integrating different protocols and technologies usually involves higher development complexity and lower software maintainability and reusability. In order to overcome these drawbacks at implementation level, new elements have been incorporated to the initial BlueRose design. These elements are depicted in Figure 6.8

¹Note that the BlueRose design has been extended using MUSYC, but a description of the whole development process is not provided, only the results at US-PIM level

through a UML class diagram.

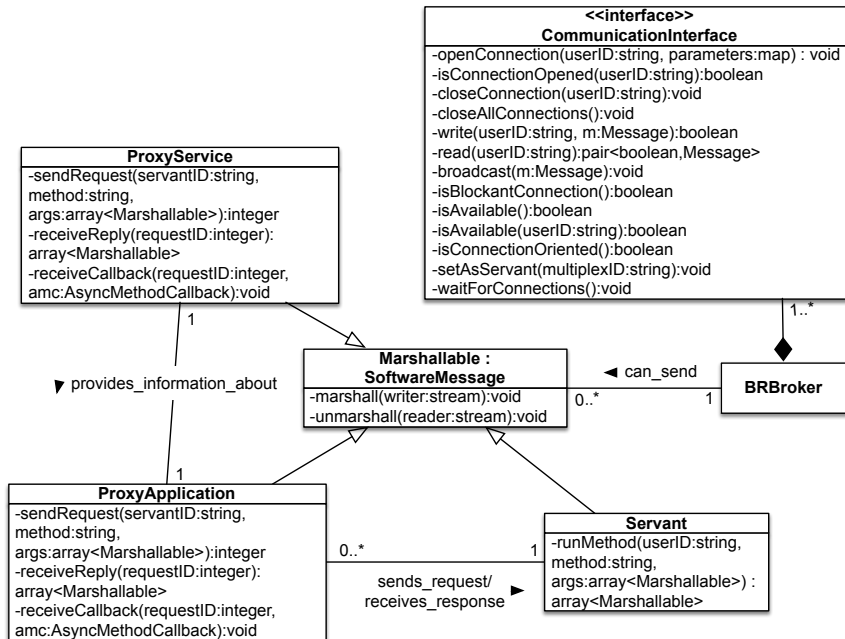


Figure 6.8: New elements incorporated to the BlueRose design and related to the message exchanging functionality, depicted as a UML class diagram

Firstly, a new kind of software message has been incorporated: *Marshallable*. A marshallable message can contain any type of information, without any constraints about the semantics of its contents or delivery mode. This way, BlueRose is not restrained to the use of messages related to events, information requests or information supplies. For example, it can be adapted to provide *documents*, in order to support the **DCM communication paradigm**. Additionally, proxies and servants are considered as specializations of

marshallable messages. This way, it is possible to even **transfer proxies and servants** across the network, in order to, for example, implement **mobility of code**.

Consequently, the integration of multiple protocols involves the specification of different implementations of the messages described in previous section (information petition request, information petition response, information request and information response), plus the specification of the codification format to *marshal* the different proxies and servants. Therefore, in BlueRose, **the messages can be considered as a hot spot**, since they need to be specialized by the developer according to the technical specifications of the adopted protocols.

Moreover, the treatment of the messages as hot spots increases the flexibility of the software framework attached to BlueRose, since it enables the incorporation of different communication protocols into the same software solution, while promoting a lower cohesion between the use of specific protocols and the specific implementations of the proxies and servants.

The BRBroker can be considered as a **frozen spot**, since it provides an unalterable part of the software framework that supports the exchange of messages through heterogeneous networking technologies. Therefore, it is a key element to allow both the integration

and homogeneous management of different networking technologies.

However, a new element, which behaves as a **hot spot**, has been associated with the BRBroker: **Communication interfaces**. These interfaces represent the different networking technologies that can be supported by the broker, and can be dynamically added to it as needed by the developer. Therefore, it is possible to integrate new networking technologies to the current BlueRose base implementation. In consequence, the general behavior of the BRBroker is depicted in Figure 6.9 as a UML sequence diagram, and can be summarized as follows:

1. The BRBroker receives a message from a source software agent (for instance, an application).
2. The attributes associated with the message (operation mode, number of recipients and type of message) are analyzed by the broker.
3. The broker makes use of a specific implementation of a communication interface that is appropriate to deliver the message (e.g., the interface that might be optimized for that type of message).
4. The implementation of the communication interface sends it

to the corresponding receiver/s.

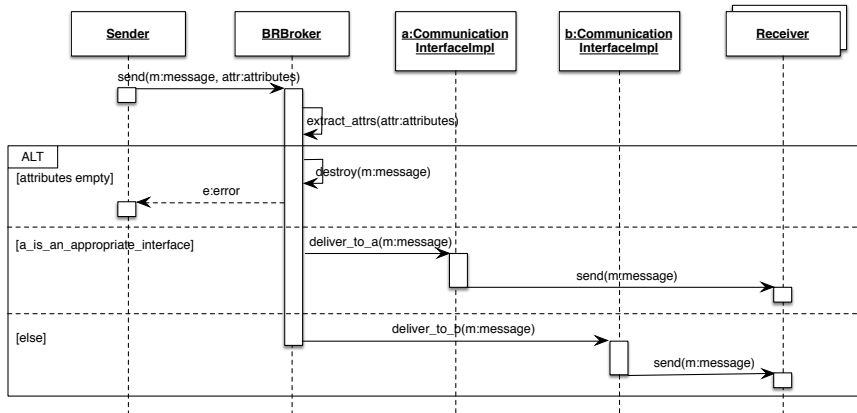


Figure 6.9: Run-time operation of the proposed *BRBroker*, depicted as a UML sequence diagram

Additionally, the *BRBroker* can be considered to provide a point of “inversion of control” to the software to be developed using BlueRose, since it is in charge of appropriately using the implementations of the communication interfaces provided by the developers. The inversion of control is, by the way, a key functionality that should be provided by any software framework (see Chapter 2, Subsection 2.1.4.2).

Finally, both the proxies and servants can be considered as **hot spots**, since they are a basis that needs to be specialized and implemented according to the specific requirements of any software that is developed using BlueRose.

6.2.1.2. Event Distribution

In the proposal of the US-PIM metamodel it is specified that each event has associated semantics in the form of instances of *topics*. Therefore, the topic element is a **hot spot** that can be adapted to the requirements of the ubiquitous system to be implemented.

However, in order to retrieve information about the topics, and their associations to the different events that might be distributed across the network, a **semantic servant** has been added to the initial design of BlueRose. This servant is represented in Figure 6.10 as a UML class diagram. As it is possible to observe in that figure, the servant has a public interface that allows to retrieve information about the hierarchy of the topics and their semantic properties (their constraints, relations to other topics, etc.). In consequence, it is a **frozen spot**, since it has a fixed behavior that is not meant to be modified by developers.

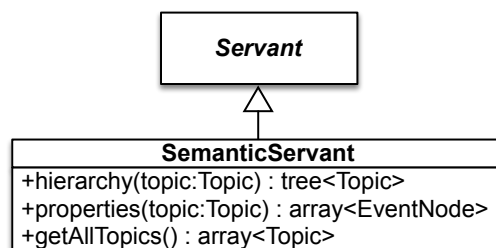


Figure 6.10: The *semantic servant* present in the BlueRose design, depicted as a UML class diagram

Events are also **hot spots**, since there can be specialized im-

plementations to deal with the notification of any important occurrences related to the execution of the ubiquitous system to be developed. Nonetheless, the event handler is a **frozen spot** that manages the delivery of each event and the subscriptions of the proxies or servants to the notification of certain events.

As it has been mentioned before in Chapter 4, Section 4.2, the subscriptions are modeled in the US-PIM through event listeners. Thus, the different event listeners are **hot spots** whose implementations can be specialized to carry out any specific actions whenever a specific event is received. Similarly, the predicates related to each event listener are considered hot spots that represent any conditions associated with each event listener, so as to filter the events to be received by that event listener.

In consequence, the event handler provides “inversion of control” in relation to the event listeners and predicates, since it is in charge of executing the appropriate event listeners whenever their associated predicate is satisfied.

6.2.1.3. Dynamic Discovery

In ubiquitous systems, *how* software agents are discovered depends specifically on the underlying technology (i.e., if it supports broadcasts and/or if it provides a method for discovering nearby

software agents, like BlueTooth or Wi-Fi). Therefore, the discoverers are considered as **hot spots** in the software framework. Moreover, the discovery listeners need to be specialized to carry out the actions that are meant to implement in each concrete ubiquitous system to be developed. Hence, they are also **hot spots**.

6.2.2. Behavioral Elements

The software agent choreography of BlueRose is a **frozen spot**, since it consists on the execution of the different activities that were described in previous section. Nonetheless, the communication activities associated with the choreography can be specialized to carry out the different tasks related to the different specializations of the proxies and servants that may exist in the ubiquitous system to be developed. Anyhow, the elemental communication activities are meant to be fixed (they always behave as it was described in the previous section) and, consequently, they are **frozen spots**.

The choreography also provides “inversion of control” in relation to the activities to be executed in a ubiquitous system. Thereby, the software framework defines through the choreography a systematic way of executing the different activities that are associated with the ubiquitous system to be developed.

Finally, the gateways are **hot spots**, since they are meant to

be specialized to activate the different communication activities that are implemented by the developer.

6.3. Quality Attributes of BlueRose

The quality attributes of BlueRose have been identified in the following subsections using the international quality standard **ISO/IEC 25010:2011** [54]².

In the first subsection, the performance efficiency of BlueRose is measured through **quantitative metrics**, and by comparing the results with some well-known middleware solutions. In the last subsection, the level of accomplishment of some of other quality attributes is described through qualitative observations, since there are no conventions to measure the degree of fulfillment of those quality attributes [23].

6.3.1. Performance Efficiency

The performance efficiency is the degree to which BlueRose provides appropriate performance, relative to the amount of resources used, under stated conditions. It can be quantitatively measured using **benchmarks**, in order to obtain information

²The definitions of the different quality attributes to be mentioned are directly extracted from the ISO/IEC 25010:2011 standard [54]

about the **time behavior** and **resource utilization** of the software. However, benchmarks do not offer absolute measurements, instead, their results need to be compared against the results obtained from applying the same benchmarks to other similar software. In this case, **BlueRose has been compared against ICE and CORBA**, since they are well-known middleware technologies that are comparable to BlueRose in terms of provided functionalities.

The benchmark consists of creating N object proxies and destroying them, in order to test the amount of **memory** that is used by those proxies. After the destruction, one object proxy is created, and N messages containing two numbers are sent to a service. The service returns the sum of both number to the proxy. Afterwards, N messages are again sent to the same service, this time including an string that contains “*hello world ñññáéíóù!!*”. The idea is to test the amount of **time** that is needed to send $2N$ messages, in order to obtain the **average throughput** (messages per second) of each middleware. Moreover, the content of the messages is intended to test the **average CPU use** that is needed to code and decode numbers and complex strings (including non-ASCII symbols), which are very commonly used types of data. Finally the N value has been varied from 1.000 to 100.000, so as to study how the different metrics change when the N is gradually increased.

The benchmark has a similar implementation in all the studied

middleware solutions, and no optimizations have been applied to any of them. The benchmark has been executed, in all the cases, 10 times and in a machine with the following characteristics: Ubuntu 13.10, Intel Core i5 1.8GHZ (dual core) CPU, 8GB RAM and a Solid-State Disk (SSD). Also, the last versions of each middleware (as of March, 2014) have been used, all of them implemented in C++³, and compiled using the same compiler flags for code optimization. Additionally, the tests have been made using the *loopback* communication interface (the communication interface that directly references to the local machine), in order to avoid any mis-measurements related to the status of a networking interface at a specific moment.

The results are presented in Figures 6.11, 6.12, 6.13 and 6.14. As it is possible to observe in those charts, BlueRose has a bigger throughput than ICE and CORBA, thus using less time to send the same amount of messages. However, ICE uses less memory than BlueRose, whose memory footprint is similar to CORBA. In the server side, again, ICE uses less memory, since it requires 432KB to execute the service, whereas BlueRose and CORBA require, respectively, 484KB and 552KB. Finally, BlueRose requires much more CPU than CORBA and ICE, which involves that, probably, the current implementation of the codification and decodification of messages should be optimized in future versions.

³CORBA is an specification, but the OmniORB C++ implementation of that specification has been used for the benchmark

To conclude, BlueRose is comparable in terms of performance efficiency to CORBA and ICE, being better in some aspects (throughput) and worse in others (memory and CPU use). This way, it is possible to assess that MUSYC is able to produce middleware solutions that are, at least, similar in terms of efficiency to other middleware solutions. However, the main benefit of using MUSYC is that the obtained middleware has a direct match with several abstract models whose design can be easily extended in order to accommodate future required functionalities or

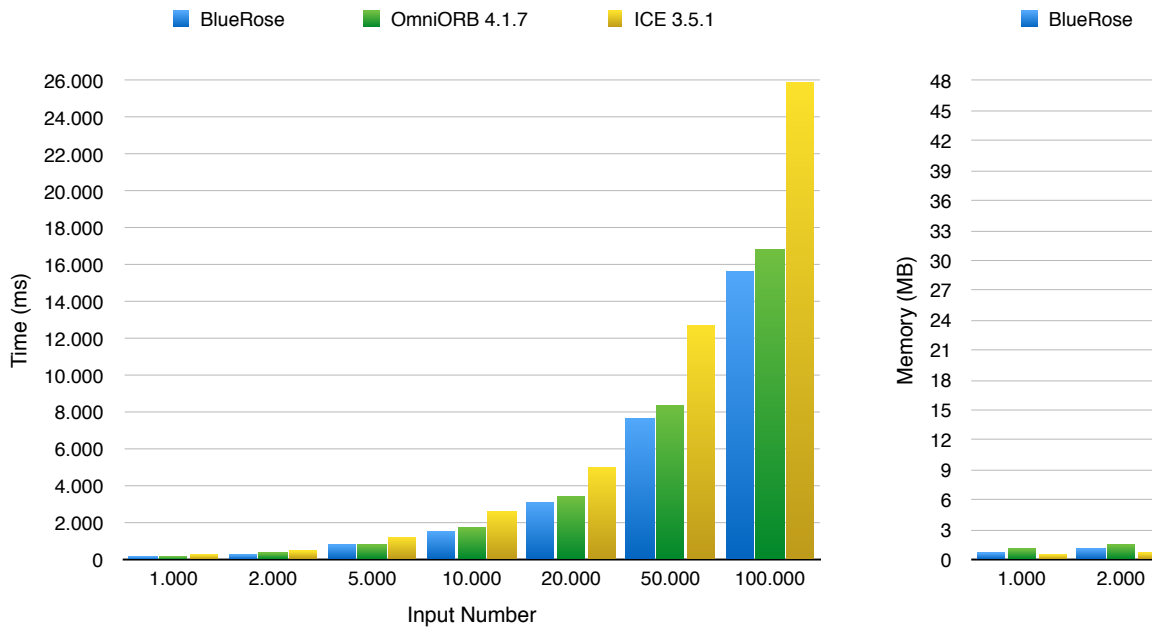
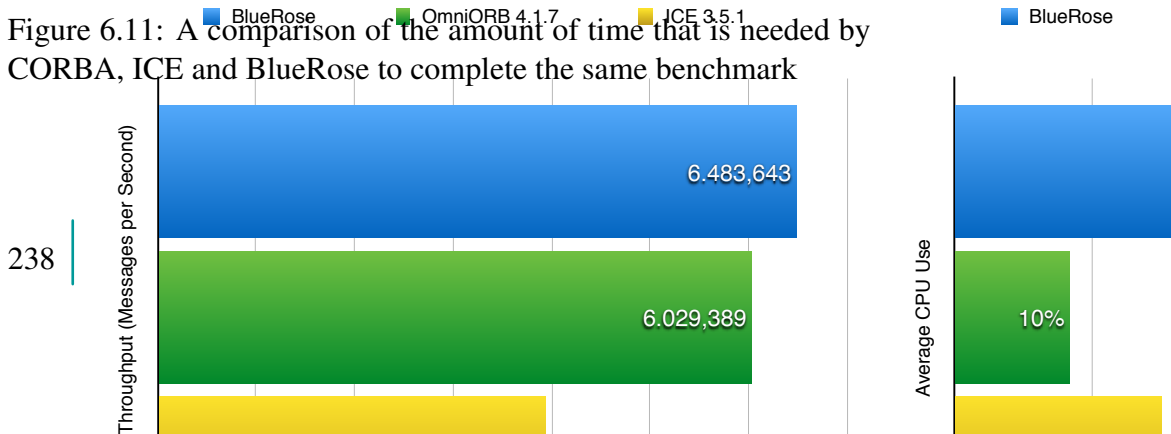


Figure 6.11: A comparison of the amount of time that is needed by CORBA, ICE and BlueRose to complete the same benchmark



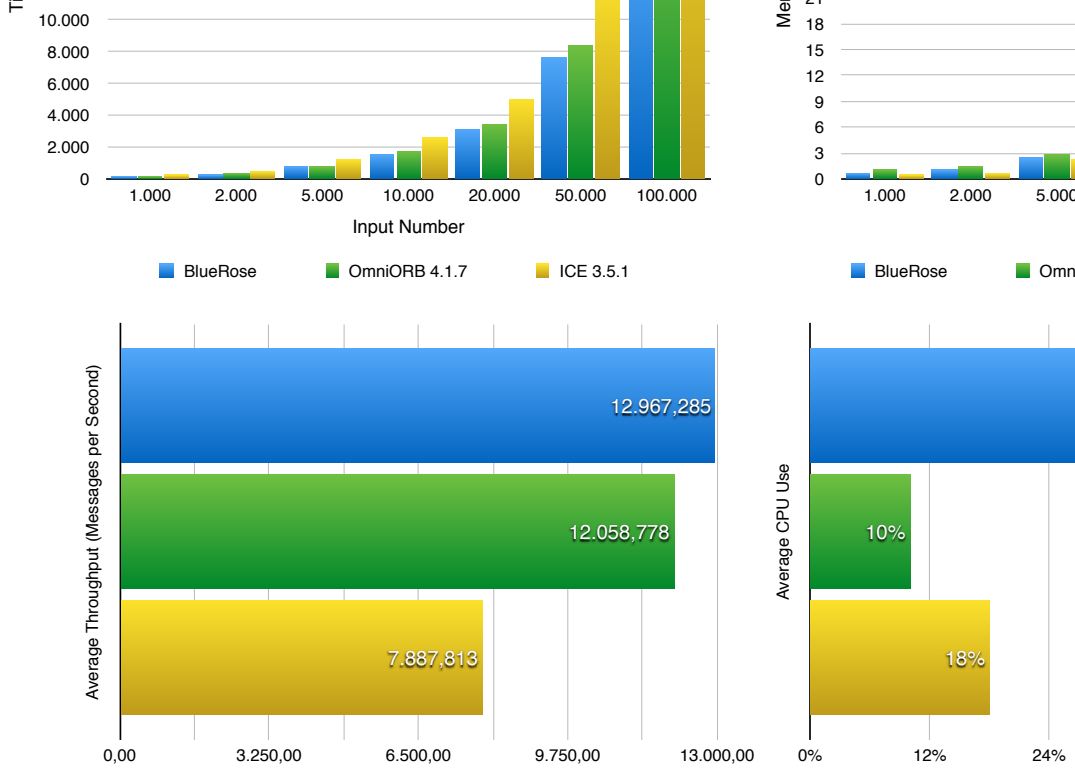


Figure 6.12: A comparison of the average throughputs (messages per second) of CORBA, ICE and BlueRose.

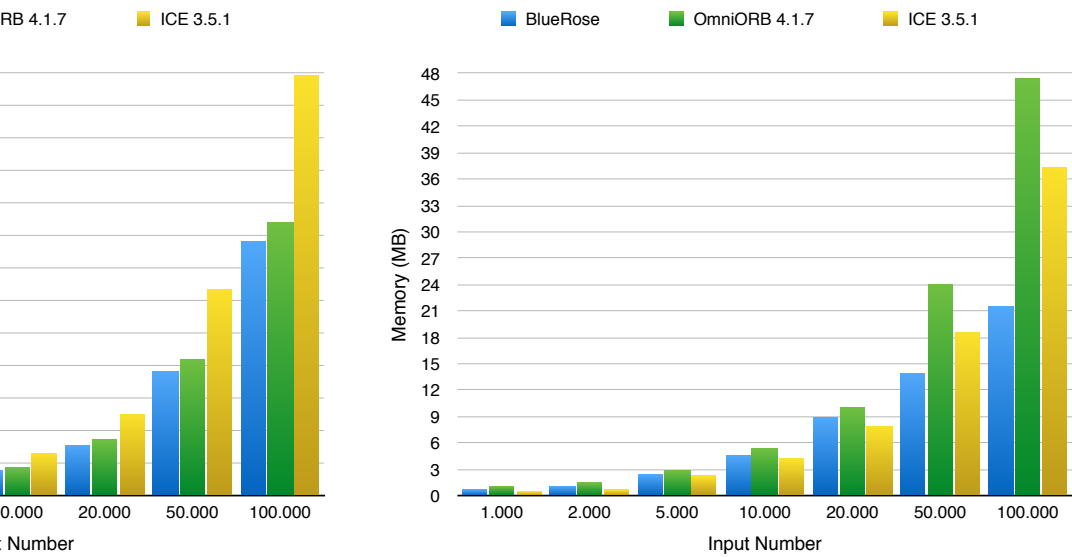
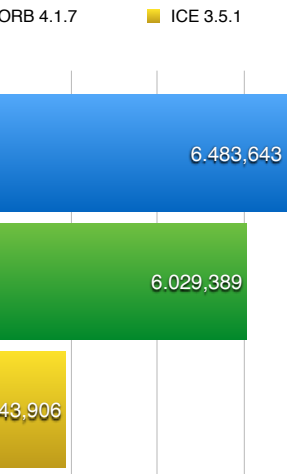


Figure 6.13: A comparison of the amount of memory that is needed by CORBA, ICE and BlueRose to complete the same benchmark.



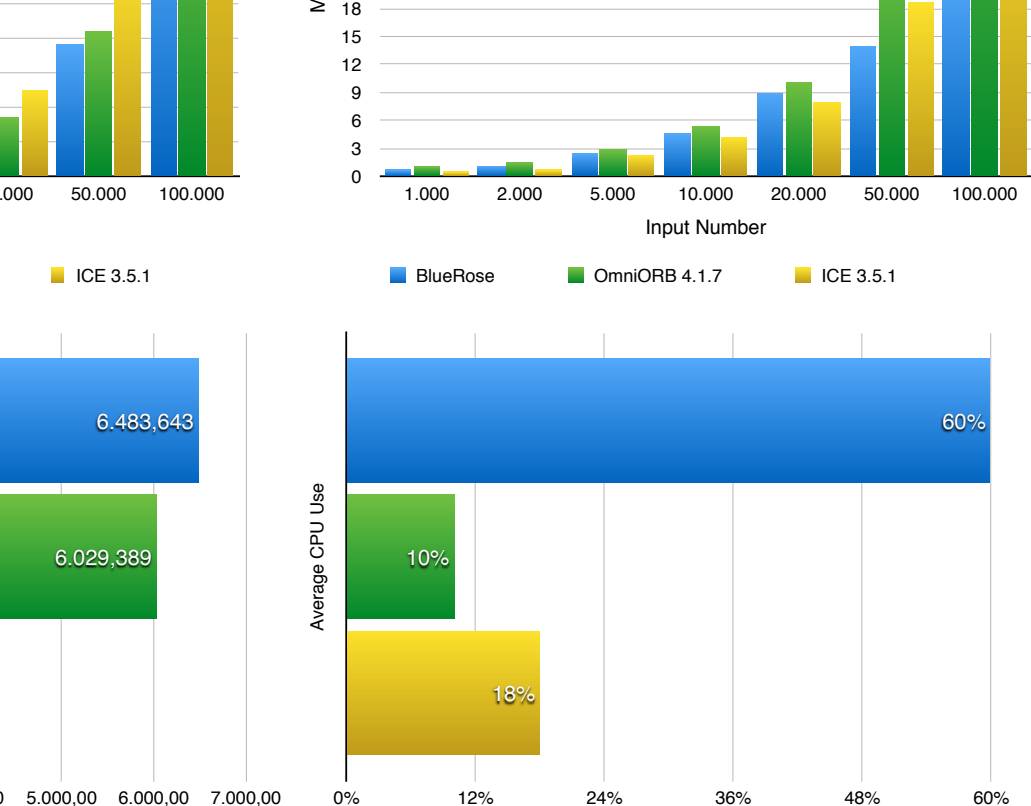


Figure 6.14: A comparison of the average CPU use of CORBA, ICE and BlueRose to complete the same benchmark

6.3.2. Additional Quality Attributes

6.3.2.1. Functional Suitability

The functional suitability is the degree to which BlueRose provides functions that meet stated and implied needs when the software is used under specified conditions. In relation to that, BlueRose is able to provide the communication functionalities of a ubiquitous system (message exchanging, event distribution and dynamic discovery) at implementation level. It also includes mechanisms to implement communication-related activities (i.e., the set of communication activities related to the choreography). Therefore, it is able

to support both the specification of the structure and behavior of a ubiquitous system, which are, ultimately, the objectives that are needed to be accomplished by the developer of a ubiquitous system.

Moreover, the functional suitability of BlueRose is comparable to other existing, well-known and established middleware solutions, like CORBA or ICE, since it provides similar mechanisms to support the communication functionalities that are provided by those middleware solutions.

6.3.2.2. Reliability

The reliability is the degree to which BlueRose can maintain a specified level of performance when used under specified conditions. The accomplishment of this quality attribute is directly related to the **availability, fault tolerance and recoverability** of BlueRose. The availability is the degree to which BlueRose is operational and available when required for use. The fault tolerance is the degree to which BlueRose can maintain a specified level of performance in cases of software faults or of infringement of its specified interface. Finally, the recoverability is the degree to which BlueRose can re-establish a specified level of performance and recover the data directly affected in the case of a failure.

The availability has been taken into consideration even during

the definition of the US-PIM metamodel, since its design pursues the inclusion of distributed elements that are replicated in each application or service to be instantiated. The main benefit of this design choice is that the applications and services supported by BlueRose are not avoided to carry out their tasks in certain scenarios with low or null long-range connectivity possibilities (e.g., no internet connection), which could avoid them to connect to any centralized element. Consequently, the availability of the applications and services developed using BlueRose is increased.

An example of those facts is the *event handler*, which has been defined as a software component that should be replicated in each software agent, instead of being a centralized service, as it is presented in many middleware solutions, like CORBA (the *Event Service*) or ICE (the *ICE Storm* service).

The fault tolerance and recoverability attributes highly depend on the specific implementation of the software supporting the networking technologies and protocols provided by BlueRose. Hence, BlueRose can be considered fault tolerant or recoverable only if the implementations of the networking technologies and software protocols have those attributes. In consequence, BlueRose can only be considered as reliable under those technical conditions. Nonetheless, the current implementation of BlueRose includes mechanisms to support fault tolerance and recoverability at networking technol-

ogy and protocol levels. Thereby, that implementation can be effectively considered as reliable.

6.3.2.3. Operability

The operability is the degree to which BlueRose can be understood, learned, used and attractive to the user, when used under specified conditions. In this case, the design of BlueRose includes several notions, like proxy and servant, that can be found in other middleware solutions, which contributes to its easier comprehension by experts in the middleware field.

Moreover, the BlueRose design is based on the CS-CIM and US-PIM metamodels that were explained in previous chapters. This way, the elements in BlueRose can be easily traced back to some more abstract notions that are present in those metamodels. In consequence, it is easier to understand *why* each element is present in BlueRose and *what* are the objectives of each of them.

Finally, BlueRose contributes to an homogeneous use of heterogenous communication-related technologies, which should be helpful in practice to make an appropriate use of them without tackling with their technical particularities.

6.3.2.4. Security

The security is the protection of system items from accidental or malicious access, use, modification, destruction, or disclosure. It is closely related to the confidentiality, integrity, non-repudiation, accountability (The degree to which the actions of a software agent can be traced uniquely to it) and authenticity.

The security is related to the design and specific implementation of all the elements of the middleware. Therefore, in this case, it is not possible to make any assessments about BlueRose in terms of security, since both secure or insecure protocols and networking technologies could be integrated into it. However, in the current implementation, there is an optional implementation of the BRBroker that encrypts all the messages before sending them according to the different protocols and networking technologies, and checks their authenticity and integrity after receiving them. In consequence, a certain level of security is achieved.

6.3.2.5. Compatibility

The compatibility is the ability of two or more software components to exchange information and/or to perform their required functions while sharing the same hardware or software environment.

BlueRose is able to integrate several implementations of pro-

protocols and networking technologies in order to ensure that the different applications and services can exchange information. Moreover, BlueRose can co-exist and interoperate with other middleware solutions if their associated protocols and networking technologies are integrated into it. In consequence, BlueRose could achieve a good level of compatibility if these technical conditions are satisfied.

6.3.2.6. Maintainability

The maintainability is the degree to which BlueRose can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

The design of BlueRose can be modified or extended by re-applying MUSYC, and modifying the CS-CIM, US-PIM and US-PSM as needed. Moreover, the implementation of BlueRose includes mechanisms to integrate new technologies, that could be intended to support new functionalities, include corrections or improve the behavior of the middleware. Thereby, BlueRose can be considered as maintainable.

6.3.2.7. Transferability

The transferability is the degree to which BlueRose can be transferred from one environment to another. This property is achieved in BlueRose through the fulfillment of the following sub-characteristics:

- **Portability:** The ease with which a system or component can be transferred from one hardware or software environment to another. In BlueRose the portability has been achieved by implementing the middleware using several programming languages, and testing the compatibility of the different codes in multiple operating systems (Windows, MacOSX, Ubuntu Linux, iOS and Android). Moreover, some of the target programming languages include portability mechanisms (i.e., Java, Python and C#), like virtual machines or interpreters, which reinforces the portability of the middleware itself.
- **Adaptability:** The degree to which BlueRose can be adapted to different specified environments without applying actions or means other than those provided for this purpose for the software considered. It is fulfilled by supporting the integration of new implementations of the protocols and networking technologies, which contributes to the use of the middleware in the needed computing environments.

- **Installability:** The degree to which BlueRose can be successfully installed and uninstalled in a specified environment. This quality attribute has been fulfilled by providing compilation, installation (and uninstallation) tools for each programming language and target operating systems. Particularly, in Python, the standard installation mechanisms that this language provides have been used.

6.4. Practical Validation

BlueRose has been used to develop multiple real R&D projects, which shows that this middleware, and its associated software framework, can be effectively used in real implementations of ubiquitous systems. The following subsections provide a summarized description of those projects, which also serve to validate MUSYC in practice.

6.4.1. Mobile Forensic Workspace [100]

6.4.1.1. Overview of the Workspace

Governments and specific police forces (like Interpol) apply official protocols of action intended to support victim identification in different scenarios: natural disasters, accidents, terrorist attacks,

murders, etc. These protocols try to deal with how victim data is collected and how professionals (e.g., members of police forces and forensic experts) have to cooperate. Currently, governments and police forces do not use supporting technologies for both in-situ data collection and cooperation, since it was necessary to fulfill several requirements that were not technologically addressed:

- **Data collection.** Since there are several official protocols intended to support victim identification, it was not possible to specify a uniform data model for collecting and sharing information. Thus, software solutions had to be “customized” for each specific protocol and scenario.
- **Data sharing.** In order to share information in a group or even between different groups (e.g. police and forensic experts), software applications made use of common network infrastructures, like Internet, which may not be available in some scenarios (e.g., maritime accidents, natural disasters, rural environments, etc.).

The Mobile Forensic Workspace (MFW) is a **ubiquitous collaborative system based on mobile technologies**. The aim is to assist forensic experts in collecting *in-situ* data about victims, while overcoming previous issues. In the workspace it is required to ex-

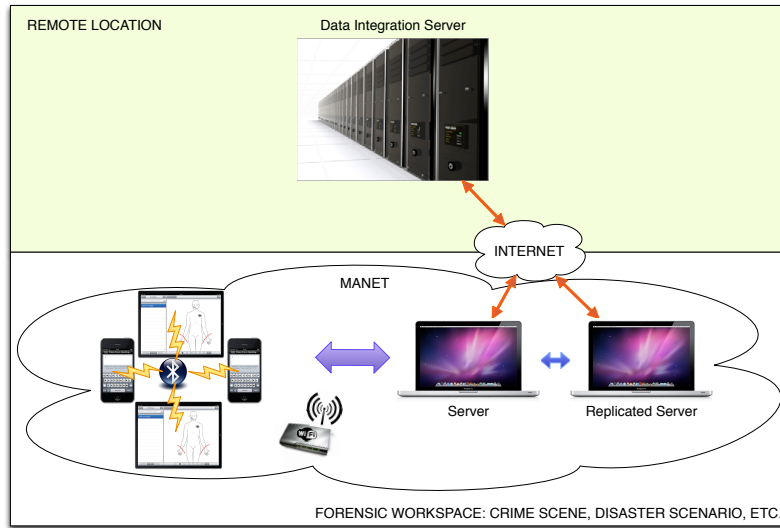


Figure 6.15: Deployment Architecture for Mobile Forensic Workspace

change information with nearby applications, devices, etc. so as to support data sharing between nearby forensic experts.

MFW makes use of the system architecture that is shown in Figure 6.15. It is important to remark that this architecture has been devised in such a way that none of its elements is totally required, except for the devices to execute the application by final users. This way, professionals, will not be limited, in any case, by the availability of a specific underlying communication technology.

In this architecture, devices can exchange information, for instance, by making use of BlueTooth technology. This way, a network infrastructure is not necessary. These devices exchange information with two servers that could ideally be available in location

(as the only basic desirable infrastructure provided by the official institutions that are acting in the disaster, accident, crime, etc.). These servers will always store the last available data (victim information, multimedia, etc.) collected with the mobile devices used by the forensic experts. The two servers are advised to ensure that, in case of a critical failing happens in the main server, the replicated server will be available instead (without losing any data). The mobile devices would mainly exchange information with servers by making use of Wi-Fi since it is a quicker than BlueTooth. However, if Wi-Fi is not available, devices will make use of BlueTooth.

Whenever Internet is available, servers will communicate the information that they store to a remote data integration server, which will depend on the specific official institution that makes use of the MFW. Additionally, if required, *in-situ* servers may communicate their information to more than one remote server (e.g., more than one official institution, replicated servers, etc.).

6.4.1.2. Collaboration Capabilities of MFW

MFW supports several functionalities related to communication and collaboration between experts in a specific forensic scenario (crime scenes, natural disasters, multiple-victim accidents, and so on):

- **Informal communication.** It will be supported by mechanisms to carry out voice-calls and messaging between forensic experts.
- **Document authoring.** Collected information about victims is associated with an author and a time stamp. Documents (texts, images, videos, etc.) and their authoring information are available for every expert in a specific forensic scenario.
- **Concurrent document modification.** Forensic experts are allowed to modify the same document at the same time and to observe other' changes in real-time.
- **Automatically synchronized changes.** Each change to a document is automatically (and transparently) synchronized between forensic experts, even if they lose their connection and they re-connect after a while.
- **Security and privacy of shared information.** Some rules are established for specifying access control, how and which data is exchanged, etc.

These functionalities should not rely on specific communication technologies or mechanisms, since the MFW aims to be adaptable to the requirements of any official authority, and to be ubiquitously available in any physical scenario (e.g., in locations without

a connectivity or any kind of networking infrastructure).

In consequence, **BlueRose** has been adopted as the supporting technology to implement those functionalities, which involve message exchanging, event distribution and dynamic discovery, as it is expected in ubiquitous systems. BlueRose also allows to easily integrate networking technologies and communication protocols, as needed by each specific official authority (i.e., each authority may use different protocols and allow or disallow the use of specific networking technologies for security reasons).

6.4.1.3. Implementation of MFW

The MFW has been developed for **iOS devices** (iPhone, iPad and iPod Touch). Several screenshots of the resulting mobile application are shown in Figure 6.16. In that figure, several functionalities of the MFW that have been technically supported by BlueRose are depicted: (A) shared information about a victim; (B) authoring information about a data entry; (C) a shared whiteboard for drawing an outline of a body description (tattoos, body marks, etc.) over several body pictures (front, back, top, both sides, hands, etc.); and (D) a text and voice chat.

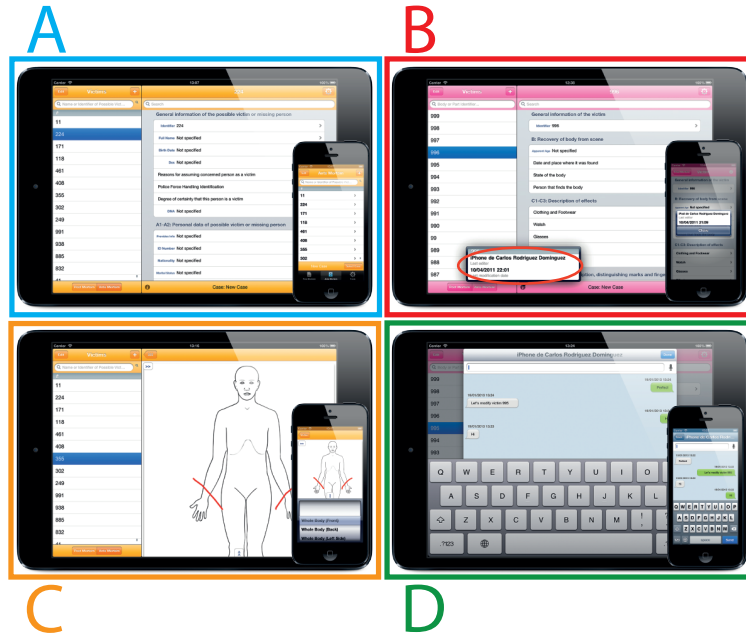


Figure 6.16: The Mobile Forensic Workspace in iOS devices

6.4.2. VIRTRA-EL: A Web Platform to Support a Collaborative Virtual Training for Elderly People [102]

6.4.2.1. Overview of the Requirements

VIRTRA-EL is a web platform to support a collaborative virtual training for elderly people. It mainly consists of a set of exercises intended to improve the cognitive skills of the users, under the supervision of psychologists. Additionally, this platform supports:

- Remote cognitive training and evaluation of elderly people.
- User profiles to configure the difficulty level of the exercises,

and to facilitate their adaptation to the specific culture, interests and needs of the end users.

- Ubiquitous monitoring of how the users realize the different exercises.
- Collaborative exercises intended to promote the socialization and communication between the users.
- Communication tools to support the inter-personal relationships between the users and the psychologists, and to obtain feedback about the platform.
- Statistical tools to track and compare the progress of the users (difficulty levels, cognitive aspects, the amount of time that is spent in each exercise, etc.).
- Virtual and augmented reality exercises.
- Fulfillment of, at least, the following quality properties: security, scalability, portability, usability and extensibility.

To meet the previous requirements, a component-based architecture has been devised. It is depicted in Figure 6.17. The focus of the next subsection is to explain the collaboration component, which is the one that has been implemented using BlueRose.

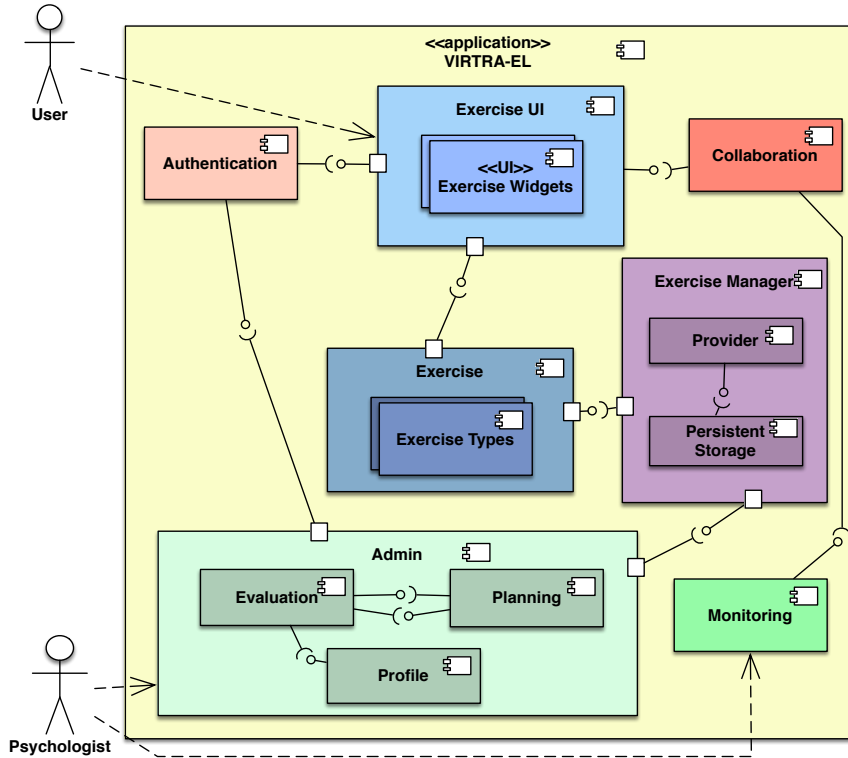


Figure 6.17: Component-based architecture of VIRTRA-EL

6.4.2.2. The Collaboration Component of VIRTRA-EL

The collaboration component of VIRTRA-EL allows the realization of collaborative exercises, and has been implemented using BlueRose. This component is in charge of synchronizing the state of multiple instances of the same graphical widget in the same exercise, but in different computers. The widgets are the graphical elements that are presented to the user whenever they carry out an exercise. This way, multiple users will be able to make use of the same widgets at the same time, observing in real-time how the other

users interact with those widgets.

The collaborative component is only used in selected exercises that have been designed to be completed through the collaboration of multiple users. In VIRTRA-EL, the users can collaborate in the following ways:

- Concurrently, to complete different tasks of the same exercise.
- Through a turn-based scheme.
- Some users interact with the exercise (using any of the previous collaboration ways), while others observe their interactions and communicate with the first group in real-time to assist them in the completion of the exercise.
- Psychologists and users can communicate through chats to exchange any ideas, or to provide and request help about an exercise.

6.4.3. Domo and Kora: Management of Home Automation Environments [99]

Domo is a service for managing sensors and actuators in a ubiquitous home automation environment, independently of the underlying standard (X.10, LonWorks, etc.). Kora is its companion

mobile application, which interacts with Domo to allow the user management of home automation environments.

In Figure 6.18 it is depicted a scheme of Kora, which is a mobile application for Android operating systems, interacting (through BlueRose) with the Domo service in order to manage a set of home automation devices.

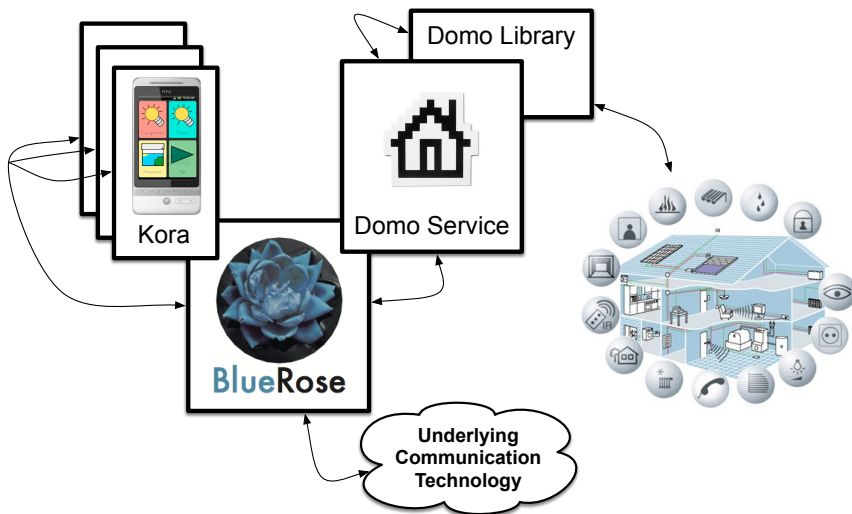


Figure 6.18: An iPhone application acting as a client of the Domo service

Note that Domo and Kora were developed with a primitive version of BlueRose, whose main limitation was that it only supported event distribution, thus making it not possible to establish synchronous and direct connections between the application and the service. The feedback that was received during the implementation of this software enriched the design of the US-PIM and, conse-

quently, BlueRose.

Since message exchanging was not supported, the Domo service is not an instantiation of the *servant* hot spot of the identified software framework associated to BlueRose. In this case, it is an application that acts as both a publisher and a subscriber to a set of events.

Each event was published whenever a home automation device changed its internal state. For example, if a light bulb was switched on, then the service published an event indicating the new state of that light bulb. Since Kora was designed to be subscribed to these events, several instantiations of this mobile application may be concurrently executing in multiple devices. Each instance of Kora receives the corresponding state changes, independently of whether these changes result from the interaction between the user and another instantiation of Kora, or from the physical interaction between the user and the home automation device. Consequently, Kora provides a graphical interface to show the environment state in real-time.

Finally, Kora makes use of an adaptable user interface that was specially designed for people with special needs, and with different profiles. Figure 6.19 illustrates two sample adaptations of its user interface.



Figure 6.19: Two sample adaptations of the Kora user interface for different users

6.4.4. Sherlock: A Location Service for Both Outdoors and Indoors [104]

Sherlock is a location service that was developed using BlueRose. It was designed to be incorporated into ubiquitous systems to allow both outdoors and indoors positioning.

Sherlock supports the integration of several technologies like GPS, for outdoor positioning, and Wi-Fi or ZigBee, for indoors positioning. It also allows several methods for calculating the position of a user. For example, if Wi-Fi is used, then a triangulation algorithm is applied.

In Figure 6.20 it is shown the general architecture of the ser-

vice. As it is depicted in that figure, BlueRose is used to support the exchange of location information between Sherlock and any client applications. In that sense, a client application was also developed with the technical support of BlueRose. The application was developed in Java for mobile phones using **Android** operating system.

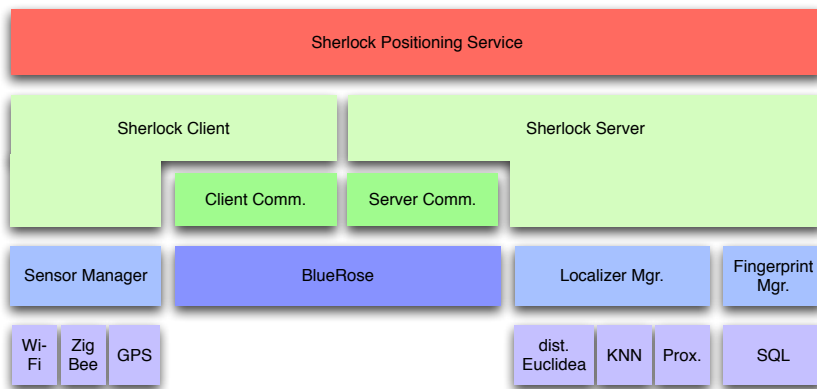


Figure 6.20: Overview of the architecture of the Sherlock positioning service

6.5. Conclusions

In this chapter, it is described the MUSYC-based development of a middleware for ubiquitous systems, called BlueRose. The development of this middleware shows that **it is possible to develop reusable supporting technologies using MUSYC that may help during the implementation stages of different ubiquitous systems.**

BlueRose has been designed to be easily extended with new communication-related technologies, also **integrating** them in an homogeneous manner. The elements comprising the design of the middleware have been identified as frozen or hot spots, or as mechanisms of “inversion of control”, so as to make explicit the **software framework** that is related to BlueRose. This way, the aim has been to show in practice that the design of the middleware, using the proposed CS-CIM and US-PIM metamodels as a foundation, is flexible and complete enough to guide a part of the implementation process that is expected to be carried out during the final implementation stage established in MUSYC itself. Besides, BlueRose can be considered as a flexible and adaptable middleware that could be used as a reusable target platform during the specification of the US-PSM of a ubiquitous system to be developed using MUSYC.

Additionally, the different **quality properties** of BlueRose have been analyzed and contrasted with those present in CORBA and ICE, which are two well-known and used middleware technologies. Particularly, in terms of performance efficiency, a **benchmark** has provided metrics that enable a quantitative comparison between the implementations of BlueRose, CORBA and ICE. The result is that **BlueRose is comparable in terms of efficiency** to those other middleware technologies.

Therefore, **the model-driven engineering methodology en-**

couraged by MUSYC does not necessarily have to affect the performance of the resulting software, even if part of the implementation of that software is automatically done through code generators. Moreover, in contrast with a code-centric development, the specification of CS-CIMs, US-PIMs and US-PSMs facilitates future modifications to the core design of the ubiquitous systems to be developed through MUSYC, makes it easier to understand all the devised elements in the design and to integrate existing and future technologies in a more systematic manner.

Finally, it is worth to be noted that BlueRose has been used during the development of several **R&D projects**: A mobile forensic workspace (MFW), a Web platform to support a collaborative virtual training in elderly people (VIRTRA-EL), a service and a companion application to manage home automation environments (Domo and Kora), and a location service for both indoors and outdoors (Sherlock). In this way, MUSYC has been validated in practice, both to develop ubiquitous systems and supporting technologies (like middleware) to facilitate their implementation.

Chapter 7

Conclusions

This chapter presents the conclusions drawn from the thesis work described herein. The different results are presented and discussed, and some lines of future work are proposed.

7.1. Results and Discussion

The idea of this thesis work has been to show that it is possible to manage the communications in ubiquitous systems through the specification of highly abstract models (CS-CIM and US-PIM), and their systematic transformation into more concrete models targeting specific computing platforms (US-PSM).

As a result, it is possible to state that the management of the communications is not a technical-only problem, but an issue that is

directly related to the whole analysis and design of the ubiquitous system to be developed and the functionalities to be provided, as it was hypothesized in the introduction of this thesis work.

In the following subsections the conceptual, methodological and technical results of this thesis work are discussed.

7.1.1. Conceptualization

This thesis work has introduced two **conceptual models** to expose the notions (and relationships between them) that are present in any communication system, and more particularly, in any ubiquitous system. Hence, the models conceptualize **Computations-Independent Communication Systems (CI-CS)** and **Platform-Independent Ubiquitous Systems (PI-US)**.

The structural view of the CI-CS metamodel captures all the structural elements that are present in a communication system, extending and completing the most well-known and accepted **communication theories**. The behavioral view includes the organizational elements that should be present in a communication system. The organization of a communication system has been devised as a “choreography”, that is, as an ordered sequence of interactions between the participants of the communication. The BPMN 2.0 choreography metamodel has been taken as a reference to figure out the elements

and relationships that should be present in the behavioral view.

Similarly, the structural view of the PI-US metamodel captures all the structural and basic operational software artifacts that should be present in a ubiquitous system to support **message exchanging, event distribution and dynamic discovery**. To devise the metamodel, some standard protocols, networking technologies, communication paradigms and well-known middleware technologies have been analyzed.

The behavioral view of this metamodel includes elements to represent the organization of the communication in a ubiquitous system a set of organized communication activities involving transactional exchanges of messages, events and discoveries. The behavioral view of the PI-US metamodel is based on the behavioral view of the CI-CS metamodel. Nonetheless, it takes into account the software focus at this abstraction level, which involves relating all the elements present in the view to software artifacts.

Both the CI-CS and PI-US conceptualizations have been semantically formalized through **ontologies**. This way, it is possible to check if a designed CI-CS or PI-US is *well-formed*, that is, if a model contains all the necessary concepts and relationships to be considered, respectively, in a communication system or a ubiquitous system (i.e., if a given model is consistent with the proposed con-

ceptualizations).

Furthermore, the ontology of a CI-CS has been extended by incorporating some key notions present in the PI-US metamodel. This semantic extension has allowed to formally define **a ubiquitous system as a software communication system with specific requirements and supporting concrete communication functionalities.**

On the basis of the international quality standard for software systems defined in **ISO/IEC 25010:2011**, a qualitative description of the proposed metamodels has been provided. The ontologies assist in the quality analysis, since they allow to assess some of the qualitative features of the metamodels.

Finally, it is important to highlight that the PI-US conceptualization provides a **system view** of the ubiquitous systems, but it does not offer an internal view of the different applications and services that could encompass these systems. Therefore, the conceptualization of the other aspects of a ubiquitous system (user interfaces, data models, etc.) could contribute to a more complete formalized conception of a ubiquitous system, which could help to tackle with the design of this type of systems in a holistic way (e.g., taking into account the communication and presentational aspects, the data models, and the relationships between them).

7.1.2. Methodology

The conceptual models have been respectively re-interpreted as the **metamodels** of a **Computation-Independent Model of a Communication System** (CS-CIM) and a **Platform-Independent Model of a Ubiquitous System** (US-PIM), in order to be used in an **MDA-based methodology to develop Ubiquitous SYstems on the basis of the Communications** (MUSYC). Therefore, the CI-CS and PI-US conceptual models have been used as a conceptual framework on which the whole MUSYC methodology is founded.

The methodology proposes the specification of a brief use case model and a BPMN 2.0 Choreography model. On their basis, a CS-CIM is specified. Then, the specification of the CS-CIM can be systematically transformed into the specification of a US-PIM through a set of proposed QVT rules. Afterwards, a set of target platforms (programming languages, middleware, operating systems, etc.) have to be adopted in order to specify a US-PSM. Finally, the US-PSM can be converted into code using code generators implemented on the basis of the MOFM2T standard.

The whole development process encouraged by MUSYC avoids the adoption and use of specific technologies until the specification of the US-PSM. In consequence, if more advanced technologies become available during the development of a

ubiquitous system, or even later, the CS-CIM and US-PIM designs could be completely reused.

Moreover, **the technologies do not guide the whole development process from the initial stages**, as it commonly occurs in many current developments. This way, **the focus is on the specification of a design that fulfills with the user requirements**, rather than on the adoption or use of concrete technologies. In fact, MUSYC proposes a direct match between the requirements specified through a use case model and BPMN choreographies, and the CS-CIM specification.

To assist in the development of ubiquitous systems through MUSYC, and to show the feasibility of **automatizing and validating** the transformation from the specification of a CS-CIM to executable code for specific target computing platforms (operating systems, middleware, etc.), a set of **CASE tools** have been implemented.

Finally, MUSYC shows that **it could be possible to approach the whole development of a ubiquitous system by focusing on the design of the communications**. For example, a MUSYC-based development produces an initial design of the applications and services to be developed (i.e., MUSYC does not tackle with the design of the user interface, data model, etc.), and the interactions between

them. Therefore, it is possible to establish that **the communications should be considered as a central part of any ubiquitous system development**: the specification of the mechanisms supporting the communications should not be approached during the implementation stage, since they are a fundamental part of the system that even constrains the functionalities and quality properties that can be provided to the end user.

7.1.3. Technology

BlueRose is a middleware that has been developed using MUSYC. This middleware validates that MUSYC, in addition to facilitate the development of ubiquitous systems, enables the development of reusable supporting technologies that may help during their implementation.

BlueRose has been designed to be easily extended with new communication-related technologies, also **integrating** them in an homogeneous manner. This way, it is shown that the design of the middleware, using the proposed CS-CIM and US-PIM metamodels as a foundation, is flexible and complete enough to guide a part of the implementation process that is expected to be carried out during the final implementation stage established in MUSYC itself. Besides, BlueRose can be considered as a highly generic middleware

that could be used as a target platform during the specification of the US-PSM of a ubiquitous system to be developed applying MUSYC.

The elements comprising the design of the middleware have been identified as frozen or hot spots, or as mechanisms of “inversion of control”, so as to expose a **software framework associated to the middleware itself**. This way, it is specified how the middleware can be used during the development of a ubiquitous system and the elements that can be adapted to the fulfillment of specific functional or non-functional requirements.

In addition, BlueRose has been compared against CORBA and ICE in terms of performance efficiency, since they well-known and established middleware solutions. The result is that **BlueRose has a similar performance**, even if part of its implementation has been automatically generated from a US-PSM. Additionally, a qualitative description of BlueRose has been provided, in order to make explicit its different characteristics.

Therefore, **the model-driven engineering methodology encouraged by MUSYC does not necessarily have to affect the performance of the resulting software, even if part of the implementation of that software is automatically done through code generators**. Moreover, in contrast with a code-centric development, the specification of CS-CIMs, US-PIMs and US-PSMs facilitates future

modifications to the core design of the ubiquitous systems to be developed through MUSYC, which makes it easier to understand all the devised elements in the design and to integrate existing and future technologies in a more systematic manner.

Finally, some R&D projects in which BlueRose has been applied serve to validate the proposal in practice: the mobile forensic workspace (MFW), a Web platform to support a collaborative virtual training in elderly people (VIRTRA-EL), a service and a companion application to manage home automation environments (Domo and Kora), and a location service for both indoors and outdoors (Sherlock). Furthermore, the development of these projects make it possible to assess the practical feasibility of MUSYC, both to develop ubiquitous systems and supporting technologies (like middleware) to facilitate their implementation.

7.2. Future Work

Several lines of future work are currently being explored.

The metamodels and the ontologies could allow the **analysis** of ubiquitous systems [9], which would be helpful to check their behavior before finalizing their development. Moreover, simulation tasks could be included into MUSYC. Additionally, the ontologies could be used to model, check for the consistency and simplify spe-

cific CS-CIMs or US-PIMs.

Also, the metamodels could be extended to include **Quality of Service (QoS)** attributes, so as to design ubiquitous systems in which a certain networking performance has to be assessed (e.g., real-time communications, critical systems, etc.). This extension could help into assessing the **dependability** (i.e., according to ISO/IEC 25010:2011, the reliability, fault tolerance, recoverability, integrity, security, maintainability, durability and maintenance support) of the ubiquitous systems that are developed using MUSYC.

Moreover, a **quality framework** could be devised to measure the quality of the specified CS-CIMs, US-PIMs and US-PSMs during a MUSYC-based development. That quality framework could be based on some of the research works mentioned in [77], which presents a survey about the some of the existing approaches to analyze the quality of a model.

Finally, the re-design and integration of existing ubiquitous systems is a challenge that can be addressed through **model to metamodel transformations** [58]. To overcome that issue in a systematic manner, a **reverse engineering** methodology addressing the reverse transformation from a US-PSM to a CS-CIM could be devised. The ontological representations of both a CI-CS and a PI-US may

serve to formally trace back certain notions present in a PI-US to the more abstract level of a CI-CS. Therefore, these representations could help into analyzing the design of existing ubiquitous systems in order to devise these future reverse engineering methodologies.

Acknowledgements

This thesis work is funded by the Innovation Office from the Andalusian Government through the project P10-TIC-6600.

Also, during the elaboration of this work, all the authors have participated in the following projects:

- TIN2012-38600. Spanish Ministry of Economy and Competitiveness with European Regional Development Funds (FEDER).
- TIN2008-05995/TSI. Spanish Ministry of Education and Science.
- 20F2/36. CEI-BioTIC (University of Granada).
- SIMFO. Telvent Interactiva S.A and the University of Granada.
- “Desarrollo de Software para la Estimulación Cognitiva”. University of Granada and the Andalusian Government.
- “Entorno software para el desarrollo y la evaluación de habilidades comunicativas y de competencias de trabajo en grupo mediante debate virtual”. University of Granada.
- “Extensión de la Plataforma de Debate Virtual para Dar Soporte al Análisis y Mejora de la Usabilidad y Evaluación del Alumnado”. University of Granada.
- “Plataforma de soporte a la coordinación y mejora de comunicación docente para la mejora de la calidad en la Educación Superior”. University of Granada.

Publications

1. M. V. Hurtado, M. L. Rodríguez, M. Noguera, K. Benghazi, C. Rodríguez Domínguez. An innovation perspective for the creation of teaching/learning environments based on groupware applications. Proceedings of the V International Conference on Multimedia and ICT in Education (M-ICTE). Edited by Formatex, ISBN (Vol. II): 978-84-692-1790-0, pp. 694-698. 2009.
2. Á. Fernández, C. Rodríguez Domínguez, M. J. Rodríguez. Diseño de una plataforma móvil de apoyo al aprendizaje cooperativo en educación especial. Actas del X Congreso Internacional de Interacción Persona-Ordenador. ISBN: 13:978-84-692-5005-1. 2009.
3. A. B. Pelegrina, C. Rodríguez Domínguez, J. L. Garrido, M. L. Rodríguez, M. Bermúdez. Un Marco de Trabajo de Soporte a la Integración de Aplicaciones Groupware. X Congreso Internacional de Interacción Persona-Ordenador. ISBN: 13:978-84-692-5005-1. **3rd best paper prize.** 2009.
4. K. Benghazi, M. L. Rodríguez, M. Noguera, J. L. Garrido, C. Rodríguez Domínguez. Diseño de aplicaciones groupware adaptables para fomentar el aprendizaje colaborativo en el EEES. ISBN: 978-84-692-7263-3. Jornadas Andaluzas de Innovación Docente Universitaria. 2009.
5. C. Rodríguez Domínguez, A. B. Pelegrina, J. L. Garrido, M. L. Rodríguez, M. Bermúdez. Diseño e Implementación de Software Distribuido de Soporte a la Integración e Interoperatividad de Aplicaciones Groupware. Revista Avances en

- Sistemas e Informática Vol. 7, No. 1. ISSN 1657-7663. 2010. **Indexed in DBLP, DOAJ, Dialnet, Periódica and PASCAL.** 2010
6. C. Rodríguez Domínguez, K. Benghazi, M. Noguera, J. L. Garrido. Redefinable Events for Dynamic Reconfiguration of Communications in Ubiquitous Computing. 1st International Workshop on Data Dissemination for Large scale Complex Critical Infrastructures (DD4LCCI). ACM Press, ISBN: 978-1-60558-917-6, pp. 17-22. 2010. **Indexed in ACM Digital Library. Quality publisher.**
 7. A. Belén Pelegrina, C. Rodríguez Domínguez, J. L. Garrido, M. L. Rodríguez, M. Bermúdez. Integrating Groupware Applications into Shared Workspaces. Proceedings of the 4th International Conference on Research Challenges in Information Science (RCIS). Niza, Francia. ISBN 978-1-4244-4840-1, DOI 10.1109/RCIS.2010.5507305, pp. 557-568. 2010. **Indexed in ISI PROCEEDINGS, DBLP, SCOPUS. Quality publisher. Computer Research and Education index (CORE), category B.**
 8. K. Benghazi, M. Noguera, C. Rodríguez Domínguez, Ana Belén Pelegrina, J. L. Garrido. Real-Time Web Services Orchestration and Choreography. 6th International Workshop on Enterprise and Organizational Modeling and Simulation (EOMAS 2010). ISBN: 978-1-4503-0463-4, pp. 142-153. 2010. **Celebrated with the International Conference On Advanced Information Systems Engineering (CAISE 2010), indexed in Computer Research and Education index (CORE), category A. Published by ACM Digital Library (Quality publisher).**
 9. C. Rodríguez Domínguez, Á. Fernández, J. Alcalá-Correa, M. J. Rodríguez-Fórtiz, J. L. Garrido. Una Propuesta de Diseño para la Integración e Interoperabilidad de Aplicaciones para Personas con Necesidades Especiales. XI Congreso Internacional de Interacción Persona-Ordenador. ISBN: 978-84-92812-52-3, pp. 401-410. 2010.

10. S. M. Gómez, M. L. Rodríguez, K. Benghazi, C. Rodríguez Domínguez. Un Diseño Basado en Componentes para el Desarrollo de Aplicaciones Web Adaptativas y Colaborativas. XI Congreso Internacional de Interacción Persona-Ordenador. ISBN: 978-84-92812-52-3, pp. 435-444. 2010.
11. J. Alcalá-Correa, M. J. Rodríguez-Fórtiz, C. Rodríguez Domínguez. Control del entorno para la diversidad funcional: Kora. VI Congreso Nacional de Tecnología Educativa y Atención a la Diversidad. ISBN: 978-84-693-1781-5, pp. 1-8. 2010.
12. C. Rodríguez Domínguez, K. Benghazi, M. Noguera, M. Bermúdez-Edo, J. L. Garrido. Dynamic Ontology-Based Redefinition of Events Intended to Support the Communication of Complex Information in Ubiquitous Computing. Journal of Network Protocols and Algorithms, Special Issue on Data Dissemination for Large Scale Complex Critical Infrastructures 2 (2). Macrothink Institute, ISSN 1943-3581. 2010. **Indexed in Index Copernicus, ProQuest, EBSCO, Ulrichsweb.com, Directory of Open Access Journals (DOAJ), Open J-Gate, Gale, Socolar, io-port Database, NewJour - Electronic Journals and Newsletters, Public Knowledge Project metadata harvester, Ovid LinkSolver, Genamics JournalSeek, PublicationsList.org.**
13. M. V. Hurtado, R. Ramos, K. Benghazi, M. Noguera, C. Rodríguez Domínguez. Groupbate: Soporte al Debate Virtual en Entornos de Aprendizaje Colaborativo. Avances en Ingeniería del Software Aplicada al E-Learning. Universidad Complutense de Madrid, ISBN: 978-84-693-9422-9, pp. 177-192. 2011.
14. C. Rodríguez Domínguez, K. Benghazi, M. Noguera, M. J. Rodríguez-Fórtiz, T. Ruiz-López. Framework de Soporte al Desarrollo Integrado de Sistemas Ubicuos. XIX Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2011). ISBN: 978-84-96737-99-0. 2011.
15. T. Ruiz López, J. L. Garrido, C. Rodríguez Domínguez, M. Noguera. Sherlock: A Hybrid, Adaptive Positioning Service

based on Standard Technologies. Evaluating AAL Systems through Competitive Benchmarking. 2011.

16. K. Benghazi, M. V. Hurtado, Miguel J. Hornos, M. L. Rodríguez, C. Rodríguez Domínguez, Ana B. Pelegrina, M. J. Rodríguez-Fórtiz. Enabling correct design and formal analysis of Ambient Assisted Living systems. *Journal of Systems and Software*. Elsevier. ISSN: 0164-1212. 2011. **Journal Citation Reports (JCR) (last published) 1.135, 5-year impact factor: 1.322. Indexed in CORE.**
17. M. V. Hurtado, R. Ramos, E. Trigueros, K. Benghazi, M. Noguera, C. Rodríguez Domínguez. Entorno de Interacción Colaborativa mediante Debate Virtual. *IEEE-RITA, Revista Iberoamericana de Tecnologías del Aprendizaje*, ISSN: 1932-8540, IEEE Education Society, Spain. 2011. **Indexed in DBLP and Ulrichsweb.com. Quality publisher.**
18. C. Rodríguez Domínguez, K. Benghazi, M. Noguera, J. L. Garrido, M. L. Rodríguez, T. Ruiz López. Seamless Integration of Communication Paradigms for Ubiquitous Systems. *Proceedings of Ubiquitous Computing and Ambient Intelligence (UCAmI '11)*. 2011.
19. R. Duque, M. L. Rodríguez, M. V. Hurtado, C. Bravo, C. Rodríguez Domínguez. Integration of collaboration and interaction analysis mechanisms in a concern-based architecture for groupware systems. *Journal of Science of Computer Programming, Special Issue: Solution-Oriented Architectures*, 77 (1). ISSN: 0167-6423. Elsevier Editorial. 2012. **Journal Citation Reports (JCR) 0.568, 5-year impact factor: 0.982.**
20. C. Rodríguez Domínguez, T. Ruiz López, K. Benghazi, J. L. Garrido. Designing a Middleware-Based Framework to Support Multiparadigm Communications in Ubiquitous Systems. *3rd International Symposium on Ambient Intelligence (ISAMI '12)*. 2012.
21. T. Ruiz López, C. Rodríguez Domínguez, M. Noguera, M. J. Rodríguez-Fórtiz. A Model-Driven Approach to Require-

- ments Engineering in Ubiquitous Systems. 3rd International Symposium on Ambient Intelligence (ISAMI '12). 2012.
22. C. Rodríguez Domínguez, T. Ruiz López, K. Benghazi, J. L. Garrido. A Communication Model to Integrate the Request-Response and the Publish-Subscribe Paradigms in Ubiquitous Systems. *Journal of Sensors, Special Issue on Select papers from UCAmI 2011*. 2012. **Journal Citation Reports (JCR) 1.953, 5-year impact factor: 2.395.**
 23. E. Villanueva, C. Rodríguez Domínguez, K. Benghazi, J. L. Garrido, A. Valenzuela. Applying Information Technology to Forensic Sciences. Springer, *International Journal of Legal Medicine* 126 (1) (Supplement), pp. S2. 2012. **Journal Citation Reports (JCR) 2.686.**
 24. T. Ruiz López, C. Rodríguez Domínguez, M. J. Rodríguez, J. L. Garrido. Mecanismos de Adaptación basados en Propiedades de Calidad: Un Caso de Estudio de un Servicio de Localización. XIII Congreso Internacional de Interacción Persona Ordenador. 2012.
 25. C. Rodríguez Domínguez, A. Caracuel, S. Santiago, M. J. Rodríguez, M. V. Hurtado. Plataforma Virtual de apoyo al Envejecimiento Activo. XIII Congreso Internacional de Interacción Persona Ordenador. 2012.
 26. C. Rodríguez Domínguez, K. Benghazi, J. L. Garrido, A. Valenzuela Garach. A Platform Supporting the Development of Applications in Ubiquitous Systems: The Collaborative Application Example of Mobile Forensics. XIII Congreso Internacional de Interacción Persona Ordenador. 2012.
 27. T. Ruiz López, C. Rodríguez Domínguez, M. Noguera, J. L. Garrido. Towards a Reusable Design of a Positioning System for AAL Environments. *Evaluating AAL Systems Through Competitive Benchmarking. Communications in Computer and Information Science* 309, pp. 65-79. 2012.
 28. M. A. Burgos, I. Garrido, Juan Martos, C. Rodríguez Domínguez, T. Ruiz López, M. Cabrera, M. L. Rodríguez,

- Á. Fernández, M. J. Rodríguez. Aplicación SÍGUEME. Estimulación para autismo de bajo nivel de funcionamiento. Toma de contacto para evaluar la captación de atención. VII Congreso Nacional de Tecnología Educativa y Atención a la Diversidad. 2013.
29. T. Ruiz López, C. Rodríguez Domínguez, M. Noguera, M. J. Rodríguez, K. BENGHAZI, J. L. Garrido. Applying Model-Driven Engineering to a method for systematic treatment of NFRs in AmI systems. IOS Press, *Journal of Ambient Intelligence and Smart Environments* 5 (3), pp. 287-310. 2013. **Journal Citation Reports (JCR) 1.298.**
30. C. Rodríguez Domínguez, T. Ruiz López, K. BENGHAZI, M. Noguera, J. L. Garrido. A Model-Driven Approach for the Development of Middleware Technologies for Ubiquitous Systems. 9th International Conference on Intelligent Environments (IE '13). 2013.
31. M. Cinque, D. Cotroneo, C. Rodríguez Domínguez, J. L. Garrido. Automatic Collection of Failure Data from the iOS Platform. Proc. of 2013 IEEE/IFIP 43rd International Conference on Dependable Systems and Networks Workshops (DSN-W). Workshop on Reliability and Security Data Analysis (RSDA 2013). ISBN: 978-1-4799-0181-4. IEEE Computer Society Press. 2013.
32. C. Rodríguez Domínguez, K. BENGHAZI, J. L. Garrido, A. Valenzuela. Designing a Communication Platform for Ubiquitous Systems: The Case Study of a Mobile Forensic Workspace. *New Trends in Interaction, Virtual Reality and Modeling*, pp. 97-111. ISBN 978-1-4471-5444-0. Springer, Human-Computer Interaction Series. 2013.
33. M. Vélez, Á. Burgos, I. Garrido, C. Rodríguez Domínguez, T. Ruiz López. Aplicación para el Desarrollo de Habilidades Perceptivo-Cognitivas y Conductuales en Niños y Niñas con Trastorno del Espectro Autista. VI Congreso de la de la Federación de Asociaciones de Neuropsicología Españolas (FANPSE). 2013.

34. G. Guerrero-Contreras, J. L. Garrido, C. Rodríguez Domínguez, M. Noguera, K. Benghazi. Designing a Service Platform for Sharing Internet Resources in MANETs. ESOCC Workshops, pp. 331-345. 2013.
35. T. Ruiz-López, C. Rodríguez Domínguez, M. Noguera, M. J. Rodríguez, J. L. Garrido. Towards a Component-based Design of Adaptive, Context-sensitive Services for Ubiquitous Systems. Intelligent Environments (Workshops), pp. 57-68. 2013.
36. T. Ruiz López, C. Rodríguez Domínguez, M. J. Rodríguez, S. F. Ochoa, J. L. Garrido. Context-aware Self-Adaptations: From Requirements Specification to Code Generation. Journal of Sensors, Special Issue on Select papers from UCAM I and IWAAL 2013. 2014. **Journal Citation Reports (JCR) 1.953, 5-year impact factor: 2.395.**
37. C. Rodríguez Domínguez, T. Ruiz López, K. Benghazi, J. L. Garrido. An MDA-based approach for the integration of communication technologies in ubiquitous systems. IOS Press, Journal of Ambient Intelligence and Smart Environments, thematic issue titled “Challenges of Engineering Intelligent Environments” (*selected, under review*). 2014. **Journal Citation Reports (JCR) 1.298.**
38. C. Rodríguez Domínguez, T. Ruiz López, J. L. Garrido, M. Noguera, K. Benghazi. A Model-Driven Approach to Service Composition on the basis of the Specification of BPMN Choreographies. International Journal of Computer Systems, Science and Engineering (*selected, under review*). 2014. **Journal Citation Reports (JCR) 0.08.**

Patents

Método y Sistema de Coordinación de Sistemas Software Basado en Arquitecturas Multiparadigma (Method and Coordination System for Software Systems based on Multiparadigm

Architectures). Main inventor: C. Rodríguez Domínguez. Patent owner: University of Granada.

Bibliography

- [1] ALEXANDER, C., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I., AND ANGEL, S. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [2] AMBLER, S. W. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004.
- [3] BAKRE, A., AND BADRINATH, B. Reworking the rpc paradigm for mobile clients. *Mobile Networks and Applications: Special issue on mobile computing and system services 1*, 4 (December 1996), 371–385.
- [4] BALASUBRAMANIAN, K., SCHMIDT, D. C., MOLNÁR, Z., AND LÉDECZI, Á. System integration using model-driven engineering. *Designing Software-Intensive Systems: Methods and Principles* (2008).
- [5] BALDONI, R., COTENTI, M., AND VIRGILLITO, A. The evolution of publish/subscribe communication paradigm. In *Future directions in distributed computing*, vol. 2584 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 137–141.
- [6] BARNLUND, D. C. *Communication Theory (2nd edition)*. Transaction Publishers, 2008, ch. A Transactional Model of Communication.
- [7] BATES, P. C. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.* 13, 1 (1995), 1–31.

- [8] BECKETT, D., AND MCBRIDE, B. RDF/XML Syntax Specification (Revised), W3C recommendation. Tech. rep., W3C, 2004.
- [9] BENJAMIN, P., PATKI, M., AND MAYER, R. Using ontologies for simulation modeling. In *Proceedings of the Winter Simulation Conference (WSC)* (Dec 2006), pp. 1151–1159.
- [10] BERJON, R., FAULKNER, S., LEITHEAD, T., NAVARA, E. D., O’CONNOR, E., PFEIFFER, S., AND HICKSON, I. Html5: A vocabulary and associated apis for html and xhtml. Candidate Recommendation 6, W3C, August, 2013.
- [11] BERLO, D. K. *The process of communication*. Ed. Holt, Rinehart and Winston, 1960.
- [12] BERNSTEIN, P. A. Middleware: a model for distributed system services. *Communications of the ACM* 39, 2 (February 1996), 86–98.
- [13] BERTOIA, M. F., AND VALLECILLO, A. Quality attributes for software metamodels. In *Proceedings of the 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010)* (Málaga, Spain, July 2010).
- [14] BIRRELL, A., AND NELSON, B. Implementing remote procedure calls. *Transactions on Computer Systems (TOCS)* 2, 1 (1984).
- [15] BISIGNANO, M., CALVAGNA, A., MODICA, G. D., AND TOMARCHIO, A. Expeerience: A jxta middleware for mobile ad hoc networks. *Proceedings of the third international conference on P2P computing* (2003).
- [16] BOOTH, D., HASS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C., AND ORCHARD, D. Web services architecture. *W3C Working Group Note 11* (Nov 2004), 1–98.

- [17] BORDIN, M., TSIODRAS, T., AND PERROTIN, M. Experience in the integration of heterogeneous models in the model-driven engineering of high-integrity systems. In *Reliable Software Technologies (Ada-Europe)*, vol. 5026 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 171–184.
- [18] BRAGHETTO, K. R., FERREIRA, J. E., AND VINCENT, J.-M. Rt-mac-2011-03: From business process model and notation to stochastic automata network. Tech. rep., University Of São Paulo, Institute Of Mathematics and Statistics, Department of Computer Science, March 2011.
- [19] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E., AND YERGAU, F. Extensible markup language (xml) 1.0. *W3C Recommendation* (2006).
- [20] BROWN, P. J. The Stick-e Document: a framework for creating context-aware applications. In *Proceedings of the Electronic Publishing '96* (1996), pp. 259–272.
- [21] CAMP, J., AND KNIGHTLY, E. The IEEE 802.11s Extended Service Set Mesh Networking Standard. *IEEE Communications Magazine* 46, 8 (2008), 120–126.
- [22] CARRIERO, N., AND GELERNTER, D. Linda in context. *Communications of the ACM* 32, 4 (1989), 444–458.
- [23] CECHICH, A., AND PIATTINI, M. On the measurement of cots functional suitability. In *COTS-Based Software Systems*, vol. 2959 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 31–40.
- [24] CHEN, H., FININ, T., AND JOSHI, A. An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review* 18, 3 (2004), 197–207.
- [25] CHEN, H., PERICH, F., FININ, T., AND JOSHI, A. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)* (2004).

- [26] CHINNICI, R., MOREAU, J. J., RYMAN, A., AND WEERAWARANA, S. Web services description language (wsdl) version 2.0 part 1: Core language. *W3C Recommendation 26* (June 2007), 1–103.
- [27] COMBEMALE, B., CRÉGUT, X., GIACOMETTI, J.-P., MICHEL, P., AND PANTEL, M. Introducing simulation and model animation in the mde topcased toolkit. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS 2008)* (2008).
- [28] CORSARO, A., QUERZONI, L., SCIPIONI, S., TUCCI, S., AND VIRGILLITO, A. Quality of Service in Publish/Subscribe Middleware. *Global Data Management* 8 (July 2006).
- [29] DENTLER, K., CORNET, R., TEN TEIJE, A., AND DE KEIZER, N. Comparison of reasoners for large ontologies in the owl 2 el profile. *Journal of Semantic Web* 2, 2 (April 2011), 71–87.
- [30] DEY, A. K. Context-aware computing: The cyberdesk project. In *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments* (1998), pp. 51–54.
- [31] DIMOKAS, N., KATSAROS, D., AND MANOLOPOULOS, Y. Node clustering in wireless sensor networks by considering structural characteristics of the network graph. In *Proceedings of the International Conference on Information Technology* (2007), pp. 122–127.
- [32] DUPUY-CHESSA, S. Quality in ubiquitous information system design. In *Proceedings of the 3rd Int. Conf. on Research Challenge in Information Science (RCIS)* (2009).
- [33] EUGSTER, P. T., FELBER, P. A., GUERRAOU, R., AND KERMARREC, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (2003), 114–131.
- [34] FARHOODI, F., AND GRAHAM, I. A practical approach to designing and building intelligent software agents. In *Proceedings of the 1st International Conference and Exhibition*

- of The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM '96)* (April 1996), pp. 181–204.
- [35] FIEGE, L., ZEIDLER, A., BUCHMANN, A., KEHR, R. K., AND MUHL, G. Security aspects in publish/subscribe systems. In *Proc. of the 3rd International Workshop on Distributed Event-based Systems (DEBS'04)* (Edinburgh, Scotland, UK, May 2004).
- [36] FLEUREY, F., MORIN, B., SOLBERG, A., AND BARAIS, O. Mde to manage communications with and between resource-constrained systems. In *Model Driven Engineering Languages and Systems*, vol. 6981 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 349–363.
- [37] FORWARD, A., AND LETHBRIDGE, T. C. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. In *Proceedings of the 2008 International Workshop on Models in Software Engineering* (2008), ACM, pp. 27–32.
- [38] FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.
- [39] FOWLER, M. *UML distilled: a brief guide to the standard object modeling language (3 ed.)*. Addison Wesley, 2004.
- [40] FREEMAN, E., ROBSON, E., BATES, B., AND SIERRA, K. *Head First Design Patterns*. O'Reilly Media, November 2004.
- [41] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [42] GELERNTER, D., AND CARRIERO, N. Coordination languages and their significance. *Communications of the ACM* 35, 2 (1992), 97–107.
- [43] GÖRGEN, D., FREY, H., LEHNERT, J. K., AND STURM, P. Selma: A middleware platform for self-organizing distributed

- applications in mobile multihop ad-hoc networks. *Western Simulation Multiconference* (2003).
- [44] GRACE, P., BLAIR, G. S., AND SAMUEL, S. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review* 9, 1 (January 2005), 2–14.
- [45] GRUBER, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition* 5, 2 (1993), 199–220.
- [46] GÜDEMANN, M., POIZAT, P., SALAÜN, G., AND DUMONT, A. Verchor: A framework for verifying choreographies. In *Fundamental Approaches to Software Engineering*, vol. 7793 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 226–230.
- [47] GUDGIN, M., HADLEY, M., MENDELSON, N., MOREAU, J. J., NIELSEN, H. F., KARMARKAR, A., AND LAFON, Y. Soap version 1.2 part 1: Messaging framework (second edition). *W3C Recommendation 27* (2007).
- [48] HADIM, S., AL-JAROODI, J., AND MOHAMED, N. Trends in middleware for mobile ad hoc networks. *Journal of Communications* 1, 4 (2006), 11–21.
- [49] HAILPERN, B., AND TARR, P. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal* 45, 3 (2006), 451–461.
- [50] HAYES, B. Cloud computing. *Communications of the ACM* 51, 7 (July 2008).
- [51] HE, Y., LIU, H., AND ZHU, F. Improved schemes for node clustering in decentralized peer-to-peer networks. In *Proceedings of the 3rd International IEEE Conference on Signal-Image Technologies and Internet-Based System* (2007), pp. 461–467.
- [52] HERRMANN, K., MÜHL, G., AND JAEGER, A. Meshmdl event spaces - a coordination middleware for self-organizing

- applications in ad hoc networks. *Pervasive and Mobile Computing* 3, 4 (2007), 467–487.
- [53] HERVÁS, R., BRAVO, J., AND FONTECHA, J. Awareness marks: Adaptive services through user interactions with augmented objects. *Personal and Ubiquitous Computing, Special Issue on Ubiquitous Computing and Ambient Intelligence* 5 (2011), 409–418.
- [54] INTERNATIONAL STANDARDS ORGANIZATION (ISO). Iso/iec 25010:2011 - systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. *ISO/IEC Standard* (2011).
- [55] JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. *The Unified Software Development Process*. Addison Wesley, 1999.
- [56] JETCHEVA, J., HU, Y., MALTZ, D., AND JOHNSON, D. A simple protocol for multicast and broadcast in mobile ad hoc networks. Tech. rep., IETF MANET Working Group: Internet Draft. Available online at: <http://www.monarch.cs.rice.edu/internet-drafts/draft-ietf-manet-simple-mbcast-01.txt>, 2007.
- [57] JONES, K. Building a context-aware service architecture. Developer works, IBM Corporation, December 2008.
- [58] KAINZ, G., BUCKL, C., AND KNOLL, A. Automated model-to-metamodel transformations based on the concepts of deep instantiation. In *Model Driven Engineering Languages and Systems*, vol. 6981 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 17–31.
- [59] KANG, K., KIM, S., LEE, J., KIM, K., KIM, G., AND SHIN, E. Form: A feature- oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering* 5, 1 (1998), 143–168.
- [60] KAPTEIJNS, T., JANSEN, S., BRINKKEMPER, S., HOUËT, H., AND BARENDSE, R. A comparative case study of model

- driven development vs traditional development: The tortoise or the hare. In *4th European Workshop: From code centric to model centric software engineering: Practices, Implications and ROI* (2009), pp. 22–33.
- [61] KARTHIKEYAN, N., PALANISAMY, V., AND DURAISWAMY, K. Performance comparison of broadcasting methods in mobile ad hoc network. *International Journal of Future Generation Communication and Networking* 2, 2 (June 2009), 47–58.
- [62] KOEHLER, J. The role of bpmn in a modeling methodology for dynamic process solutions. In *Business Process Modeling Notation*, vol. 67 of *Business Process Modeling Notation*. Springer Berlin Heidelberg, 2010, pp. 46–62.
- [63] KÖLLING, M., AND ROSENBERG, J. Blue - a language for teaching object-oriented programming. *SIGCSE Bulletin* 28, 1 (March 1996), 190–194.
- [64] LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (July 1978), 558–568.
- [65] LEVIS, P., AND CULLER, D. Mate: A tiny virtual machine for sensor networks. *Proceedings of the international conference on architectural support of programming languages and operative systems* (October 2002).
- [66] LEVY, H., AND TEMPERO, E. Modules, objects and distributed programming: Issues in RPC and remote object invocation. *Software Practice and Experience* 21, 1 (January 1991), 77–90.
- [67] LIM, S., LEE, W.-C., CAO, G., AND DAS, C. R. Performance comparison of cache invalidation strategies for internet-based mobile ad hoc networks. In *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems* (2004), pp. 104–113.

- [68] LIU, N., LIU, M., ZHU, J., AND GONG, H. A community-based event delivery protocol in publish/subscribe systems for delay tolerant sensor networks. *Journal of Sensors* 9, 10 (September 2009), 7580–7594.
- [69] MAIA, M., ROCHA, L., AND ANDRADE, R. Requirements and challenges for building service-oriented pervasive middleware. *Procs. of the 2009 intl. conf. on pervasive services* (2009).
- [70] MARTIN, A., AND LOOS, P. Software support for the computation independent modelling in the mda context. In *Proceedings of the 1st international workshop on business support for MDA* (2008).
- [71] MARTINEZ, J., MERINO, P., AND SALMERON, A. Applying mde methodologies to design communication protocols for distributed systems. In *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on* (2007), pp. 185–190.
- [72] MASCOLO, C., CAPRA, L., ZACHARIADIS, S., AND EMMERICH, W. Xmiddle: A data-sharing middleware for mobile computing. *Wireless Personal Computing* 21, 1 (2002), 77–103.
- [73] MEIER, R., AND CAHILL, V. Steam: Event-based middleware for wireless ad-hoc networks. *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops* (2002), 639–644.
- [74] MELLOR, S. J., CLARK, A. N., AND FUTAGAMI, T. Guest editor’s introduction: Model-driven development. *IEEE Software* 20, 5 (2003), 14–18.
- [75] MICHELSON, B. M. Event-driven architecture overview. *Patricia Seybold Group* (January 2006).
- [76] MILLER, F. P., VANDOME, A. F., AND MCBREWSTER, J. *Moore’s law: History of computing hardware, Integrated circuit, Accelerating change, Amdahl’s law, Metcalfe’s law,*

Mark Kryder, Jakob Nielsen (usability consultant), Wirth's law. Alpha Press, 2009.

- [77] MOHAGHEGHI, P., AND DEHLEN, V. Developing a quality framework for model-driven engineering. In *Models in Software Engineering*, vol. 5002 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 275–286.
- [78] MOHAGHEGHI, P., GILANI, W., STEFANESCU, A., FERNANDEZ, M., NORDMOEN, B., AND FRITZSCHE, M. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling* 12, 3 (2013), 619–639.
- [79] MULLER, G., MARLET, R., VOLANSCHI, E., CONSEL, C., PU, C., AND GOEL, A. Fast, optimized sun rpc using automatic program specialization. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS'98)* (May 1998), IEEE Computer Society Press, pp. 240–249.
- [80] MURPHY, A., PICCO, G., AND ROMAN, G. C. Lime: A coordination model and middleware supporting mobility of hosts and agents. *Transactions on Software Engineering and Methodology (TOSEM)* 15, 3 (Jul 2006).
- [81] MUSOLESI, M., MASCOLO, C., AND HAILES, S. Emma: Epidemic messaging middleware for ad-hoc networks. *Personal and Ubiquitous Computing* 10, 1 (2005), 28–36.
- [82] NAHRSTEDT, K. Distributed systems: Mobile and ubiquitous computing, lecture 25. Tech. rep., University of Illinois, 2009.
- [83] NWANA, H. S. Software agents: An overview. *Knowledge Engineering Review* 11, 3 (1996), 205–244.
- [84] OASIS SOA-RM TECHNICAL COMMITTEE. Service oriented architecture reference model (SOA-RM). Tech. rep., OASIS, 2006.

- [85] OBJECT MANAGEMENT GROUP (OMG). Model driven architecture. *OMG Specification*, <http://www.omg.org/mda> (2003).
- [86] OBJECT MANAGEMENT GROUP (OMG). Data Distribution Service for Real-time Systems Version 1.2. *OMG Specification* (January 2007), 1–260.
- [87] OBJECT MANAGEMENT GROUP (OMG). Common object request broker architecture (corba) specification version 3.1. part 1: Corba interfaces. *OMG Specification* (2008), 1–540.
- [88] OBJECT MANAGEMENT GROUP (OMG). Common object request broker architecture (corba) specification version 3.1. part 2: Corba interoperability. *OMG Specification* (January 2008), 1–260.
- [89] OBJECT MANAGEMENT GROUP (OMG). Business process model and notation 2.0. *OMG Specification*, <http://www.omg.org/spec/BPMN/2.0/> (January 2011), 1–538.
- [90] OBJECT MANAGEMENT GROUP (OMG). Meta object facility 2.0 query/view/transformation (qvt). *OMG Specification*, <http://www.omg.org/spec/QVT> (2011).
- [91] OBJECT MANAGEMENT GROUP (OMG). Service oriented architecture modeling language (soaml) version 1.0.1. *OMG Adopted Specification* (May 2012), 1–132.
- [92] OBJECT MANAGEMENT GROUP (OMG). Meta object facility (mof) core 2.4.1. *OMG Specification*, <http://www.omg.org/spec/MOF> (June 2013).
- [93] PAIGE, R. F., OSTROFF, J. S., AND BROOKE, P. J. Principles for modeling language design. *Journal of Information and Software Technology* 42, 10 (July 2000), 665–675.
- [94] PASCOE, J. The stick-e note architecture: Extending the interface beyond the user. *Proceedings of the international conference on intelligent user interfaces* (1997), 261–264.

- [95] PREE, W. Meta patterns - a means for capturing the essentials of reusable object-oriented design. *Proc. of the 8th European Conf. on Object-Oriented Programming* (1994), 150–162.
- [96] PREE, W. Hot-spot-driven framework development. *Building Application Frameworks: Object-Oriented Foundations* (2000).
- [97] RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. Corona: A high performance publish-subscribe system for the world wide web. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation* (San Jose, California, USA, May 2007), pp. 15–28.
- [98] RIEHLE, D. *Framework Design*. Swiss Federal Institute of Technology, 2000.
- [99] RODRÍGUEZ-DOMÍNGUEZ, C., A., F., ALCALÁ-CORREA, J., RODRÍGUEZ-FÓRTIZ, M., AND GARRIDO, J. A design proposal to support the integration and interoperability of applications for people with special needs (originally available in spanish). In *Proceedings of the 11th International Conference on Human-Computer Interaction* (2011), ACM, pp. 401–410.
- [100] RODRÍGUEZ-DOMÍNGUEZ, C., BENGHAZI, K., GARRIDO, J. L., AND VALENZUELA, A. A platform supporting the development of applications in ubiquitous systems: the collaborative application example of mobile forensics. In *Proceedings of the 13th International Conference on Human-Computer Interaction* (2012), ACM, pp. 41:1–41:7.
- [101] RODRÍGUEZ-DOMÍNGUEZ, C., BENGHAZI, K., NOGUERA, M., GARRIDO, J. L., RODRÍGUEZ, M. L., AND RUIZ-LÓPEZ, T. A communication model to integrate the request-response and the publish-subscribe paradigms in ubiquitous systems. *Journal of Sensors* 12, 6 (2012), 7648–7668.
- [102] RODRÍGUEZ-DOMÍNGUEZ, C., CARACUEL, A., SANTIAGO-RAMAJÓ, S., RODRÍGUEZ-FÓRTIZ, M. J., HURTADO, M. V., AND FERNÁNDEZ-LÓPEZ, Á.

- Plataforma virtual de apoyo al envejecimiento activo. In *Proceedings of the 13th International Conference on Human-Computer Interaction* (2012), ACM, pp. 151–158.
- [103] ROSS, A. M., RHODES, D. H., AND HASTINGS, D. E. Defining changeability: Reconciling flexibility, adaptability, scalability, modifiability, and robustness for maintaining system lifecycle value. *Journal of Systems Engineering* 11, 3 (August 2008), 246–262.
- [104] RUIZ-LÓPEZ, T., GARRIDO, J. L., RODRÍGUEZ-DOMÍNGUEZ, C., AND NOGUERA, M. Sherlock: A hybrid and adaptive positioning system based on standard technologies. In *Evaluating AAL Systems through Competitive Benchmarking* (2011).
- [105] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.
- [106] SAIF, U., AND GREAVES, D. Communication primitives for ubiquitous systems or rpc considered harmful. In *Proceedings of the 21st International Conference on Distributed Computing Systems Workshops* (2001), pp. 240–245.
- [107] SCHLIT, B., ADAMS, N., AND WANT, R. Context-aware computing applications. *Proc. of Workshop on Mobile Computing Systems and Applications* (1994), 85–90.
- [108] SCHILLING, M. A. Towards a general modular systems theory and its application to inter-firm product modularity. *Academy of Management Review* 25 (September 1999), 312–334.
- [109] SCHMIDT, A. *Ubiquitous Computing - Computing in Context*. PhD thesis, Computing Department, Lancaster University, UK, November 2002.
- [110] SCHMIDT, D., AND BUSCHMANN, F. Patterns, frameworks, and middleware: their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering* (2003), pp. 694–704.

- [111] SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. *IEEE Computer Magazine* 39, 2 (2006), 25–31.
- [112] SELIC, B. From model-driven development to model-driven engineering. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS) (2007)*, p. 3.
- [113] SELTVEIT, A. *Complexity Reduction in Information Systems Modelling*. PhD thesis, University of Trondheim, Norway, 1994.
- [114] SHANKAR, C., AL-MUHTADI, J., CAMPBELL, R., AND MICKUNAS, M. D. Mobile gaia: A middleware for ad hoc pervasive computing. *IEEE Consumer Communications and Networking Conference (CCNC 2005)* (January 2005).
- [115] SHANNON, C. E., AND WEAVER, W. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.
- [116] STÖRRLE, H., AND HAUSMANN, J. H. Towards a formal semantics of uml 2.0 activities. In *German Software Engineering Conference (2005)*, pp. 117–128.
- [117] TANENBAUM, A., AND VAN RENESSE, R. A critique of the remote procedure call paradigm. In *Proceedings of the EUTECO 88 Conference* (April 1988), R. Speth, Ed., Elsevier Science Publishers B. V., pp. 775–783.
- [118] TAPIA, D. I., ALONSO, R. S., DE LA PRIETA, F., ZATO, C., RODRÍGUEZ, S., CORCHADO, E., BAJO, J., AND CORCHADO, J. M. SYLPH: An ambient intelligence based platform for integrating heterogeneous wireless sensor networks. *IEEE International Conference on Fuzzy Systems* (2010).
- [119] TERRIER, F., AND GÉRARD, S. Mde benefits for distributed, real time and embedded systems. *IFIP International Federation for Information Processing, From Model-Driven Design to Resource Management for Distributed Embedded Systems* 225 (2006), 15–24.

- [120] TOUIL, A., VAREILLE, J., LHERMINIER, F., AND LE PARC, P. Modeling and analysing ubiquitous systems using mde approach. *Proceedings of the 4th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM)* (2010), 1–6.
- [121] TRUJILLO, S., BATORY, D., AND DIAZ, O. Feature oriented model driven development: A case study for portlets. In *29th International Conference on Software Engineering (ICSE)* (2007), pp. 44–53.
- [122] TSENG, Y. C., NI, S. Y., CHEN, Y. S., AND SHEU, J. P. The Broadcast Storm Problem in a Mobile Ad Hoc Network. *Wireless Networks* 8 (March 2002), 153–167.
- [123] VAN DER AALST, W., AND STAHL, C. *Modeling Business Processes: A Petri Net-Oriented Approach*. MIT Press, 2011.
- [124] W3C. OWL 2 web ontology language, W3C recommendation. Tech. rep., W3C OWL Working Group, 2009.
- [125] WADDINGTON, D. G., AND LARDIERI, P. Model centric software development. *IEEE Computer Magazine* 39, 2 (February 2006).
- [126] WARD, A., JONES, A., AND HOPPER, A. A new location technique for the active office. *IEEE Personal Communications* 4, 5 (1997), 42–47.
- [127] WEISER, M. The computer for the 21st century. *Scientific American* 265, 3 (September 1991), 94–104.
- [128] WEISS, A. Computing in the clouds. *netWorker* 11, 4 (December 2007).
- [129] YOSHITAKA, S., TAKADA, K., ANISETTI, M., BELLANDI, V., CERAVOLO, P., DAMIANI, E., AND TSURUTA, S. Toward sensor-based context aware systems. *Journal of Sensors* 12 (2012), 632–649.
- [130] ZERO C INC. Distributed programming with ICE. *Available online at: <http://doc.zero.c.com/display/Ice/Ice+Manual>* (2013).

BIBLIOGRAPHY

Appendices

I. Implementation of the CI-CS ontology in OWL

```
<?xml version="1.0"?>
```

```
<!DOCTYPE Ontology [  
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >  
  <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >  
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >  
  >  
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-  
-ns#" >  
>
```

```
<Ontology xmlns="http://www.w3.org/2002/07/owl#"  
  xml:base="http://www.semanticweb.org/  
  carlosrodriguezdominguez/ontologies/2014/1/  
  communication_system"  
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"  
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-  
ns#"  
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
```

```
ontologyIRI="http://www.semanticweb.org/
  carlosrodriguezdominguez/ontologies/2014/1/
  communication_system">
<Prefix name="" IRI="http://www.w3.org/2002/07/owl#"
  />
<Prefix name="owl" IRI="http://www.w3.org/2002/07/owl
  #"/>
<Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-
  rdf-syntax-ns#"/>
<Prefix name="xsd" IRI="http://www.w3.org/2001/
  XMLSchema#"/>
<Prefix name="rdfs" IRI="http://www.w3.org/2000/01/
  rdf-schema#"/>
<Annotation>
  <AnnotationProperty abbreviatedIRI="rdfs:comment"
  />
  <Literal datatypeIRI="&xsd:string">An ontology to
  represent Computation Independent
  Communication Systems.</Literal>
</Annotation>
<Annotation>
  <AnnotationProperty abbreviatedIRI="owl:
  versionInfo"/>
  <Literal datatypeIRI="&xsd:string">1.0</Literal>
</Annotation>
<Declaration>
  <Class IRI="#Active"/>
</Declaration>
<Declaration>
  <Class IRI="#Channel"/>
</Declaration>
<Declaration>
  <Class IRI="#Choreography"/>
</Declaration>
<Declaration>
  <Class IRI="#ChoreographyActivity"/>
</Declaration>
<Declaration>
  <Class IRI="#CommunicationSystem"/>
</Declaration>
<Declaration>
```

```
<Class IRI="#Complex"/>
</Declaration>
<Declaration>
  <Class IRI="#Conditional"/>
</Declaration>
<Declaration>
  <Class IRI="#Content-Based"/>
</Declaration>
<Declaration>
  <Class IRI="#Default"/>
</Declaration>
<Declaration>
  <Class IRI="#EndEvent"/>
</Declaration>
<Declaration>
  <Class IRI="#Event"/>
</Declaration>
<Declaration>
  <Class IRI="#Event-Based"/>
</Declaration>
<Declaration>
  <Class IRI="#Exclusive"/>
</Declaration>
<Declaration>
  <Class IRI="#FlowObject"/>
</Declaration>
<Declaration>
  <Class IRI="#Gateway"/>
</Declaration>
<Declaration>
  <Class IRI="#Inclusive"/>
</Declaration>
<Declaration>
  <Class IRI="#InitialEvent"/>
</Declaration>
<Declaration>
  <Class IRI="#Link"/>
</Declaration>
<Declaration>
  <Class IRI="#Message"/>
</Declaration>
```

```
<Declaration>
  <Class IRI="#Parallel"/>
</Declaration>
<Declaration>
  <Class IRI="#Participant"/>
</Declaration>
<Declaration>
  <Class IRI="#ParticipantRole"/>
</Declaration>
<Declaration>
  <Class IRI="#Passive"/>
</Declaration>
<Declaration>
  <Class IRI="#Peer"/>
</Declaration>
<Declaration>
  <Class IRI="#Protocol"/>
</Declaration>
<Declaration>
  <Class IRI="#Sequential"/>
</Declaration>
<Declaration>
  <Class IRI="#SubChoreographyActivity"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasActivityParticipants"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasChannels"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasChoreographyMessages"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasChoreographyParticipants"
  />
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasLink"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#hasLinks"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasParticipants"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasProtocols"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasRole"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasSubactivities"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isActivityParticipant"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChannelOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyMessageOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyParticipant"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isCommunicativeCommonalityOf
    "/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isComplierWith"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isComposedBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isConnectedTo"/>
```

```
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isConnectorOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isFinalizedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isFinalizerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isInitializedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isInitializerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isLinkOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isMediumOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isParticipantOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isProtocolOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isRoleOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isStarterOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isStructurerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isSubactivityOf"/>
</Declaration>
<Declaration>
```

```

    <ObjectProperty IRI="#isTransferUnitOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isTransportedBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isTransporterOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isTriggeredBy"/>
</Declaration>
<EquivalentClasses>
    <Class IRI="#ParticipantRole"/>
    <ObjectUnionOf>
        <Class IRI="#Active"/>
        <Class IRI="#Passive"/>
        <Class IRI="#Peer"/>
    </ObjectUnionOf>
</EquivalentClasses>
<SubClassOf>
    <Class IRI="#Active"/>
    <Class IRI="#ParticipantRole"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Channel"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#isChannelOf"/>
        <Class IRI="#CommunicationSystem"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Channel"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#isMediumOf"/>
        <Class IRI="#Protocol"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Channel"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#isTransporterOf"/>

```



```
        <Class IRI="#Message"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#"
            hasChoreographyParticipants"/>
        <Class IRI="#Participant"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasLinks"/>
        <Class IRI="#Link"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#isFinalizedBy"/>
        <Class IRI="#EndEvent"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectMinCardinality cardinality="2">
        <ObjectProperty IRI="#hasChoreographyMessages"
            />
        <Class IRI="#Message"/>
    </ObjectMinCardinality>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectExactCardinality cardinality="1">
        <ObjectProperty IRI="#isInitializedBy"/>
        <Class IRI="#InitialEvent"/>
    </ObjectExactCardinality>
</SubClassOf>
<SubClassOf>
```

```

    <Class IRI="#ChoreographyActivity"/>
    <Class IRI="#FlowObject"/>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#ChoreographyActivity"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasActivityParticipants"
        />
      <Class IRI="#Participant"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#ChoreographyActivity"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasChoreographyMessages"
        />
      <Class IRI="#Message"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationSystem"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasChannels"/>
      <Class IRI="#Channel"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationSystem"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasParticipants"/>
      <Class IRI="#Participant"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationSystem"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasProtocols"/>
      <Class IRI="#Protocol"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>

```

```
    <Class IRI="#Complex"/>
    <Class IRI="#Gateway"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Conditional"/>
    <Class IRI="#Link"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Content-Based"/>
    <Class IRI="#Exclusive"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Default"/>
    <Class IRI="#Link"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#EndEvent"/>
    <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#EndEvent"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#isFinalizerOf"/>
        <Class IRI="#Choreography"/>
    </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Event"/>
    <Class IRI="#FlowObject"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Event"/>
    <Class IRI="#Message"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Event-Based"/>
    <Class IRI="#Exclusive"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Exclusive"/>
    <Class IRI="#Gateway"/>
```

```

</SubClassOf>
<SubClassOf>
  <Class IRI="#Exclusive"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isTriggeredBy"/>
    <Class IRI="#Event"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#FlowObject"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasLink"/>
    <Class IRI="#Link"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Gateway"/>
  <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Gateway"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isStarterOf"/>
    <Class IRI="#FlowObject"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Inclusive"/>
  <Class IRI="#Gateway"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#InitialEvent"/>
  <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#InitialEvent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isInitializerOf"/>
    <Class IRI="#Choreography"/>
  </ObjectSomeValuesFrom>
</SubClassOf>

```

```
<SubClassOf>
  <Class IRI="#Link"/>
  <ObjectMinCardinality cardinality="1">
    <ObjectProperty IRI="#isLinkOf"/>
    <Class IRI="#Choreography"/>
  </ObjectMinCardinality>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Link"/>
  <ObjectExactCardinality cardinality="2">
    <ObjectProperty IRI="#isConnectorOf"/>
    <Class IRI="#FlowObject"/>
  </ObjectExactCardinality>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Message"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isComplierWith"/>
    <Class IRI="#Protocol"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Message"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isTransferUnitOf"/>
    <Class IRI="#Participant"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Parallel"/>
  <Class IRI="#Gateway"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Participant"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isActivityParticipant"/>
    <Class IRI="#ChoreographyActivity"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Participant"/>
```

```

    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isConnectedTo"/>
      <Class IRI="#Participant"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Participant"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isParticipantOf"/>
      <Class IRI="#CommunicationSystem"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Participant"/>
    <ObjectExactCardinality cardinality="1">
      <ObjectProperty IRI="#hasRole"/>
      <Class IRI="#ParticipantRole"/>
    </ObjectExactCardinality>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Passive"/>
    <Class IRI="#ParticipantRole"/>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Peer"/>
    <Class IRI="#ParticipantRole"/>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Protocol"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#
        isCommunicativeCommonalityOf"/>
      <Class IRI="#Channel"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Protocol"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isProtocolOf"/>
      <Class IRI="#CommunicationSystem"/>
    </ObjectSomeValuesFrom>

```

```
</SubClassOf>
<SubClassOf>
  <Class IRI="#Sequential"/>
  <Class IRI="#Link"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SubChoreographyActivity"/>
  <Class IRI="#ChoreographyActivity"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SubChoreographyActivity"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isSubactivityOf"/>
    <Class IRI="#ChoreographyActivity"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#Choreography"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#CommunicationSystem"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#FlowObject"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#Link"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#Message"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
```

```

    <Class IRI="#Channel"/>
    <Class IRI="#ParticipantRole"/>
  </DisjointClasses>
<DisjointClasses>
  <Class IRI="#Channel"/>
  <Class IRI="#Protocol"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#CommunicationSystem"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#Link"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#Message"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#ParticipantRole"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#Choreography"/>
  <Class IRI="#Protocol"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#ChoreographyActivity"/>
  <Class IRI="#Event"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#CommunicationSystem"/>
  <Class IRI="#FlowObject"/>
</DisjointClasses>
<DisjointClasses>
  <Class IRI="#CommunicationSystem"/>

```



```
    <Class IRI="#Link"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CommunicationSystem"/>
    <Class IRI="#Message"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CommunicationSystem"/>
    <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CommunicationSystem"/>
    <Class IRI="#ParticipantRole"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CommunicationSystem"/>
    <Class IRI="#Protocol"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Complex"/>
    <Class IRI="#Exclusive"/>
    <Class IRI="#Inclusive"/>
    <Class IRI="#Parallel"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Conditional"/>
    <Class IRI="#Default"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Conditional"/>
    <Class IRI="#Sequential"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Content-Based"/>
    <Class IRI="#Event-Based"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Default"/>
    <Class IRI="#Sequential"/>
</DisjointClasses>
<DisjointClasses>
```

```
    <Class IRI="#EndEvent"/>
    <Class IRI="#Gateway"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#EndEvent"/>
    <Class IRI="#InitialEvent"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#FlowObject"/>
    <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#FlowObject"/>
    <Class IRI="#ParticipantRole"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#FlowObject"/>
    <Class IRI="#Protocol"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Gateway"/>
    <Class IRI="#InitialEvent"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Link"/>
    <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Link"/>
    <Class IRI="#ParticipantRole"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Link"/>
    <Class IRI="#Protocol"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Message"/>
    <Class IRI="#Participant"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Message"/>
```

```
    <Class IRI="#ParticipantRole"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#Message"/>
    <Class IRI="#Protocol"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#Participant"/>
    <Class IRI="#ParticipantRole"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#Participant"/>
    <Class IRI="#Protocol"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#ParticipantRole"/>
    <Class IRI="#Protocol"/>
  </DisjointClasses>
  <InverseObjectProperties>
    <ObjectProperty IRI="#hasActivityParticipants"/>
    <ObjectProperty IRI="#isActivityParticipant"/>
  </InverseObjectProperties>
  <InverseObjectProperties>
    <ObjectProperty IRI="#hasChannels"/>
    <ObjectProperty IRI="#isChannelOf"/>
  </InverseObjectProperties>
  <InverseObjectProperties>
    <ObjectProperty IRI="#isChoreographyMessageOf"/>
    <ObjectProperty IRI="#hasChoreographyMessages"/>
  </InverseObjectProperties>
  <InverseObjectProperties>
    <ObjectProperty IRI="#hasChoreographyParticipants"
      />
    <ObjectProperty IRI="#isChoreographyParticipant"/>
  </InverseObjectProperties>
  <InverseObjectProperties>
    <ObjectProperty IRI="#isConnectorOf"/>
    <ObjectProperty IRI="#hasLink"/>
  </InverseObjectProperties>
  <InverseObjectProperties>
    <ObjectProperty IRI="#isLinkOf"/>
```

```
    <ObjectProperty IRI="#hasLinks"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasParticipants"/>
    <ObjectProperty IRI="#isParticipantOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isProtocolOf"/>
    <ObjectProperty IRI="#hasProtocols"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasRole"/>
    <ObjectProperty IRI="#isRoleOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isSubactivityOf"/>
    <ObjectProperty IRI="#hasSubactivities"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isCommunicativeCommonalityOf
"/>
    <ObjectProperty IRI="#isMediumOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isComplierWith"/>
    <ObjectProperty IRI="#isStructurerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isFinalizerOf"/>
    <ObjectProperty IRI="#isFinalizedBy"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isInitializedBy"/>
    <ObjectProperty IRI="#isInitializerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isTransporterOf"/>
    <ObjectProperty IRI="#isTransportedBy"/>
</InverseObjectProperties>
<FunctionalObjectProperty>
    <ObjectProperty IRI="#hasRole"/>
```

```
</FunctionalObjectProperty>
<FunctionalObjectProperty>
  <ObjectProperty IRI="#isInitializedBy"/>
</FunctionalObjectProperty>
<FunctionalObjectProperty>
  <ObjectProperty IRI="#isTriggeredBy"/>
</FunctionalObjectProperty>
<InverseFunctionalObjectProperty>
  <ObjectProperty IRI="#isRoleOf"/>
</InverseFunctionalObjectProperty>
<SymmetricObjectProperty>
  <ObjectProperty IRI="#isConnectedTo"/>
</SymmetricObjectProperty>
<TransitiveObjectProperty>
  <ObjectProperty IRI="#isComposedBy"/>
</TransitiveObjectProperty>
<TransitiveObjectProperty>
  <ObjectProperty IRI="#isConnectedTo"/>
</TransitiveObjectProperty>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasActivityParticipants"/>
  <Class IRI="#ChoreographyActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChannels"/>
  <Class IRI="#CommunicationSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChoreographyMessages"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChoreographyParticipants"
  />
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasLink"/>
  <Class IRI="#FlowObject"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
```

```

    <ObjectProperty IRI="#hasLinks"/>
    <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasParticipants"/>
    <Class IRI="#CommunicationSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasProtocols"/>
    <Class IRI="#CommunicationSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasRole"/>
    <Class IRI="#Participant"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#hasSubactivities"/>
    <Class IRI="#ChoreographyActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isActivityParticipant"/>
    <Class IRI="#Participant"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isChannelOf"/>
    <Class IRI="#Channel"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isChoreographyMessageOf"/>
    <Class IRI="#Message"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isChoreographyOf"/>
    <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isChoreographyParticipant"/>
    <Class IRI="#Participant"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>

```

```
    <ObjectProperty IRI="#isCommunicativeCommonalityOf
      "/>
    <Class IRI="#Protocol"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isComplierWith"/>
    <Class IRI="#Message"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isComposedBy"/>
    <Class IRI="#Protocol"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isConnectedTo"/>
    <Class IRI="#Participant"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isConnectorOf"/>
    <Class IRI="#Link"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isFinalizedBy"/>
    <Class IRI="#Choreography"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isFinalizerOf"/>
    <Class IRI="#EndEvent"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isInitializedBy"/>
    <Class IRI="#Choreography"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isInitializerOf"/>
    <Class IRI="#InitialEvent"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isLinkOf"/>
    <Class IRI="#Link"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
```

```
    <ObjectProperty IRI="#isMediumOf"/>
    <Class IRI="#Channel"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isParticipantOf"/>
    <Class IRI="#Participant"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isProtocolOf"/>
    <Class IRI="#Protocol"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isRoleOf"/>
    <Class IRI="#ParticipantRole"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isStarterOf"/>
    <Class IRI="#Gateway"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isStructurerOf"/>
    <Class IRI="#Protocol"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isSubactivityOf"/>
    <Class IRI="#SubChoreographyActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isTransferUnitOf"/>
    <Class IRI="#Message"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isTransportedBy"/>
    <Class IRI="#Message"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isTransporterOf"/>
    <Class IRI="#Channel"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isTriggeredBy"/>
```



```
    <Class IRI="#Exclusive"/>
  </ObjectPropertyDomain>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasActivityParticipants"/>
    <Class IRI="#Participant"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasChannels"/>
    <Class IRI="#Channel"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasChoreographyMessages"/>
    <Class IRI="#Message"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasChoreographyParticipants"
      />
    <Class IRI="#Participant"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasLink"/>
    <Class IRI="#Link"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasLinks"/>
    <Class IRI="#Link"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasParticipants"/>
    <Class IRI="#Participant"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasProtocols"/>
    <Class IRI="#Protocol"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasRole"/>
    <Class IRI="#ParticipantRole"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#hasSubactivities"/>
```

```
        <Class IRI="#SubChoreographyActivity"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isActivityParticipant"/>
        <Class IRI="#ChoreographyActivity"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isChannelOf"/>
        <Class IRI="#CommunicationSystem"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isChoreographyMessageOf"/>
        <Class IRI="#Choreography"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isChoreographyOf"/>
        <Class IRI="#CommunicationSystem"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isChoreographyParticipant"/>
        <Class IRI="#Choreography"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isCommunicativeCommonalityOf
        "/>
        <Class IRI="#Channel"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isComplierWith"/>
        <Class IRI="#Protocol"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isComposedBy"/>
        <Class IRI="#Protocol"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isConnectedTo"/>
        <Class IRI="#Participant"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#isConnectorOf"/>
```

```
    <Class IRI="#FlowObject"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isFinalizedBy"/>
    <Class IRI="#EndEvent"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isFinalizerOf"/>
    <Class IRI="#Choreography"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isInitializedBy"/>
    <Class IRI="#InitialEvent"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isInitializerOf"/>
    <Class IRI="#Choreography"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isLinkOf"/>
    <Class IRI="#Choreography"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isMediumOf"/>
    <Class IRI="#Protocol"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isParticipantOf"/>
    <Class IRI="#CommunicationSystem"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isProtocolOf"/>
    <Class IRI="#CommunicationSystem"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isRoleOf"/>
    <Class IRI="#Participant"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isStarterOf"/>
    <Class IRI="#FlowObject"/>
```

```
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isStructurerOf"/>
  <Class IRI="#Message"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isSubactivityOf"/>
  <Class IRI="#ChoreographyActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransferUnitOf"/>
  <Class IRI="#Participant"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransportedBy"/>
  <Class IRI="#Channel"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransporterOf"/>
  <Class IRI="#Message"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTriggeredBy"/>
  <Class IRI="#Event"/>
</ObjectPropertyRange>
</Ontology>
```

II. Quality Attributes of the CI-CS Metamodel

II.1 Functional Suitability

The CI-CS metamodel is able to represent different communication scenarios with multiple specific requirements, as has been described along Section 3.2. For instance, it is possible to describe communication systems composed only by passive participants, only by active participants or by peers.

Moreover, the metamodel can capture the requirements that a communication system with a certain structure should satisfy. For example, given the set of participants, messages, channels and protocols (i.e., the concepts present in the structural view), it is possible to identify the needed interactions between those elements (i.e., the concepts present in the behavioral view), and to explicitly define a choreography to organize them.

Conversely, given an organization of a communication system, it is possible to obtain the structural elements that are required to be present in the communication system itself. As an example, if a choreography specifies that two participants interact, then both participants should, at least, share a channel and use some common protocol.

Finally, the functional suitability of a metamodel can be also defined on the basis of a set of sub-characteristics described in the ISO/IEC 25010:2011 standard. The following subsections explain how the metamodel is able to address the analysis of those sub-characteristics. The resulting conclusion is that the functional suitability of the metamodel can be considered to be high. Nonetheless, note that the standard defines a security sub-characteristic, which has not been taken into account, since it is not suitable to describe the quality of a metamodel [13].

II.1.1. Functional Appropriateness

The functional appropriateness is the degree to which the metamodel provides an appropriate set of functions for specified tasks and user objectives. It can be studied by analyzing multiple aspects, also defined in the ISO/IEC 25010:2011 standard.

For example, the metamodel is **relevant**, that is, it only contains the concepts and relationships that are necessary for a particular transformation (i.e., from a very general communication system to a concrete ubiquitous system). The transformation of the CI-CS into more concrete models is explored in Chapter 5, which contributes to the clarification of this assessment. Also, the metamodel specification takes into account that all the concepts and relationships *must be* part of any communication system. Therefore, there should be not a communication system without some of the concepts present in the metamodel. Consequently, the metamodel is also **concise**, in the sense that it is not unnecessarily extensive and it *just* contains the needed concepts and relationships.

Finally, the metamodel is **cohesive**, since it is only focused on one topic, that is, in the description of a communication system, and **confined**, since it only contains concepts at one abstraction level.

II.1.2. Accuracy

The accuracy is the degree to which the metamodel provides the right or specified results with the needed degree of precision. One related aspect is the **validity**, that is, the metamodel contains information that has been contrasted against a reliable source of information. In this case, the metamodel has been devised from the most well-known and accepted communication models (i.e., SMCR, Shannon-Weaver, Barnlund and the BPMN 2.0 choreography metamodel). Finally, another notion related to the accuracy is the **precision**, which is directly accomplished if the metamodel is **relevant** [13], as it was described in previous subsection.

II.1.3. Interoperability

The interoperability is the degree to which the metamodel can be cooperatively operable with one or more other metamodels. Since the CI-CS metamodel has been formally specified as an ontology, it can interoperate with other ontologies specifying other metamodels in order to complete its information in some aspects, to include new notions related to other domains, etc.

II.1.4. Compliance

The functional suitability compliance is the degree to which the metamodel adheres to standards, conventions or regulations in laws and similar prescriptions relating to functional suitability. Consequently, since the metamodel is *valid*, as specified in Subsubsection II.1.2, then it adheres to certain well-established conventions. Additionally, as the metamodel has been formally described as an ontology, then it adheres to an standard way of formally specifying knowledge.

II.2 Reliability

The reliability is the degree to which the metamodel can maintain a specified level of performance when used under specified conditions. In other words, the reliability of the metamodel refers to the degree of expressiveness that can be assessed, even if a fault in the metamodel specification is detected. Therefore, the proposed metamodel can be considered partially reliable in the sense that, even if a fault in the metamodel specification is detected, a minimum degree of expressiveness can be assessed, since the metamodel includes the concepts and relationships that are present in the most widely accepted communication theories. This way, the reliability of the metamodel is linked to its functional suitability compliance (Subsubsection II.1.4) and accuracy (Subsubsection II.1.2).

II.3 Performance Efficiency

The performance efficiency of the metamodel is related to the amount of classes and relationships that it uses. As previously stated in subsection II.1.1, the metamodel is intended to be simple enough to represent any communication system and to just include the needed elements to do so. In this way, it is possible to relate its conciseness to its efficiency. Moreover, using the metamodel to produce models can be considered as an efficient task, since the produced models should only have the strictly needed elements to represent a concrete communication system.

II.4 Operability

The operability of the metamodel is closely related to its simplicity and conciseness. In that sense, it is possible to attribute the metamodel with a recognizable minimum operability level. Anyhow, the operability is a completely subjective attribute that is also associated to many different characteristics, as will be explored in the following subsections.

II.4.1. Appropriateness Recognizability

The appropriateness recognizability refers to the degree to which the metamodel enables users to recognize whether it is appropriate for their needs or not. To this respect, the metamodel is clearly focused on describing communication systems as a whole and in a computation-independent manner (i.e., without including any technical artifacts). Thereby, it should be recognizable by the users as appropriate if they need to describe one of these systems. Additionally, the specification of the metamodel as an ontology should help to clear up the focus of the metamodel and serves, at least, to check if a description of a system is compliant with the proposed conceptualization of a communication system.

II.4.2. Learnability

The learnability is the capability of the metamodel to enable the user to learn its application (use, meaning, representation). This attribute is considered to be linked to the presence of *clean concepts*, that is, concepts that directly reflect the theoretical model behind the metamodel and not related to any technical or secondary issues [63]. This way, the metamodel can be considered to be learnable by any person that is familiar with the communication theory, since it explicitly reflects the notions present in those theories without any additional artifacts.

II.4.3. Helpfulness

The helpfulness of the metamodel is referred to the degree to which it provides help when users need assistance. The CI-CS metamodel may help into analyzing, designing or validating a communication system. Consequently, it may considered to be helpful for those tasks.

II.4.4. Attractiveness

The attractiveness is the capability of the metamodel to be attractive to the user. Even if this is a completely subjective attribute that is more related to the attraction of the user of the metamodel to the field of the communications, it can also be related to its learnability. Consequently, since the metamodel has that attribute, it could be determined that it exhibits a certain level of attractiveness.

II.5 Compatibility

The ontological representation of the metamodel allows to ensure a certain level of compatibility with other ontological representations. However, certain equivalency rules should be established to ensure that there is a logical match between the concepts present in the CI-CS metamodel and other metamodels. That way, it is not

possible to assess any level of compatibility between the proposal and other proposals, due to the wide range of different concepts and relationships that could be present in other proposals related to the communication field. Anyhow, it should be able to **co-exist** with other independent metamodels in a common environment sharing common concepts. Moreover, its specificity is very low (i.e., it is computation-independent), which contributes to assess a certain level of **replaceability**, that is, to be used in place of another specified metamodel for the same purpose in the same environment. Since both characteristics are associated with the compatibility in the ISO/IEC 25010:2011 standard, then, at least, a minimum level of compatibility can be attributed to the metamodel.

II.6 Maintainability

The maintainability of the metamodel can be analyzed through the conjunction of the characteristics described in the following subsections, which intend to explain how the metamodel can be modified or analyzed. The resulting conclusion is that the metamodel can be considered to be highly maintainable.

It is worth to be mentioned that during the analysis of the maintainability it is necessary to treat the metamodel as a *white box*. In contrast, the analysis of the transferability characteristic, which will be explored in Subsection II.7, requires to treat it as a whole or as a *black box* [13].

II.6.1. Modularity

The modularity is “a continuum describing the degree to which a systems components may be separated and recombined” [108]. To this respect, the metamodel have been split into two different views, that is, the structural and the behavioral ones. The concepts associated with one view can be used “as is”, without taking into account the concepts present in the other view. Therefore, the metamodel can be divided into two completely different metamodels to tackle with the different aspects of a communication system in a separated way.

In fact, the structural view could be used to represent the structure of a communication system, whereas the behavioral view, which, in turn, is based on the BPMN 2.0 choreography metamodel, could be used to model a choreography or a certain organization of a communication system. In consequence, the metamodel can be considered to be at a good modularity level.

II.6.2. Reusability

The reusability is the degree to which the metamodel can be used in more than one software system, or in building other metamodels. The level of reusability of the CI-CS metamodel can be considered to be high, since it is computation-independent (i.e., it can be used in multiple software systems and, even, to model human-based communications) and it conceptualizes a very general notion (i.e., the notion of a communication system) that is used as a part of many different types of systems. Moreover, it can be used as a basis to build more complex metamodels, as the metamodel of a ubiquitous system, which, in particular, is explored in Chapter 4.

II.6.3. Analyzability

The analyzability is the capability of the metamodel to be diagnosed for deficiencies or causes of failures, or for the parts to be modified to be identified. In this case, the metamodel has been divided into two different views, namely, the structural and the behavioral ones, which could contribute to more easily analyze its multiple concepts under different perspectives.

II.6.4. Changeability

The changeability is the capability of the metamodel to enable a specified modification to be implemented. In this case, the changeability level of the CI-CS metamodel is low, since a change in one concept or relationship will probably require a change in many other concepts and relationships, due to the cohesiveness that all

the notions related to a communication system have. Nonetheless, the modularity can be considered a characteristic that contributes to the changeability [103]. Therefore, at least, a minimum degree of changeability has been achieved in the proposal.

II.6.5. Testability

The testability is the capability of the metamodel to enable a modified (meta)model to be validated. Since it has been formally defined as an ontology, it is possible to check for inconsistencies in the models derived from the metamodel. Furthermore, the testability is also associated with the documentability [13], which is the degree to which the metamodel is documented or self-explanatory. In this case, most of the notions specified in the metamodel are well-known concepts. Anyway, all the concepts and relationships present in the metamodel (and the ideas behind their proposal) have been described in Section 3.2. Thereby, the metamodel can be considered to be testable.

II.7 Transferability

The degree of transferability of the metamodel can be analyzed through the study of the characteristics described in the following subsections, which are intended to describe how the metamodel can be adapted or transferred between environments as a whole (i.e., as a *black box*). As will be detailed, the transferability of the metamodel can only be considered to be average, since the portability degree of the proposal can not be completely assessed.

II.7.1. Adaptability

The adaptability is the capability of the metamodel to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. The adaptability is a conjunction of the flexibility, scalability and reducibility of a metamodel [13]:

- **Flexibility:** As specified in Section 3.2, the proposal is flexible enough to be considered computation-independent and to be able to represent the communication scenarios analyzed in the existing communication theories, plus some additional, very special ones (i.e., only passive or active participants, peer-to-peer environments, scenarios with only one participant in order to model self-communications, etc.).
- **Scalability:** The scalability of a metamodel refers to the possibility of using it to model both large and small systems [93]. In this case, the CI-CS metamodel is able to conceptualize communication systems with a minimum complexity (i.e., one participant communicating with itself) and very complex ones, involving multiple sets of interactions organized in very complex manners (i.e., with different gateways, several events, etc.).
- **Reducibility:** The reducibility is characterized by the amount of concepts that are present only to support the modeling of complex systems [113]. It refers to the possibility of removing certain concepts in the metamodel to only tackle with simpler systems. In this case, the CI-CS metamodel contains some concepts that are oriented towards supporting complex systems, namely, events, gateway and sub-activities. That way, the metamodel can be considered to be *reducible*.

As a consequence of the analysis of previous characteristics, the CI-CS metamodel can be considered to be adaptable.

II.7.2. Portability

The portability is the ease with which the metamodel can be transferred from one environment to another. Also, a metamodel is considered to be portable if it is adaptable and replaceable [13]. The adaptability level of the metamodel has been described to be good in previous subsection, while its replaceability was discussed to be minimal in subsection II.5. Therefore, the metamodel exhibits a minimum level of portability.

III. Implementation of the PI-US Ontology in OWL

```

<?xml version="1.0"?>

<!DOCTYPE Ontology [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#"
    >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-
    -ns#" >
]>

<Ontology xmlns="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.semanticweb.org/
  carlosrodriguezdominguez/ontologies/2014/1/
  ubiquitous_system"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
  ns#"
  xmlns:xml="http://www.w3.org/XML/1998/namespace"
  ontologyIRI="http://www.semanticweb.org/
  carlosrodriguezdominguez/ontologies/2014/1/
  ubiquitous_system">
  <Prefix name="" IRI="http://www.w3.org/2002/07/owl#"
    />
  <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl
    #"/>
  <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-
    rdf-syntax-ns#"/>
  <Prefix name="xsd" IRI="http://www.w3.org/2001/
    XMLSchema#"/>
  <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/
    rdf-schema#"/>
  <Annotation>

```

```
<AnnotationProperty abbreviatedIRI="rdfs:comment"
  />
<Literal datatypeIRI="&xsd:string">An ontology to
  represent Platform-Independent Ubiquitous
  Systems.</Literal>
</Annotation>
<Annotation>
  <AnnotationProperty abbreviatedIRI="owl:
    versionInfo"/>
  <Literal datatypeIRI="&xsd:string">1.0</Literal>
</Annotation>
<Declaration>
  <Class IRI="#Application"/>
</Declaration>
<Declaration>
  <Class IRI="#Choreography"/>
</Declaration>
<Declaration>
  <Class IRI="#CommunicationActivity"/>
</Declaration>
<Declaration>
  <Class IRI="#Complex"/>
</Declaration>
<Declaration>
  <Class IRI="#Conditional"/>
</Declaration>
<Declaration>
  <Class IRI="#Content-Based"/>
</Declaration>
<Declaration>
  <Class IRI="#Default"/>
</Declaration>
<Declaration>
  <Class IRI="#Discovery"/>
</Declaration>
<Declaration>
  <Class IRI="#DiscoveryActivity"/>
</Declaration>
<Declaration>
  <Class IRI="#DiscoveryHandler"/>
</Declaration>
```

```
<Declaration>
  <Class IRI="#DiscoveryListener"/>
</Declaration>
<Declaration>
  <Class IRI="#ElementalCommunicationActivity"/>
</Declaration>
<Declaration>
  <Class IRI="#EndEvent"/>
</Declaration>
<Declaration>
  <Class IRI="#Event"/>
</Declaration>
<Declaration>
  <Class IRI="#Event-Based"/>
</Declaration>
<Declaration>
  <Class IRI="#EventDistributionActivity"/>
</Declaration>
<Declaration>
  <Class IRI="#EventHandler"/>
</Declaration>
<Declaration>
  <Class IRI="#EventListener"/>
</Declaration>
<Declaration>
  <Class IRI="#EventNode"/>
</Declaration>
<Declaration>
  <Class IRI="#Exclusive"/>
</Declaration>
<Declaration>
  <Class IRI="#Gateway"/>
</Declaration>
<Declaration>
  <Class IRI="#Inclusive"/>
</Declaration>
<Declaration>
  <Class IRI="#InitialEvent"/>
</Declaration>
<Declaration>
  <Class IRI="#MessageExchangingActivity"/>
```



```
</Declaration>
<Declaration>
  <Class IRI="#NetworkingTechnology"/>
</Declaration>
<Declaration>
  <Class IRI="#Parallel"/>
</Declaration>
<Declaration>
  <Class IRI="#Predicate"/>
</Declaration>
<Declaration>
  <Class IRI="#RequestMessage"/>
</Declaration>
<Declaration>
  <Class IRI="#ResponseMessage"/>
</Declaration>
<Declaration>
  <Class IRI="#Sequential"/>
</Declaration>
<Declaration>
  <Class IRI="#Service"/>
</Declaration>
<Declaration>
  <Class IRI="#SoftwareAgent"/>
</Declaration>
<Declaration>
  <Class IRI="#SoftwareMessage"/>
</Declaration>
<Declaration>
  <Class IRI="#SoftwareProtocol"/>
</Declaration>
<Declaration>
  <Class IRI="#Topic"/>
</Declaration>
<Declaration>
  <Class IRI="#UbiquitousSystem"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#hasActivityParticipants"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#hasChoreography"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasChoreographyMessages"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasChoreographyParticipants"
    />
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasCommunicationActivity"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasConditionalPredicate"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasElementalActivities"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasEventHandler"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasEventListener"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasParticipants"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasPredicate"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasProtocols"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#"
    hasPublicationOrSubscriptionOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasTechnologies"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#hasUnsatisfiedPredicate"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isActivityParticipant"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyMessageOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isChoreographyParticipant"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isCommunicationActivityOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isCommunicativeCommonalityOf
    "/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isComplierWith"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isComposedBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isConnectedTo"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isDiscoveredBy"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isDiscovererOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isElementalActivityOf"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#isEventHandlerOf"/>
```

```
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isEventListenerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isEventNodeOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isExchangedDuring"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isExchangerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isFiltererOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isFinalizedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isFinalizerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isInitializedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isInitializerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isLinkOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isListenedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isListenerOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isMediumOf"/>
</Declaration>
<Declaration>
```

```
    <ObjectProperty IRI="#isNotifiedBy"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isParticipantOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isPredicateOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isProtocolOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isPublishedSubscribedDuring"
      />
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isRequestedBy"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isRequesterOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isRequesterOfResponse"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isResponseOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isSemanticallyRelatedTo"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isSemanticsOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isStarterOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isStructurerOf"/>
  </Declaration>
  <Declaration>
    <ObjectProperty IRI="#isSubtopicOf"/>
```

```

</Declaration>
<Declaration>
  <ObjectProperty IRI="#isTechnologyOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isTransferUnitOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isTransportedBy"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isTransporterOf"/>
</Declaration>
<Declaration>
  <ObjectProperty IRI="#isTriggeredBy"/>
</Declaration>
<SubClassOf>
  <Class IRI="#Application"/>
  <Class IRI="#SoftwareAgent"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Application"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isRequesterOf"/>
    <Class IRI="#Service"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Choreography"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasChoreographyMessages"
      />
    <Class IRI="#SoftwareMessage"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Choreography"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#
      hasChoreographyParticipants"/>
    <Class IRI="#SoftwareAgent"/>
  </ObjectSomeValuesFrom>
</SubClassOf>

```

```
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasCommunicationActivity
        "/>
      <Class IRI="#CommunicationActivity"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isChoreographyOf"/>
      <Class IRI="#UbiquitousSystem"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isFinalizedBy"/>
      <Class IRI="#EndEvent"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Choreography"/>
    <ObjectExactCardinality cardinality="1">
      <ObjectProperty IRI="#isInitializedBy"/>
      <Class IRI="#InitialEvent"/>
    </ObjectExactCardinality>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationActivity"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasActivityParticipants"
        />
      <Class IRI="#SoftwareAgent"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationActivity"/>
```

```

    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasChoreographyMessages"
        />
      <Class IRI="#SoftwareMessage"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationActivity"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasElementalActivities"
        />
      <Class IRI="#ElementalCommunicationActivity"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#CommunicationActivity"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#"
        isCommunicationActivityOf"/>
      <Class IRI="#Choreography"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Complex"/>
    <Class IRI="#Gateway"/>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Conditional"/>
    <Class IRI="#Event"/>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Conditional"/>
    <ObjectExactCardinality cardinality="1">
      <ObjectProperty IRI="#hasConditionalPredicate"
        />
      <Class IRI="#Predicate"/>
    </ObjectExactCardinality>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#Content-Based"/>
    <Class IRI="#Exclusive"/>

```



```
</SubClassOf>
<SubClassOf>
  <Class IRI="#Default"/>
  <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Default"/>
  <ObjectExactCardinality cardinality="1">
    <ObjectProperty IRI="#hasUnsatisfiedPredicate"
      />
    <Class IRI="#Predicate"/>
  </ObjectExactCardinality>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Discovery"/>
  <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Discovery"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isNotifiedBy"/>
    <Class IRI="#DiscoveryHandler"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#DiscoveryActivity"/>
  <Class IRI="#EventDistributionActivity"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#DiscoveryActivity"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#"
      hasPublicationOrSubscriptionOf"/>
    <Class IRI="#Discovery"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#DiscoveryHandler"/>
  <Class IRI="#EventHandler"/>
</SubClassOf>
<SubClassOf>
```

```

    <Class IRI="#DiscoveryHandler"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isDiscovererOf"/>
      <Class IRI="#DiscoveryListener"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#DiscoveryListener"/>
  <Class IRI="#EventListener"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#DiscoveryListener"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isListenerOf"/>
    <Class IRI="#Discovery"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#ElementalCommunicationActivity"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isElementalActivityOf"/>
    <Class IRI="#CommunicationActivity"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EndEvent"/>
  <Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EndEvent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isFinalizerOf"/>
    <Class IRI="#Choreography"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Event"/>
  <Class IRI="#SoftwareMessage"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Event"/>

```

```
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isNotifiedBy"/>
      <Class IRI="#EventHandler"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
<SubClassOf>
  <Class IRI="#Event"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isSemanticallyRelatedTo"
      />
    <Class IRI="#Topic"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Event-Based"/>
  <Class IRI="#Exclusive"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EventDistributionActivity"/>
  <Class IRI="#ElementalCommunicationActivity"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EventDistributionActivity"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#"
      hasPublicationOrSubscriptionOf"/>
    <Class IRI="#Event"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EventHandler"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isEventHandlerOf"/>
    <Class IRI="#SoftwareAgent"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#EventListener"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasPredicate"/>
    <Class IRI="#Predicate"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
```

```
        </ObjectSomeValuesFrom>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#EventListener"/>
        <ObjectSomeValuesFrom>
            <ObjectProperty IRI="#isListenerOf"/>
            <Class IRI="#Event"/>
        </ObjectSomeValuesFrom>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#EventNode"/>
        <Class IRI="#Event"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#EventNode"/>
        <ObjectSomeValuesFrom>
            <ObjectProperty IRI="#isEventNodeOf"/>
            <Class IRI="#Event"/>
        </ObjectSomeValuesFrom>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Exclusive"/>
        <Class IRI="#Gateway"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Exclusive"/>
        <ObjectSomeValuesFrom>
            <ObjectProperty IRI="#isTriggeredBy"/>
            <Class IRI="#Event"/>
        </ObjectSomeValuesFrom>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Gateway"/>
        <Class IRI="#Event"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Inclusive"/>
        <Class IRI="#Gateway"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#InitialEvent"/>
```

```
<Class IRI="#Event"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#InitialEvent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isInitializerOf"/>
    <Class IRI="#Choreography"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#MessageExchangingActivity"/>
  <Class IRI="#ElementalCommunicationActivity"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#MessageExchangingActivity"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isExchangerOf"/>
    <Class IRI="#RequestMessage"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#NetworkingTechnology"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isMediumOf"/>
    <Class IRI="#SoftwareProtocol"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#NetworkingTechnology"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isTechnologyOf"/>
    <Class IRI="#UbiquitousSystem"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#NetworkingTechnology"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isTransporterOf"/>
    <Class IRI="#SoftwareMessage"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
```

```

<SubClassOf>
  <Class IRI="#Parallel"/>
  <Class IRI="#Gateway"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Predicate"/>
  <ObjectUnionOf>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isFiltererOf"/>
      <Class IRI="#Event"/>
    </ObjectSomeValuesFrom>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#isPredicateOf"/>
      <Class IRI="#EventListener"/>
    </ObjectSomeValuesFrom>
  </ObjectUnionOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#RequestMessage"/>
  <Class IRI="#SoftwareMessage"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#RequestMessage"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isExchangedDuring"/>
    <Class IRI="#MessageExchangingActivity"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#RequestMessage"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isRequesterOfResponse"/>
    <Class IRI="#ResponseMessage"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#ResponseMessage"/>
  <Class IRI="#SoftwareMessage"/>
</SubClassOf>
<SubClassOf>
  <Class IRI="#ResponseMessage"/>

```

```
    <ObjectMinCardinality cardinality="1">
      <ObjectProperty IRI="#isResponseOf"/>
      <Class IRI="#RequestMessage"/>
    </ObjectMinCardinality>
  </SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Sequential"/>
  <Class IRI="#Event"/>
</SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Service"/>
  <Class IRI="#SoftwareAgent"/>
</SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Service"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isRequestedBy"/>
    <Class IRI="#Application"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SoftwareAgent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isActivityParticipant"/>
    <Class IRI="#CommunicationActivity"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SoftwareAgent"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isParticipantOf"/>
    <Class IRI="#UbiquitousSystem"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SoftwareMessage"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isComplierWith"/>
    <Class IRI="#SoftwareProtocol"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
```

```

<SubClassOf>
  <Class IRI="#SoftwareMessage"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isTransferUnitOf"/>
    <Class IRI="#SoftwareAgent"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SoftwareProtocol"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#
      isCommunicativeCommonalityOf"/>
    <Class IRI="#NetworkingTechnology"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#SoftwareProtocol"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isProtocolOf"/>
    <Class IRI="#UbiquitousSystem"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#Topic"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#isSemanticsOf"/>
    <Class IRI="#Event"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#UbiquitousSystem"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasChoreography"/>
    <Class IRI="#Choreography"/>
  </ObjectSomeValuesFrom>
</SubClassOf>
<SubClassOf>
  <Class IRI="#UbiquitousSystem"/>
  <ObjectSomeValuesFrom>
    <ObjectProperty IRI="#hasProtocols"/>
    <Class IRI="#SoftwareProtocol"/>
  </ObjectSomeValuesFrom>
</SubClassOf>

```



```
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#UbiquitousSystem"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasTechnologies"/>
      <Class IRI="#NetworkingTechnology"/>
    </ObjectSomeValuesFrom>
  </SubClassOf>
  <SubClassOf>
    <Class IRI="#UbiquitousSystem"/>
    <ObjectMinCardinality cardinality="2">
      <ObjectProperty IRI="#hasParticipants"/>
      <Class IRI="#SoftwareAgent"/>
    </ObjectMinCardinality>
  </SubClassOf>
  <DisjointClasses>
    <Class IRI="#Application"/>
    <Class IRI="#Service"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#Choreography"/>
    <Class IRI="#CommunicationActivity"/>
    <Class IRI="#ElementalCommunicationActivity"/>
    <Class IRI="#EventHandler"/>
    <Class IRI="#EventListener"/>
    <Class IRI="#NetworkingTechnology"/>
    <Class IRI="#Predicate"/>
    <Class IRI="#SoftwareAgent"/>
    <Class IRI="#SoftwareMessage"/>
    <Class IRI="#SoftwareProtocol"/>
    <Class IRI="#Topic"/>
    <Class IRI="#UbiquitousSystem"/>
  </DisjointClasses>
  <DisjointClasses>
    <Class IRI="#Complex"/>
    <Class IRI="#Exclusive"/>
    <Class IRI="#Inclusive"/>
    <Class IRI="#Parallel"/>
  </DisjointClasses>
  <DisjointClasses>
```

```

    <Class IRI="#Conditional"/>
    <Class IRI="#Default"/>
    <Class IRI="#Discovery"/>
    <Class IRI="#EndEvent"/>
    <Class IRI="#EventNode"/>
    <Class IRI="#Gateway"/>
    <Class IRI="#InitialEvent"/>
    <Class IRI="#Sequential"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Content-Based"/>
    <Class IRI="#Event-Based"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Event"/>
    <Class IRI="#RequestMessage"/>
    <Class IRI="#ResponseMessage"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#EventDistributionActivity"/>
    <Class IRI="#MessageExchangingActivity"/>
</DisjointClasses>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasActivityParticipants"/>
    <ObjectProperty IRI="#isActivityParticipant"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasChoreography"/>
    <ObjectProperty IRI="#isChoreographyOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isChoreographyMessageOf"/>
    <ObjectProperty IRI="#hasChoreographyMessages"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasChoreographyParticipants"
    />
    <ObjectProperty IRI="#isChoreographyParticipant"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isCommunicationActivityOf"/>

```

```
    <ObjectProperty IRI="#hasCommunicationActivity"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasElementalActivities"/>
    <ObjectProperty IRI="#isElementalActivityOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isEventHandlerOf"/>
    <ObjectProperty IRI="#hasEventHandler"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasEventListener"/>
    <ObjectProperty IRI="#isEventListenerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isParticipantOf"/>
    <ObjectProperty IRI="#hasParticipants"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasPredicate"/>
    <ObjectProperty IRI="#isPredicateOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isProtocolOf"/>
    <ObjectProperty IRI="#hasProtocols"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isPublishedSubscribedDuring"
        />
    <ObjectProperty IRI="#
        hasPublicationOrSubscriptionOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#hasTechnologies"/>
    <ObjectProperty IRI="#isTechnologyOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
    <ObjectProperty IRI="#isMediumOf"/>
    <ObjectProperty IRI="#isCommunicativeCommonalityOf
        "/>
</InverseObjectProperties>
```

```
<InverseObjectProperties>
  <ObjectProperty IRI="#isComplierWith"/>
  <ObjectProperty IRI="#isStructurerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isDiscoveredBy"/>
  <ObjectProperty IRI="#isDiscovererOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isExchangedDuring"/>
  <ObjectProperty IRI="#isExchangerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isFinalizedBy"/>
  <ObjectProperty IRI="#isFinalizerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isInitializedBy"/>
  <ObjectProperty IRI="#isInitializerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isListenedBy"/>
  <ObjectProperty IRI="#isListenerOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isRequestedBy"/>
  <ObjectProperty IRI="#isRequesterOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isRequesterOfResponse"/>
  <ObjectProperty IRI="#isResponseOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isSemanticallyRelatedTo"/>
  <ObjectProperty IRI="#isSemanticsOf"/>
</InverseObjectProperties>
<InverseObjectProperties>
  <ObjectProperty IRI="#isTransportedBy"/>
  <ObjectProperty IRI="#isTransporterOf"/>
</InverseObjectProperties>
<FunctionalObjectProperty>
```

```
    <ObjectProperty IRI="#isInitializedBy"/>
  </FunctionalObjectProperty>
  <FunctionalObjectProperty>
    <ObjectProperty IRI="#isTriggeredBy"/>
  </FunctionalObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#hasChoreography"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#hasCommunicationActivity"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#hasEventHandler"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#hasEventListener"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#"
      hasPublicationOrSubscriptionOf"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isConnectedTo"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isDiscoveredBy"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isEventHandlerOf"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isEventListenerOf"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isListenedBy"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isListenerOf"/>
  </SymmetricObjectProperty>
  <SymmetricObjectProperty>
    <ObjectProperty IRI="#isPredicateOf"/>
```

```

</SymmetricObjectProperty>
<SymmetricObjectProperty>
  <ObjectProperty IRI="#isPublishedSubscribedDuring"
  />
</SymmetricObjectProperty>
<SymmetricObjectProperty>
  <ObjectProperty IRI="#isRequestedBy"/>
</SymmetricObjectProperty>
<SymmetricObjectProperty>
  <ObjectProperty IRI="#isRequesterOf"/>
</SymmetricObjectProperty>
<AsymmetricObjectProperty>
  <ObjectProperty IRI="#isSubtopicOf"/>
</AsymmetricObjectProperty>
<TransitiveObjectProperty>
  <ObjectProperty IRI="#isComposedBy"/>
</TransitiveObjectProperty>
<TransitiveObjectProperty>
  <ObjectProperty IRI="#isConnectedTo"/>
</TransitiveObjectProperty>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasActivityParticipants"/>
  <Class IRI="#CommunicationActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChoreography"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChoreographyMessages"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasChoreographyParticipants"
  />
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasCommunicationActivity"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>

```

```
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasConditionalPredicate"/>
  <Class IRI="#Conditional"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasElementalActivities"/>
  <Class IRI="#CommunicationActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasEventHandler"/>
  <Class IRI="#SoftwareAgent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasEventListener"/>
  <Class IRI="#EventHandler"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasParticipants"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasPredicate"/>
  <Class IRI="#EventListener"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasProtocols"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#
    hasPublicationOrSubscriptionOf"/>
  <Class IRI="#EventDistributionActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasTechnologies"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#hasUnsatisfiedPredicate"/>
  <Class IRI="#Default"/>
</ObjectPropertyDomain>
```

```
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isActivityParticipant"/>
  <Class IRI="#SoftwareAgent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isChoreographyMessageOf"/>
  <Class IRI="#SoftwareMessage"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isChoreographyOf"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isChoreographyParticipant"/>
  <Class IRI="#SoftwareAgent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isCommunicationActivityOf"/>
  <Class IRI="#CommunicationActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isCommunicativeCommonalityOf
  "/>
  <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isComplierWith"/>
  <Class IRI="#SoftwareMessage"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isComposedBy"/>
  <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isConnectedTo"/>
  <Class IRI="#Service"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isDiscoveredBy"/>
  <Class IRI="#DiscoveryListener"/>
</ObjectPropertyDomain>
```



```
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isDiscovererOf"/>
  <Class IRI="#DiscoveryHandler"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isElementalActivityOf"/>
  <Class IRI="#ElementalCommunicationActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isEventHandlerOf"/>
  <Class IRI="#EventHandler"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isEventListenerOf"/>
  <Class IRI="#EventListener"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isEventNodeOf"/>
  <Class IRI="#EventNode"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isExchangedDuring"/>
  <Class IRI="#MessageExchangingActivity"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isExchangerOf"/>
  <Class IRI="#RequestMessage"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isFiltererOf"/>
  <Class IRI="#Predicate"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isFinalizedBy"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
  <ObjectProperty IRI="#isFinalizerOf"/>
  <Class IRI="#EndEvent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
```

```
    <ObjectProperty IRI="#isInitializedBy"/>
    <Class IRI="#Choreography"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isInitializerOf"/>
    <Class IRI="#InitialEvent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isLinkOf"/>
    <Class IRI="#SoftwareMessage"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isListenedBy"/>
    <Class IRI="#Event"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isListenerOf"/>
    <Class IRI="#EventListener"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isMediumOf"/>
    <Class IRI="#NetworkingTechnology"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isNotifiedBy"/>
    <Class IRI="#EventHandler"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isParticipantOf"/>
    <Class IRI="#SoftwareAgent"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isPredicateOf"/>
    <Class IRI="#Predicate"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
    <ObjectProperty IRI="#isProtocolOf"/>
    <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyDomain>
<ObjectPropertyDomain>
```

```
    <ObjectProperty IRI="#isPublishedSubscribedDuring"
      />
    <Class IRI="#Event"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isRequestedBy"/>
    <Class IRI="#Service"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isRequesterOf"/>
    <Class IRI="#Application"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isRequesterOfResponse"/>
    <Class IRI="#RequestMessage"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isResponseOf"/>
    <Class IRI="#ResponseMessage"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isSemanticallyRelatedTo"/>
    <Class IRI="#Event"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isSemanticsOf"/>
    <Class IRI="#Topic"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isStarterOf"/>
    <Class IRI="#Gateway"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isStructurerOf"/>
    <Class IRI="#SoftwareProtocol"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
    <ObjectProperty IRI="#isSubtopicOf"/>
    <Class IRI="#Topic"/>
  </ObjectPropertyDomain>
  <ObjectPropertyDomain>
```

```

        <ObjectProperty IRI="#isTechnologyOf"/>
        <Class IRI="#NetworkingTechnology"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#isTransferUnitOf"/>
        <Class IRI="#SoftwareMessage"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#isTransportedBy"/>
        <Class IRI="#SoftwareMessage"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#isTransporterOf"/>
        <Class IRI="#NetworkingTechnology"/>
    </ObjectPropertyDomain>
    <ObjectPropertyDomain>
        <ObjectProperty IRI="#isTriggeredBy"/>
        <Class IRI="#Exclusive"/>
    </ObjectPropertyDomain>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasActivityParticipants"/>
        <Class IRI="#SoftwareAgent"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasChoreography"/>
        <Class IRI="#Choreography"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasChoreographyMessages"/>
        <Class IRI="#SoftwareMessage"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasChoreographyParticipants"
        />
        <Class IRI="#SoftwareAgent"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>
        <ObjectProperty IRI="#hasCommunicationActivity"/>
        <Class IRI="#CommunicationActivity"/>
    </ObjectPropertyRange>
    <ObjectPropertyRange>

```

```
    <ObjectProperty IRI="#hasConditionalPredicate"/>
    <Class IRI="#Predicate"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasElementalActivities"/>
    <Class IRI="#ElementalCommunicationActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasEventHandler"/>
    <Class IRI="#EventHandler"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasEventListener"/>
    <Class IRI="#EventListener"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasParticipants"/>
    <Class IRI="#SoftwareAgent"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasPredicate"/>
    <Class IRI="#Predicate"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasProtocols"/>
    <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#"
        hasPublicationOrSubscriptionOf"/>
    <Class IRI="#Event"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasTechnologies"/>
    <Class IRI="#NetworkingTechnology"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#hasUnsatisfiedPredicate"/>
    <Class IRI="#Predicate"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
```

```
<ObjectProperty IRI="#isActivityParticipant"/>
  <Class IRI="#CommunicationActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isChoreographyMessageOf"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isChoreographyOf"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isChoreographyParticipant"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isCommunicationActivityOf"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isCommunicativeCommonalityOf
  "/>
  <Class IRI="#NetworkingTechnology"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isComplierWith"/>
  <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isComposedBy"/>
  <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isConnectedTo"/>
  <Class IRI="#Service"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isDiscoveredBy"/>
  <Class IRI="#DiscoveryHandler"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
```

```
    <ObjectProperty IRI="#isDiscovererOf"/>
    <Class IRI="#DiscoveryListener"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isElementalActivityOf"/>
    <Class IRI="#CommunicationActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isEventHandlerOf"/>
    <Class IRI="#SoftwareAgent"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isEventListenerOf"/>
    <Class IRI="#EventHandler"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isEventNodeOf"/>
    <Class IRI="#Event"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isExchangedDuring"/>
    <Class IRI="#RequestMessage"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isExchangerOf"/>
    <Class IRI="#MessageExchangingActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isFiltererOf"/>
    <Class IRI="#Event"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isFinalizedBy"/>
    <Class IRI="#EndEvent"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isFinalizerOf"/>
    <Class IRI="#Choreography"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
    <ObjectProperty IRI="#isInitializedBy"/>
```

```

    <Class IRI="#InitialEvent"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isInitializerOf"/>
  <Class IRI="#Choreography"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isLinkOf"/>
  <Class IRI="#CommunicationActivity"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isListenedBy"/>
  <Class IRI="#EventListener"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isListenerOf"/>
  <Class IRI="#Event"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isMediumOf"/>
  <Class IRI="#SoftwareProtocol"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isNotifiedBy"/>
  <Class IRI="#Event"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isParticipantOf"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isPredicateOf"/>
  <Class IRI="#EventListener"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isProtocolOf"/>
  <Class IRI="#UbiquitousSystem"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isPublishedSubscribedDuring"
    />

```



```
    <Class IRI="#EventDistributionActivity"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isRequestedBy"/>
    <Class IRI="#Application"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isRequesterOf"/>
    <Class IRI="#Service"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isRequesterOfResponse"/>
    <Class IRI="#ResponseMessage"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isResponseOf"/>
    <Class IRI="#RequestMessage"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isSemanticallyRelatedTo"/>
    <Class IRI="#Topic"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isSemanticsOf"/>
    <Class IRI="#Event"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isStarterOf"/>
    <Class IRI="#CommunicationActivity"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isStructurerOf"/>
    <Class IRI="#SoftwareMessage"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isSubtopicOf"/>
    <Class IRI="#Topic"/>
  </ObjectPropertyRange>
  <ObjectPropertyRange>
    <ObjectProperty IRI="#isTechnologyOf"/>
    <Class IRI="#UbiquitousSystem"/>
  </ObjectPropertyRange>
```

```
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransferUnitOf"/>
  <Class IRI="#SoftwareAgent"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransportedBy"/>
  <Class IRI="#NetworkingTechnology"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTransporterOf"/>
  <Class IRI="#SoftwareMessage"/>
</ObjectPropertyRange>
<ObjectPropertyRange>
  <ObjectProperty IRI="#isTriggeredBy"/>
  <Class IRI="#Event"/>
</ObjectPropertyRange>
</Ontology>
```

IV. Quality Attributes of the PI-US Metamodel

IV.1 Functional Suitability

The functional suitability is the degree to which the metamodel provides functions that meet stated and implied needs when the metamodel is used under specified conditions.

To this respect, the proposed PI-US metamodel is able to represent ubiquitous systems in which multiple applications and services interact to achieve certain goals. The metamodel takes into account the need of integrating multiple paradigms and technologies to fulfill the expected properties of a ubiquitous system, which were described in Section 4.1. For example, the behavioral view has been devised to consider the mobility of the software agents and the volatility of the communications between them.

Additionally, the metamodel has artifacts to support the specification of the three communication functionalities that should be present in any ubiquitous system: message exchanging, event distribution and dynamic discovery.

In that sense, the metamodel can also be considered to have **functional appropriateness**, that is, it provides an appropriate set of functions for specified tasks and user objectives (i.e., to support communications in ubiquitous systems). It can also be considered to be **concise**, since it *just* contains the needed concepts and relationships to represent the communication mechanisms of a ubiquitous system, which have been identified by studying previous research works and some communication standards and well-known middleware. Moreover, the metamodel is **cohesive** (i.e., it is only focused on one topic, that is, in the description of the structure, behavior and organization of the communication mechanisms in a ubiquitous system) and **confined** (i.e., it only contains concepts at one abstraction level). Also, it can be considered to be **relevant**, that is, it only contains the concepts and relationships that are necessary for a particular transformation (i.e., from a platform-independent ubiquitous

system to a platform-specific one). This aspect of the metamodel is explored in Chapter 5.

Furthermore, the ontological representation of the metamodel allows to formally and automatically (through a reasoner) check that all the requirements of a ubiquitous system are fulfilled, given the set of concepts in the structural view and an specification of a choreography through the concepts in the behavioral view.

Finally, the functional suitability of a metamodel can be also defined on the basis of a set of sub-characteristics described in the ISO/IEC 25010:2011 standard. The following subsections explain how the PI-US metamodel is able to accomplish those sub-characteristics. The resulting conclusion is that the functional suitability of the metamodel can be considered to be high. As in the analysis of the quality properties of the CI-CS metamodel (see Chapter 3, Section 3.3), the security sub-characteristic has not been taken into account, since it is not suitable to describe the quality of a metamodel [13].

IV.1.1. Accuracy

The accuracy is the degree to which the metamodel provides the right or specified results with the needed degree of precision. One related aspect is the **validity**, that is, the metamodel contains information that has been contrasted against a reliable source of information. Consequently, the PI-US metamodel can be considered to be *valid* since it is based on some standards and well-known middleware. Finally, the previous subsection described the metamodel as **relevant**, which is directly related to the degree of **precision** of the metamodel. The precision is another attribute that contributes to the accuracy of the metamodel.

IV.1.2. Interoperability

The interoperability is the degree to which the metamodel can be cooperatively operable with one or more other metamodels. Since the PI-US metamodel is formally specified through an ontology, it

can interoperate with other ontologies specifying other metamodels in order to complete its information in some aspects, to include new notions related to other domains, etc. Moreover, the PI-US metamodel can interoperate with the CI-CS metamodel, as it is shown in Chapter 4, Section 4.2.4.

IV.1.3. Compliance

The functional suitability compliance is the degree to which the metamodel adheres to standards, conventions or regulations in laws and similar prescriptions relating to functional suitability. As it was previously mentioned, the PI-US metamodel is based on some communication standards and well-known middleware solutions. Additionally, it includes several notions (like *event handler*) that can also be found in other specifications related to the communications in ubiquitous systems. Hence, the metamodel can be considered to have a good degree of compliance.

IV.2 Reliability

The reliability is the degree to which the metamodel can maintain a specified level of performance when used under specified conditions. In other words, the reliability of the metamodel refers to the degree of expressiveness that can be assessed, even when a fault in the metamodel specification is detected. Additionally, the reliability is connected to the functional suitability compliance and the accuracy. In this case, as described in IV.1 and IV.1.1, the reliability can be considered acceptable, since the PI-US metamodel includes concepts and relationships that are widely present in well-known standards and middleware solutions. Hence, even if a failure is detected in the metamodel specification, it includes concepts that are known to be correct (request and response messages, event, event handler, event listener, software protocol, networking technology, application, service, choreography, predicate, topic, discoverer and discovery listener), which ensures a certain level of expressiveness to represent the platform-independent communication mechanisms

of a ubiquitous system.

IV.3 Performance Efficiency

The performance efficiency is the degree to which the metamodel provides appropriate performance, relative to the amount of resources used, under stated conditions. It is related to the amount of classes and relationships that it uses. Therefore, it is related to the *conciseness* of the metamodel (see subsection IV.1), which has been described to be good, since the metamodel *just* contains the needed concepts and relationships to represent the communication mechanisms of a ubiquitous system. Moreover, using the metamodel to produce models can be considered as an efficient task, since the produced models should only have the strictly needed concepts to represent the software mechanisms that should be present in a ubiquitous system to support communications.

IV.4 Operability

The operability is the degree to which the metamodel can be understood, learned, used and attractive to the user, when used under specified conditions. Taking into account that the operability is a very subjective property, related to the expertise or the attraction of the user of the metamodel to the field of the ubiquitous systems, it is considered to be related to its simplicity and conciseness [13]. In that sense, due to the previous analysis provided in subsection IV.1, the PI-US metamodel can be considered to have, at least, a minimum operability level. Additionally, the operability is also related to many different characteristics, as will be explored in the following subsections.

IV.4.1. Appropriateness Recognizability

The appropriateness recognizability refers to the degree to which the metamodel enables users to recognize whether it is appropriate

for their needs or not. The PI-US is only focused on representing the platform-independent software artifacts that are necessary to support the communication mechanisms that should be present in a ubiquitous system. Hence, the users of the metamodel should be able to easily recognize if it is appropriate for their needs or not. In any case, the metamodel has been formalized as an ontology, which may help users to clear up its focus, or, at least, to detect if a representation of a system conforms to the proposed conceptualization or not.

IV.4.2. Learnability

The learnability is the capability of the metamodel to enable the user to learn its application (use, meaning, representation). In this case, the learnability of the metamodel directly depends on the previous knowledge that the user may have about the field of the communications in a ubiquitous system, due to the needed expertise about communication standards, middleware solutions and some technical details. Thereby, the learnability of the PI-US metamodel can not be considered to be high.

IV.4.3. Helpfulness

The helpfulness of the metamodel is referred to the degree to which it provides help when users need assistance. The PI-US metamodel may help into analyzing, designing or validating the communication mechanisms that should be present in a ubiquitous system. Consequently, it may be considered to be helpful for these tasks.

IV.4.4. Attractiveness

The attractiveness is the capability of the metamodel to be attractive to the user. Even if this is a completely subjective attribute that is more related to the attraction of the user of the metamodel to the field of the ubiquitous systems, it can also be related to its learnability. Thus, since the metamodel can not be considered to be highly learnable, it can not be determined to have attractiveness.

IV.5 Compatibility

The compatibility is the ability of the metamodel to exchange information with other metamodels and/or to perform their required functions while sharing the same domain. Since the metamodel shares certain notions and relationships with several communication standards and middleware specifications, it should be able to interact with their corresponding metamodels. Moreover, the PI-US has been formally demonstrated to have certain relationships to a CI-CS. Therefore, both metamodels can interact. Consequently, the compatibility of the PI-US can be considered to be high.

IV.6 Maintainability

The maintainability is the degree to which the metamodel can be modified. Modifications may include corrections, improvements or adaptation of the metamodel to changes in environment, and in requirements and functional specifications.

The maintainability level of the metamodel is analyzed on the basis of the characteristics described in the following subsections. The resulting conclusion is that the metamodel can be considered to be maintainable. Anyhow, during the analysis of the maintainability it is necessary to treat the metamodel as a *white box*. In contrast, the analysis of the transferability characteristic, which will be explored in Subsection IV.7, requires to treat it as a whole or as a *black box* [13].

IV.6.1. Modularity

The modularity is “a continuum describing the degree to which a systems components may be separated and recombined” [108]. The metamodel is presented through two different views (structural and behavioral one), which contributes to have a clear focus on the concepts and relationships to represent the different aspects associated to the communication mechanisms present in a ubiquitous system. Both views are complementary, but independent. Therefore, each

view can be separately used to represent different aspects of the communications in a ubiquitous system, or combined into the whole PI-US metamodel to tackle with the representation of all the different facets of the communications in a ubiquitous system. Furthermore, since the metamodel includes concepts to support different communication functionalities (message exchanging, event distribution and dynamic discovery), these concepts could be separated from the others, in order to focus on the representation of the software-related concepts related to a unique communication functionality. Consequently, the metamodel can be considered to have a high modularity level.

IV.6.2. Reusability

The reusability is the degree to which the metamodel can be used in more than one software system, or in building other metamodels. Since the metamodel is platform-independent, it can be highly reused to model the communication mechanisms associated to a ubiquitous system. Moreover, it tackles with many notions that are commonly present in any ubiquitous system, due to the dynamic, mobile and highly structural nature of these systems. Finally, the PI-US metamodel can be used in conjunction with other metamodels related to the ubiquitous computing field to design ubiquitous systems in a holistic way, taking into account their presentational aspects, data models, etc.

IV.6.3. Analyzability

The analyzability is the capability of the metamodel to be diagnosed for deficiencies or causes of failures, or for the parts to be modified to be identified. In this case, the metamodel has been divided into two different views, namely, the structural and the behavioral ones, and the concepts supporting different communication functionalities have been clearly identified. Both aspects of the metamodel could contribute to more easily analyze its concepts under different perspectives and to modify them if necessary, which also contributes to

the **changeability** of the metamodel. The changeability is the capability of the metamodel to enable a specified modification to be implemented, and it is a characteristic that is close related to the degree of maintainability provided by the metamodel.

IV.6.4. Testability

The testability is the capability of the metamodel to enable a modified (meta)model to be validated. The formal specification of the metamodel as an ontology makes it possible to check for inconsistencies in the models derived from the metamodel, which contributes to the testability of the metamodel. The testability is also associated with the documentability [13], which is the degree to which the metamodel is documented or self-explanatory. In this case, all the concepts and relationships present in the metamodel (and the ideas behind their proposal) have been described in Chapter 4, Section 4.2. However, the concepts could not be considered to be self-explanatory, since they require an explicit definition to be understood by non-experts in the field of the communications in ubiquitous systems. Consequently, the level of testability of the metamodel is good, but can not be considered to be high.

IV.7 Transferability

The transferability is the degree to which the metamodel can be transferred from one environment to another. The degree of transferability of the metamodel has been analyzed through the study of the characteristics described in the following subsections, which intend to describe how the metamodel can be adapted or transferred between environments as a whole (i.e., as a *black box*). As will be detailed, the transferability of the metamodel can only be considered to be average, since the adaptability and portability degree of the proposal can not be completely assessed.

IV.7.1. Adaptability

The adaptability is the capability of the metamodel to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. The adaptability is a conjunction of the flexibility, scalability and reducibility of a metamodel [13]:

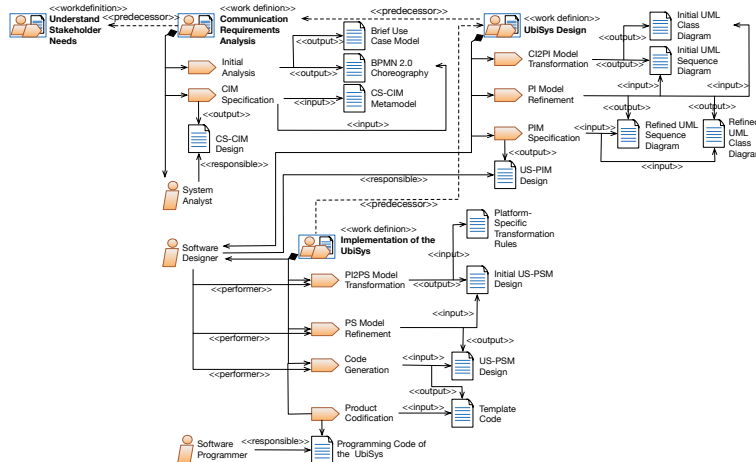
- **Flexibility:** The PI-US metamodel is flexible enough to be considered platform-independent and to be able to represent some existing standards and well-known middleware specifications, as it was described in Section 4.2.
- **Scalability:** The scalability of a metamodel refers to the possibility that it models both large and small systems [93]. In this case, the PI-US metamodel is able to represent minimal communication mechanisms for small ubiquitous systems (e.g., two software agents communicating through only one communication functionality) and very complex ones, involving multiple communication functionalities, networking technologies and other software artifacts, with a complex organization of communication activities carried out by several software agents.
- **Reducibility:** The reducibility is characterized by the amount of concepts that are present only to support the modeling of complex systems [113]. It refers to the possibility of removing certain concepts in the metamodel to only tackle with simpler systems. In this case, the PI-US metamodel merely consists of the minimum software artifacts to support the representation of communication mechanisms for ubiquitous systems. Hence, the metamodel can not be considered to be *reducible*.

As a consequence of the analysis of previous characteristics, the PI-US metamodel can only be considered to have an intermediate degree of adaptability (i.e., it is flexible and scalable, but it can not be reduced).

IV.7.2. Portability

The portability is the ease with which the metamodel can be transferred from one environment to another. A metamodel is also considered to be portable if it is adaptable and replaceable [13]. The adaptability has been described as average in previous subsection. The replaceability of the metamodel is the capability to be used in place of another specified metamodel for the same purpose in the same environment, and it is highly related with its compatibility [13]. Since the compatibility of the metamodel has been considered to be high (see Subsection IV.5), then it can be deduced that its replaceability should also be high. However, since the adaptability is intermediate, then the portability of the metamodel can only be considered to be intermediate too.

V. Detailed SPEM 2.0 diagram describing the development process proposed in MUSYC



VI. Proposed QVT rules to transform a CS-CIM into a US-PIM

```
modeltype CIM uses "http://www.ugr.es/~carlosrodriguez/
  CIM" where{self.objectsOfType(Participant)>size() >=
  2 and self.objectsOfType(Choreography)>size() >= 1};
modeltype PIM uses "http://www.ugr.es/~carlosrodriguez/
  PIM";

transformation CIM2PIM(in source:CIM, out target:PIM);

main() {
  source.rootObjects()[CommunicationSystem]>map
    toUbiquitousSystem();
}

mapping CommunicationSystem::toUbiquitousSystem() : PIM::
  UbiquitousSystem {
  name := self.name;
  agents := self.parties>map toAgent();
  var r := self.parties>map toPeerAgents();
  agents += r.app;
  agents += r.service;

  protocols := self.protocols>map toSoftwareProtocol();
  channels := self.channels>map toNetworkingTechnology
    ();
  choreographies := self.choreographies>map
    toSoftwareAgentChoreography();

  self.parties>map softwareAgentConnections();
}

mapping Participant::softwareAgentConnections() {
  self.resolve(PIM::SoftwareAgent)>forEach(agent){
    if (agent.oclIsTypeOf(PIM::Service)) then {
      var s:PIM::Service := agent.oclAsType(PIM::
        Service);
      self.sendsTo.resolve(PIM::SoftwareAgent)>
        forEach(delegate) {
```

```

        if (delegate.oclIsTypeOf(PIM::Service))
            then {
                s.connectsTo += delegate.oclAsType(PIM
                    ::Service);
            }else{
                delegate.oclAsType(PIM::Application).
                    requestsServices += s;
            }endif;
        }
    }else {
        var app:PIM::Application := agent.oclAsType(
            PIM::Application);
        self.sendsTo.resolve(PIM::SoftwareAgent)>
            forEach(delegate) {
                if (delegate.oclIsTypeOf(PIM::Service))
                    then {
                        app.requestsServices += delegate.
                            oclAsType(PIM::Service);
                    }else{
                    }endif;
            }
        }endif;
    };
}

mapping Channel::toNetworkingTechnology() : PIM::
    NetworkingTechnology {
    name := self.name;
}

mapping Protocol::toSoftwareProtocol() : PIM::
    SoftwareProtocol {
    name := self.name;
    composedBy := self.composedBy>map toSoftwareProtocol
        ();
    rules := self.rules.late resolve(PIM::
        NetworkingTechnology);
}

mapping Participant::toAgent() : PIM::SoftwareAgent when
    {self.type <> CIM::ParticipantRole::PEER} {

```

```
init {
  if (self.type = CIM::ParticipantRole::PASSIVE)
    then {
      result := object Service{
        name := self.name+'Service';
      };
    }else {
      result := object Application{
        name := self.name+'App';
      };
    }endif;
}

exchanges := self.exchanges>map toSoftwareMessage()>
  flatten();

result>map populateSoftwareAgent(self);
}

mapping Participant::toPeerAgents() : app:PIM::
  Application, service:PIM::Service when {self.type =
  CIM::ParticipantRole::PEER} {
  app.name := self.name+'App';
  service.name := self.name+'Service';

  app.exchanges := self.exchanges>map toSoftwareMessage
    ()>flatten();
  service.exchanges := self.exchanges>map
    toSoftwareMessage()>flatten();

  app>map populateSoftwareAgent(self);
  service>map populateSoftwareAgent(self);
}

mapping Message::toSoftwareMessage() : Sequence(
  SoftwareMessage) {
  init {
    if (not self.oclIsTypeOf(CIM::Initial) and not
      self.oclIsTypeOf(CIM::End)) then {
      var eventTopic := object PIM::Topic{
        name := self.name+'Topic';
```

```

};
var event := object PIM::Event{
    name := self.name+'Event';
    topic := eventTopic;
    medium := self.medium.late resolve(PIM::
        NetworkingTechnology);
    conforms := self.conforms.late resolve(PIM
        ::SoftwareProtocol);
};
var predicate := object PIM::Predicate{
    name := self.name+'Predicate';
    accepts := event;
};
var request := object PIM::Request{
    name := self.name+'Request';
    medium := self.medium.late resolve(PIM::
        NetworkingTechnology);
    conforms := self.conforms.late resolve(PIM
        ::SoftwareProtocol);
};
var response := object PIM::Response{
    name := self.name+'Response';
    medium := self.medium.late resolve(PIM::
        NetworkingTechnology);
    conforms := self.conforms.late resolve(PIM
        ::SoftwareProtocol);
};

request.reply := response;
response.petition := request;

result += event;
result += request;
result += response;
}endif;
}
}

mapping inout PIM::SoftwareAgent::populateSoftwareAgent(p
:CIM::Participant){
    var discoveryEvent := object Discovery{

```



```
    name := self.name+'DiscoveryEvent';
    medium := p.exchanges.medium.late resolve(PIM::
        NetworkingTechnology);
    conforms := p.exchanges.conforms.late resolve(PIM
        ::SoftwareProtocol);
    topic := object Topic{
        name := self.name+'DiscoveryTopic';
    };
};

self.discoverer := object Discoverer{
    name := self.name+'Discoverer';
    callback := object DiscoveryListener {
        name := self.name+'DiscoveryListener';
        discoveredBy := discoveryEvent;
        constrainedBy := object Predicate{
            name := self.name+'DiscoveryPredicate';
        };
    };
    listeners += callback;
};

self.eventhandler := object EventHandler {
    name := self.name+'EventHandler';
};

self.exchanges>select(evt | evt.oclIsTypeOf(PIM::
    Event)).oclAsType(PIM::Event)>forEach(evt){
    self.eventhandler.listeners += object
        EventListener{
            name := self.name+evt.name+'Listener';
            listens += evt;
            constrainedBy := evt.acceptedBy;
        };
    self.eventhandler.handles += evt;
};
}

mapping Choreography::toSoftwareAgentChoreography() : PIM
::SoftwareAgentChoreography {
    name := self.name;
```

```

system := self.system.late resolveone(PIM::
    UbiquitousSystem);
participants := self.participants.resolve(PIM::
    SoftwareAgent);
messages := self.messages.resolve(Sequence(PIM::
    SoftwareMessage))>flatten();
activities := self.activities>map
    toChoreographyActivity()>flatten();
activities>forEach(act){
    messages += act.messages;
};

startingEvent := self.startingEvent>map
    toStartingEvent(self)>asSequence()>at(1);

endingEvents := self.endingEvents>map toEndingEvents
    ();
activities += self.links>map toActivities(result)>
    flatten();
}

mapping CIM::Initial::toStartingEvent(choreography: CIM::
    Choreography) : PIM::Initial{
    init{
        result := object PIM::Initial{
            name := choreography.startingEvent.name;
            initiatedChoreographies := choreography.late
                resolve(PIM::SoftwareAgentChoreography);
            topic := object PIM::Topic{
                name := choreography.startingEvent.name+'
                    Topic';
            };
            conforms := choreography.startingEvent.
                conforms.late resolve(PIM::SoftwareProtocol
                    );
        };
    }
}

mapping CIM::End::toEndingEvents() : PIM::End{
    name := self.name;

```

```
topic := object Topic{
  name := self.name+'Topic';
};
endedChoreographies := self.endedChoreographies.late
  resolve(PIM::SoftwareAgentChoreography);
conforms := self.conforms.late resolve(PIM::
  SoftwareProtocol);
}

mapping CIM::ChoreographyActivity::toChoreographyActivity
() : Sequence(PIM::ChoreographyActivity) {
  init{
    var counter := 0;
    var msgs := self.messages.resolve(Sequence(PIM::
      SoftwareMessage))>flatten();

    self.sourceParticipant.resolve(PIM::SoftwareAgent)
      >forEach(sAgent){
        self.targetParticipant.resolve(PIM::
          SoftwareAgent)>forEach(tAgent){
          var activity := object PIM::
            ChoreographyActivity{
              name := self.name+counter.toString();

              participants += sAgent;
              participants += tAgent;

              counter := counter+1;
            };

          activity.transactions += object
            DiscoveryActivity{
              name := sAgent.name+'2'+tAgent.name+'
                Discovery';
              sourceParticipant := sAgent;
              targetParticipant := tAgent;
            };
          activity.transactions += self.
            subactivities>map
              toElementalCommunicationActivities(
                activity, msgs)>flatten();
        }
      }
    }
  }
}
```

```

        activity.messages += activity.transactions
            >select(t|t.ocIsTypeOf(PIM::
                EventDistributionActivity)).oclAsType(
                PIM::EventDistributionActivity).
                relatedEvent;

        result += activity;
    };
};
}
}

```

```

mapping populateTransactions(src:PIM::SoftwareAgent, tgt:
PIM::SoftwareAgent, sMessages:Sequence(PIM::
SoftwareMessage)) : Sequence(PIM::
ElementalCommunicationActivity) {
init{
    sMessages>forEach(m){
        if (m.ocIsTypeOf(PIM::Event)) then {
            result += object PIM::
                EventDistributionActivity{
                    name := src.name+'2'+tgt.name+'
                        EventDistribution';
                    relatedEvent := m.ocIsType(PIM::Event)
                        ;
                    sourceParticipant := src;
                };
        }else{
            if (m.ocIsTypeOf(PIM::Request)) then {
                var req := m.ocIsType(PIM::Request);
                switch{
                    case (src.ocIsTypeOf(PIM::
                        Application) and tgt.ocIsTypeOf
                        (PIM::Service)){
                        result += object PIM::
                            MessageExchangingActivity{
                                name := src.name+'2'+tgt.
                                    name+'
                                    RequestMessageExchanging'
                                    ;
                                message := req;
                            };
                    };
                };
            };
        };
    };
};
}
}

```

```
        req.targets += tgt;
        sourceParticipant := src;
        targetParticipant := tgt;
    };
}
case (src.oclIsTypeOf(PIM::Service)
and tgt.oclIsTypeOf(PIM::
Service)){
result += object PIM::
    MessageExchangingActivity{
        name := src.name+'2'+tgt.
            name+'
            RequestMessageExchanging'
        ;
        message := req;
        req.targets += tgt;
        sourceParticipant := src;
        targetParticipant := tgt;
    };
}
};
}
else{
var resp := m.oclAsType(PIM::Response);
switch{
case (src.oclIsTypeOf(PIM::Service)
and tgt.oclIsTypeOf(PIM::
Application)){
result += object PIM::
    MessageExchangingActivity{
        name := src.name+'2'+tgt.
            name+'
            ResponseMessageExchanging
            ';
        message := resp;
        resp.exchangedFrom += tgt;
        sourceParticipant := src;
        targetParticipant := tgt;
    };
}
}
```



```
    }  
}
```

```
mapping CIM::Link::toActivities(inout choreography:PIM::  
  SoftwareAgentChoreography) : Sequence(PIM::  
  ChoreographyActivity){  
  init{  
    var src:CIM::FlowObject := self.source;  
    var tgt:CIM::FlowObject := self.target;  
  
    var activity1:Bag(Sequence(PIM::  
      ChoreographyActivity));  
    var activity2:Bag(Sequence(PIM::  
      ChoreographyActivity));  
  
    if (src.oclIsKindOf(CIM::Event)) then{  
      activity1 := src.oclAsType(CIM::Event)>map  
        toEventDistributionActivity();  
      result += activity1>flatten();  
    }else {  
      activity1 := src.oclAsType(CIM::  
        ChoreographyActivity).resolve(Sequence(PIM  
          ::ChoreographyActivity))>asBag();  
    }endif;  
  
    if (tgt.oclIsKindOf(CIM::Event)) then{  
      activity2 := tgt.oclAsType(CIM::Event)>map  
        toEventDistributionActivity();  
      result += activity2>flatten();  
    }else {  
      activity2 := tgt.oclAsType(CIM::  
        ChoreographyActivity).resolve(Sequence(PIM  
          ::ChoreographyActivity))>asBag();  
    }endif;  
  
    activity1>flatten()>forEach(act1){  
      src.starter>map toGateway();  
      activity2>flatten()>forEach(act2){  
        var startingMessage : PIM::Event;  
        switch{
```

```

    case (self.oclIsTypeOf(CIM::Conditional
    )){
        startingMessage := object PIM::
            Conditional{
                name := act2.name+'
                    ConditionalStartingEvent';
                predicate := object PIM::
                    Predicate{
                        name := act2.name+'
                            ConditionalStartingEventPredicate
                                '
                    };
            };
    }
    case (self.oclIsTypeOf(CIM::Default)){
        startingMessage := object PIM::
            Default{
                name := act2.name+'
                    DefaultStartingEvent';
                nonSatisfiedPredicate := object
                    PIM::Predicate{
                        name := act2.name+'
                            DefaultStartingEventPredicate
                                '
                    };
            };
    }
    else{
        startingMessage := object PIM::
            Sequential{
                name := act2.name+'StartingEvent
                    '
            };
    }
};

act1.messages += startingMessage;
act1.transactions += object PIM::
    EventDistributionActivity{
        name := startingMessage.name+'
            DistributionActivity';
    }

```



```
        relatedEvent := startingMessage;
        sourceParticipant := act1.participants>
            at(1);
    };
    tgt.starter>map toGateway();
    act2.startingMessages += startingMessage;
};
};

choreography.messages += activity1>flatten().
    startingMessages;
choreography.messages += activity2>flatten().
    startingMessages;
}
}

mapping CIM::Event::toEventDistributionActivity() :
    Sequence(PIM::ChoreographyActivity){
    init{
        result += object PIM::ChoreographyActivity{
            name := self.name+'DistributionActivity';
        }
    }

    result>at(1).transactions += object PIM::
        EventDistributionActivity{
            name := self.name+'DistributionTransactionActivity
                ';
            sourceParticipant := self.exchangedFrom.resolve(
                PIM::SoftwareAgent)>at(1);
            relatedEvent := self.oclAsType(CIM::Event).resolve
                (PIM::Event)>at(1);
        };
    }

mapping CIM::Gateway::toGateway() : PIM::Gateway{
    init{
        switch{
            case (self.oclIsTypeOf(CIM::Complex)){
                result := object PIM::Complex{
                    name := self.name;
                }
            }
        }
    }
}
```

```
};
}
case (self.oclIsTypeOf(CIM::Parallel)){
    result := object PIM::Parallel{
        name := self.name;
    };
}
case (self.oclIsTypeOf(CIM::DataBased)){
    result := object PIM::DataBased{
        name := self.name;
        condition := object PIM::Predicate{
            name := self.name+'Predicate';
        };
    };
}
case (self.oclIsTypeOf(CIM::EventBased)){
    result := object PIM::EventBased{
        name := self.name;
        triggeringEvent := self.oclAsType(CIM::
            EventBased).triggeringEvent.late
            resolveone(PIM::Event);
    };
}
case (self.oclIsTypeOf(CIM::Inclusive)){
    result := object PIM::Inclusive{
        name := self.name;
    };
}
case (self.oclIsTypeOf(CIM::Chaining)){
    result := object PIM::Chaining{
        name := self.name;
        src := self.oclAsType(CIM::Chaining).
            source.late resolveone(PIM::Gateway
            );
        tgt := self.oclAsType(CIM::Chaining).
            target.late resolveone(PIM::Gateway
            );
    };
}
};
}
```

```
}
```

VII. ATL transformation rules that can be applied to the behavioral view of a CS-CIM to produce a UML sequence diagram

```
module CIM2SequenceDiagram;

@path CIM=/MUSYC_QVT/metamodels/CIM.ecore
@path UML=/MUSYC_QVT/metamodels/UML.ecore

create OUT: UML from IN: CIM;

helper def: getInteractionOperator(g : CIM!Gateway) : UML
!InteractionOperatorKind=
if(g>oclIsTypeOf(CIM!Parallel)) then
  #par
else
  if(g>oclIsTypeOf(CIM!Exclusive)) then
    #opt
  else
    if(g>oclIsTypeOf(CIM!Chaining)) then
      #seq
    else
      if(g>oclIsTypeOf(CIM!Inclusive)) then
        #alt
      else
        if(g>oclIsTypeOf(CIM!EventBased)) then
          #seq
        else
          #seq
        endif
      endif
    endif
  endif
endif;
endif;
```

```

rule InitialEvent2TriggerChoreography{
  from
    event : CIM!Initial(event.isSourceOf.target>
      oclIsKindOf(CIM!ChoreographyActivity))
  using{
    connection : CIM!Link = event.isSourceOf;
    activity : CIM!ChoreographyActivity = connection.
      target>first();
  }
  to
    combinedFragment : UML!CombinedFragment(
      name < 'InitEvent',
      interactionOperator < #seq,
      operand < OrderedSet{oper1}
    ),
    =====
    oper1 : UML!ExecutionOccurrenceSpecification(
      name < 'InitChoreographyExecution',
      execution < oper1execution
    ),
    oper1execution : UML!ActionExecutionSpecification(
      name < 'InitChoreographyExecutionSpecification
        ',
      start < oper1executionStart,
      finish < oper1executionEnd
    ),
    =====
    oper1executionStart : UML!OccurrenceSpecification(
      name < '
        InitChoreographyExecutionSpecificationStart
        ',
      event < oper1executionStartEvent
    ),
    oper1executionEnd : UML!OccurrenceSpecification(
      name < '
        InitChoreographyExecutionSpecificationEnd',
      event < oper1executionEndEvent
    ),
    =====
    oper1executionStartEvent : UML!SendOperationEvent(

```

```
        name < '
            InitChoreographyExecutionSpecificationStartEvent
        ',
        operation < oper1executionEventOperation
    ),
    oper1executionEndEvent : UML!ReceiveOperationEvent
    (
        name < '
            InitChoreographyExecutionSpecificationEndEvent
        ',
        operation < oper1executionEventOperation
    ),
    =====
    oper1executionEventOperation : UML!Operation(
        name < 'initActivity',
        class < oper1executionEventOperationClass
    ),
    oper1executionEventOperationClass : UML!Class(
        name < activity.participants>first().name + '
            Service'
    )
}

rule InitialEvent2TriggerGateway{
    from
        event : CIM!Initial(event.isSourceOf.target>
            oclIsKindOf(CIM!Gateway))
    using{
        connection : CIM!Link = event.isSourceOf;
        gateway : CIM!Gateway = connection.target>first();
    }
    to
        combinedFragment : UML!CombinedFragment(
            name < 'InitEvent',
            interactionOperator < #seq,
            operand < OrderedSet{thisModule>resolveTemp(
                gateway, 'combinedFragment')})
        )
}
```

```

rule ChoreographyActivity2CombinedFragment{
  from
    activity : CIM!ChoreographyActivity
  using{
    participants : Sequence(CIM!Participant) =
      activity.participants;
    messages : Sequence(CIM!Message) = activity.
      messages;
  }
  to
    combinedFragment : UML!CombinedFragment(
      name < activity.name + 'Loop',
      interactionOperator < #loop,
      operand < OrderedSet{oper1, oper2}
    ),
    =====
    oper1 : UML!ExecutionOccurrenceSpecification(
      name < activity.name + 'DiscoverExecution',
      execution < oper1execution
    ),
    oper1execution : UML!ActionExecutionSpecification(
      name < activity.name + '
        DiscoverExecutionSpecification',
      start < oper1executionStart,
      finish < oper1executionEnd
    ),
    =====
    oper1executionStart : UML!OccurrenceSpecification(
      name < activity.name + '
        DiscoverExecutionSpecificationStart',
      event < oper1executionStartEvent
    ),
    oper1executionEnd : UML!OccurrenceSpecification(
      name < activity.name + '
        DiscoverExecutionSpecificationEnd',
      event < oper1executionEndEvent
    ),
    =====
    oper1executionStartEvent : UML!SendOperationEvent(
      name < activity.name + '
        DiscoverExecutionSpecificationStartEvent',

```

```
        operation < oper1executionEventOperation
    ),
    oper1executionEndEvent : UML!ReceiveOperationEvent
    (
        name < activity.name + '
            DiscoverExecutionSpecificationEndEvent',
        operation < oper1executionEventOperation
    ),
    =====
    oper1executionEventOperation : UML!Operation(
        name < 'v = discover()',
        class < oper1executionEventOperationClass,
        datatype < returnDatatype
    ),
    oper1executionEventOperationClass : UML!Class(
        name < participants>first().name + 'Discoverer
        ,
    ),
    returnDatatype : UML!DataType(
        name < 'Vector<Service>'
    ),
    =====

    oper2 : UML!CombinedFragment(
        name < activity.name + 'Break',
        interactionOperator < #break,
        operand < OrderedSet{guardOper, oper3, oper4,
            oper5}
    ),
    guardOper : UML!InteractionOperand(
        name < activity.name + 'GuardBreak',
        guard < guard
    ),
    guard : UML!InteractionConstraint(
        specification < valueSpec
    ),
    valueSpec : UML!OpaqueExpression(
        body < '[v.contains(' + participants>last().
            name + 'Service)]'
    ),
    =====
```

```

oper3 : UML!ExecutionOccurrenceSpecification(
    name < messages>first().name + 'Request',
    execution < oper3execution
),
oper3execution : UML!ActionExecutionSpecification(
    name < messages>first().name + '
        RequestExecutionSpecification',
    start < oper3executionStart,
    finish < oper3executionEnd
),
=====
oper3executionStart : UML!OccurrenceSpecification(
    name < messages>first().name + '
        RequestExecutionSpecificationStart',
    event < oper3executionStartEvent
),
oper3executionEnd : UML!OccurrenceSpecification(
    name < messages>first().name + '
        RequestExecutionSpecificationEnd',
    event < oper3executionEndEvent
),
=====
oper3executionStartEvent : UML!SendOperationEvent(
    name < messages>first().name + '
        RequestExecutionSpecificationStartEvent',
    operation < oper3executionEventOperation
),
oper3executionEndEvent : UML!ReceiveOperationEvent
(
    name < messages>first().name + '
        RequestExecutionSpecificationEndEvent',
    operation < oper3executionEventOperation
),
=====
oper3executionEventOperation : UML!Operation(
    name < messages>first().name,
    class < oper3executionEventOperationClass
),
oper3executionEventOperationClass : UML!Class(
    name < participants>last().name + 'Service'
),

```



```
=====  
oper4 : UML!ExecutionOccurrenceSpecification(  
    name < messages>first().name + 'Reply',  
    execution < oper4execution  
)  
,  
oper4execution : UML!ActionExecutionSpecification(  
    name < messages>first().name + '  
        ReplyExecutionSpecification',  
    start < oper4executionStart,  
    finish < oper4executionEnd  
)  
,  
=====  
oper4executionStart : UML!OccurrenceSpecification(  
    name < messages>first().name + '  
        ReplyExecutionSpecificationStart',  
    event < oper4executionStartEvent  
)  
,  
oper4executionEnd : UML!OccurrenceSpecification(  
    name < messages>first().name + '  
        ReplyExecutionSpecificationEnd',  
    event < oper4executionEndEvent  
)  
,  
=====  
oper4executionStartEvent : UML!SendOperationEvent(  
    name < messages>first().name + '  
        ReplyExecutionSpecificationStartEvent',  
    operation < oper4executionEventOperation  
)  
,  
oper4executionEndEvent : UML!ReceiveOperationEvent  
(  
    name < messages>first().name + '  
        ReplyExecutionSpecificationEndEvent',  
    operation < oper4executionEventOperation  
)  
,  
=====  
oper4executionEventOperation : UML!Operation(  
    name < messages>last().name,  
    class < oper4executionEventOperationClass  
)  
,  
oper4executionEventOperationClass : UML!Class(  

```

```

        name < participants>first().name + 'Service'
    ),

    =====
oper5 : UML!ExecutionOccurrenceSpecification(
    name < 'InitiateActivity',
    execution < oper5execution
),
oper5execution : UML!ActionExecutionSpecification(
    name < 'InitiateActivityExecutionSpecification
        ',
    start < oper5executionStart,
    finish < oper5executionEnd
),
    =====
oper5executionStart : UML!OccurrenceSpecification(
    name < '
        InitiateActivityExecutionSpecificationStart
        ',
    event < oper5executionStartEvent
),
oper5executionEnd : UML!OccurrenceSpecification(
    name < '
        InitiateActivityExecutionSpecificationEnd',
    event < oper5executionEndEvent
),
    =====
oper5executionStartEvent : UML!SendOperationEvent(
    name < '
        InitiateActivityExecutionSpecificationStartEvent
        ',
    operation < oper5executionEventOperation
),
oper5executionEndEvent : UML!ReceiveOperationEvent
(
    name < '
        InitiateActivityExecutionSpecificationEndEvent
        ',
    operation < oper5executionEventOperation
),
    =====

```

```
oper5executionEventOperation : UML!Operation(  
    name < 'initActivity',  
    class < oper5executionEventOperationClass  
) ,  
oper5executionEventOperationClass : UML!Class(  
    name < participants>last().name + 'Service'  
) ,  
  
operationMessage1 : UML!Operation(  
    name < messages>first().name  
) ,  
operationMessage2 : UML!Operation(  
    name < messages>last().name  
)  
do{  
    thisModule>resolveTemp(participants>first(), '  
        service').ownedOperation < Sequence{thisModule  
>resolveTemp(participants>first(), 'service').  
ownedOperation, operationMessage2}>flatten()  
>asSet();  
    thisModule>resolveTemp(participants>last(), '  
        service').ownedOperation < Sequence{thisModule  
>resolveTemp(participants>last(), 'service').  
ownedOperation, operationMessage1}>flatten()  
>asSet();  
}  
}  
  
rule Participant2ServiceLifeline{  
    from  
        participant : CIM!Participant  
    to  
        service : UML!Service(  
            name < participant.name + 'Service',  
            lifeline < serviceLifeline,  
            ownedOperation < Sequence{initOperation,  
                endOperation}  
        ) ,  
        serviceLifeline : UML!Lifeline(  
            name < ':' + participant.name + 'Service'  
        ) ,  
}
```

```

discoverer : UML!Service(
    name < participant.name + 'Discoverer',
    lifeline < discovererLifeline,
    ownedOperation < Sequence{discoverOperation}
),
discovererLifeline : UML!Lifeline(
    name < ':' + participant.name + 'Discoverer'
),
initOperation : UML!Operation(
    name < 'initActivity'
),
endOperation : UML!Operation(
    name < 'endActivity'
),
discoverOperation : UML!Operation(
    name < 'discover',
    datatype < returnDatatype
),
returnDatatype : UML!DataType(
    name < 'Vector<Service>'
)
}

rule ParticipantIntermediate2CombinedFragment{
    from
        participant : CIM!Participant,
        event : CIM!Intermediate
    to
        =====
        oper : UML!ExecutionOccurrenceSpecification(
            name < 'IntermediateEventExecution',
            execution < oper1execution
        ),
        oper1execution : UML!ActionExecutionSpecification(
            name < '
                IntermediateEventExecutionSpecification',
            start < oper1executionStart,
            finish < oper1executionEnd
        ),
        =====
        oper1executionStart : UML!OccurrenceSpecification(

```

```
        name < '
            IntermediateEventExecutionSpecificationStart
        ',
        event < oper1executionStartEvent
    ),
    oper1executionEnd : UML!OccurrenceSpecification(
        name < '
            IntermediateEventExecutionSpecificationEnd'
        ,
        event < oper1executionEndEvent
    ),
    =====
    oper1executionStartEvent : UML!SendOperationEvent(
        name < '
            IntermediateEventExecutionSpecificationStartEvent
        ',
        operation < oper1executionEventOperation
    ),
    oper1executionEndEvent : UML!ReceiveOperationEvent
    (
        name < '
            IntermediateEventExecutionSpecificationEndEvent
        ',
        operation < oper1executionEventOperation
    ),
    =====
    oper1executionEventOperation : UML!Operation(
        name < 'on' + event.name + 'Received',
        class < oper1executionEventOperationClass
    ),
    oper1executionEventOperationClass : UML!Class(
        name < participant.name + 'Service'
    )
do{
    thisModule>resolveTemp(participant, 'service').
        ownedOperation < Sequence{thisModule>
            resolveTemp(participant, 'service').
                ownedOperation, oper1executionEventOperation}>
            flatten()>asSet();
}
}
```

```
rule InitiatorMessage2Message{
  from
    srcMessage : CIM!Message(srcMessage.initiator =
      true)
  to
    tgtMessage : UML!Message(
      name < srcMessage.name,
      messageKind < #complete,
      messageSort < #synchCall
    )
}

rule NonInitiatorMessage2Message{
  from
    srcMessage : CIM!Message(srcMessage.initiator =
      false)
  to
    tgtMessage : UML!Message(
      name < srcMessage.name,
      messageKind < #complete,
      messageSort < #reply
    )
}

rule DefinedConnection2GeneralOrdering{
  from
    connection : CIM!Link(not connection.source>
      oclIsUndefined() and not connection.target>
      first().oclIsUndefined())
  using{
    source : CIM!FlowObject = connection.source;
    target : CIM!FlowObject = connection.target>first
      ();
  }
  to
    ordering : UML!GeneralOrdering(
      before < occurrenceSpecSource,
      after < occurrenceSpecTarget
    ),
}
```

```
        occurrenceSpecSource : UML!OccurrenceSpecification
            (
                name < source.name
            ),
        occurrenceSpecTarget : UML!OccurrenceSpecification
            (
                name < target.name
            )
    }
}
```

```
rule IntermediateEvent2CombinedFragment{
    from
        event : CIM!Intermediate
    using{
        participants : Sequence(CIM!Participant) = CIM!
            Participant>allInstances()>asSet()>asSequence
            ();
    }
    to
        combinedFragment : UML!CombinedFragment(
            name < event.name + 'IntermediateEvent',
            interactionOperator < #seq,
            operand < participants>collect(p | thisModule.
                IntermediateEventParticipant2Execution(p,
                    event))
        )
    }
}
```

```
lazy rule IntermediateEventParticipant2Execution{
    from
        participant : CIM!Participant,
        event : CIM!Intermediate
    to
        =====
        oper : UML!ExecutionOccurrenceSpecification(
            name < event.name + 'Request',
            execution < operexecution
        ),
        operexecution : UML!ActionExecutionSpecification(
            name < event.name + '
                RequestExecutionSpecification',
```

```

        start < operexecutionStart,
        finish < operexecutionEnd
    ),
    =====
operexecutionStart : UML!OccurrenceSpecification(
    name < event.name + '
        RequestExecutionSpecificationStart',
    event < operexecutionStartEvent
),
operexecutionEnd : UML!OccurrenceSpecification(
    name < event.name + '
        RequestExecutionSpecificationEnd',
    event < operexecutionEndEvent
),
    =====
operexecutionStartEvent : UML!SendOperationEvent(
    name < event.name + '
        RequestExecutionSpecificationStartEvent',
    operation < operexecutionEventOperation
),
operexecutionEndEvent : UML!ReceiveOperationEvent(
    name < event.name + '
        RequestExecutionSpecificationEndEvent',
    operation < operexecutionEventOperation
),
    =====
operexecutionEventOperation : UML!Operation(
    name < 'on' + event.name + 'Received',
    class < operexecutionEventOperationClass
),
operexecutionEventOperationClass : UML!Class(
    name < participant.name + 'Service'
)
}

```

```

rule EndEvent2TriggerChoreography{
    from
        event : CIM!End
    using{
        connection : CIM!Link = event.isTargetOf>first();
    }
}

```



```
        activity : CIM!ChoreographyActivity = connection.
            source;
    }
    to
    combinedFragment : UML!CombinedFragment(
        name < 'EndEvent',
        interactionOperator < #seq,
        operand < OrderedSet{oper1}
    ),
    =====
    oper1 : UML!ExecutionOccurrenceSpecification(
        name < 'EndChoreographyExecution',
        execution < oper1execution
    ),
    oper1execution : UML!ActionExecutionSpecification(
        name < 'EndChoreographyExecutionSpecification'
        ,
        start < oper1executionStart,
        finish < oper1executionEnd
    ),
    =====
    oper1executionStart : UML!OccurrenceSpecification(
        name < '
            EndChoreographyExecutionSpecificationStart'
        ,
        event < oper1executionStartEvent
    ),
    oper1executionEnd : UML!OccurrenceSpecification(
        name < '
            EndChoreographyExecutionSpecificationEnd',
        event < oper1executionEndEvent
    ),
    =====
    oper1executionStartEvent : UML!SendOperationEvent(
        name < '
            EndChoreographyExecutionSpecificationStartEvent
            ',
        operation < oper1executionEventOperation
    ),
    oper1executionEndEvent : UML!ReceiveOperationEvent
    (
```

```

        name < '
            EndChoreographyExecutionSpecificationEndEvent
        ',
        operation < oper1executionEventOperation
    ),
    =====
oper1executionEventOperation : UML!Operation(
    name < 'endActivity',
    class < oper1executionEventOperationClass
),
oper1executionEventOperationClass : UML!Class(
    name < activity.participants>last().name + '
        Service'
    )
}

```

```

rule Gateway2ChoreographyCombinedFragment{
    from
        gateway : CIM!Gateway(gateway.isSourceOf.target>
            first())>oclIsTypeOf(CIM!ChoreographyActivity)
            and gateway.isTargetOf>size() = 1)
    using{
        connection : CIM!Link = gateway.isSourceOf;
        flow : Sequence(CIM!FlowObject) = connection.
            target;
    }
    to
        combinedFragment : UML!CombinedFragment(
            name < gateway.name,
            interactionOperator < thisModule.
                getInteractionOperator(gateway),
            operand < OrderedSet{seq1, seq2}
        ),
        =====
        seq1 : UML!CombinedFragment(
            name < 'Seq1',
            interactionOperator < #seq,
            operand < Sequence{oper1, thisModule>
                resolveTemp(flow>first(), 'combinedFragment
                    ')}
        ),

```

```
seq2 : UML!CombinedFragment(  
    name < 'Seq2',  
    interactionOperator < #seq,  
    operand < Sequence{oper2, thisModule>  
        resolveTemp(flow>last(), 'combinedFragment'  
        )}  
),  
=====
```

```
oper1 : UML!ExecutionOccurrenceSpecification(  
    name < 'InitChoreographyExecutionSeq1',  
    execution < oper1execution  
),  
oper1execution : UML!ActionExecutionSpecification(  
    name < '  
        InitChoreographyExecutionSpecificationSeq1'  
    ,  
    start < oper1executionStart,  
    finish < oper1executionEnd  
),  
=====
```

```
oper1executionStart : UML!OccurrenceSpecification(  
    name < '  
        InitChoreographyExecutionSpecificationStartSeq1'  
    ,  
    event < oper1executionStartEvent  
),  
oper1executionEnd : UML!OccurrenceSpecification(  
    name < '  
        InitChoreographyExecutionSpecificationEndSeq1'  
    ,  
    event < oper1executionEndEvent  
),  
=====
```

```
oper1executionStartEvent : UML!SendOperationEvent(  
    name < '  
        InitChoreographyExecutionSpecificationStartEventSeq1'  
    ,  
    operation < oper1executionEventOperation  
),  
oper1executionEndEvent : UML!ReceiveOperationEvent  
(
```

```

        name < '
            InitChoreographyExecutionSpecificationEndEventSeq1
        ',
        operation < oper1executionEventOperation
    ),
    =====
oper1executionEventOperation : UML!Operation(
    name < 'initActivity',
    class < oper1executionEventOperationClass
),
oper1executionEventOperationClass : UML!Class(
    name < flow>first().participants>first().name
        + 'Service'
),

    =====
oper2 : UML!ExecutionOccurrenceSpecification(
    name < 'InitChoreographyExecutionSeq2',
    execution < oper2execution
),
oper2execution : UML!ActionExecutionSpecification(
    name < '
        InitChoreographyExecutionSpecificationSeq2'
    ,
    start < oper2executionStart,
    finish < oper2executionEnd
),
    =====
oper2executionStart : UML!OccurrenceSpecification(
    name < '
        InitChoreographyExecutionSpecificationStartSeq2
    ',
    event < oper2executionStartEvent
),
oper2executionEnd : UML!OccurrenceSpecification(
    name < '
        InitChoreographyExecutionSpecificationEndSeq2
    ',
    event < oper1executionEndEvent
),

```

```
=====
oper2executionStartEvent : UML!SendOperationEvent(
  name < '
    InitChoreographyExecutionSpecificationStartEventSeq2
  ',
  operation < oper2executionEventOperation
),
oper2executionEndEvent : UML!ReceiveOperationEvent
(
  name < '
    InitChoreographyExecutionSpecificationEndEventSeq2
  ',
  operation < oper2executionEventOperation
),
=====
oper2executionEventOperation : UML!Operation(
  name < 'initActivity',
  class < oper2executionEventOperationClass
),
oper2executionEventOperationClass : UML!Class(
  name < flow>last().participants>first().name +
  'Service'
)
}
```

```
rule Gateway2CombinedFragment{
  from
    gateway : CIM!Gateway(not gateway.isSourceOf.
      target>first()>oclIsTypeOf(CIM!
      ChoreographyActivity)
    and gateway.isTargetOf>size() = 1)
  using{
    connection : CIM!Link = gateway.isSourceOf;
    flow : Sequence(CIM!FlowObject) = connection.
      target;
  }
  to
    combinedFragment : UML!CombinedFragment(
      name < gateway.name,
      interactionOperator < thisModule.
        getInteractionOperator(gateway),
```

```

        operand < OrderedSet{seq1, seq2}
    ),
    =====
seq1 : UML!CombinedFragment(
    name < 'Seq1',
    interactionOperator < #seq,
    operand < Sequence{thisModule>resolveTemp(flow
        >first(), 'combinedFragment')}
    ),
seq2 : UML!CombinedFragment(
    name < 'Seq2',
    interactionOperator < #seq,
    operand < Sequence{thisModule>resolveTemp(flow
        >last(), 'combinedFragment')}}
    )
}

```

```

rule MergeGateway2ChoreographyCombinedFragment{
    from
        gateway : CIM!Gateway(gateway.isSourceOf.target>
            first())>oclIsTypeOf(CIM!ChoreographyActivity)
            and gateway.isTargetOf>size() > 1)
    using{
        connection : CIM!Link = gateway.isSourceOf;
        flow : Sequence(CIM!FlowObject) = connection.
            target;
    }
    to
        combinedFragment : UML!CombinedFragment(
            name < gateway.name,
            interactionOperator < thisModule.
                getInteractionOperator(gateway),
            operand < OrderedSet{seq1}
        ),
        =====
seq1 : UML!CombinedFragment(
    name < 'SeqMerge',
    interactionOperator < #seq,
    operand < Sequence{oper1, thisModule>
        resolveTemp(flow>first(), 'combinedFragment
        ')}
}

```

```
),
=====
oper1 : UML!ExecutionOccurrenceSpecification(
    name < 'InitChoreographyExecutionSeqMerge',
    execution < oper1execution
),
oper1execution : UML!ActionExecutionSpecification(
    name < '
        InitChoreographyExecutionSpecificationSeqMerge
    ',
    start < oper1executionStart,
    finish < oper1executionEnd
),
=====
oper1executionStart : UML!OccurrenceSpecification(
    name < '
        InitChoreographyExecutionSpecificationStartSeqMerge
    ',
    event < oper1executionStartEvent
),
oper1executionEnd : UML!OccurrenceSpecification(
    name < '
        InitChoreographyExecutionSpecificationEndSeqMerge
    ',
    event < oper1executionEndEvent
),
=====
oper1executionStartEvent : UML!SendOperationEvent(
    name < '
        InitChoreographyExecutionSpecificationStartEventSeqMerge
    ',
    operation < oper1executionEventOperation
),
oper1executionEndEvent : UML!ReceiveOperationEvent
(
    name < '
        InitChoreographyExecutionSpecificationEndEventSeqMerge
    ',
    operation < oper1executionEventOperation
),
=====
```

```

    oper1executionEventOperation : UML!Operation(
        name < 'initActivity',
        class < oper1executionEventOperationClass
    ),
    oper1executionEventOperationClass : UML!Class(
        name < flow>first().participants>first().name
        + 'Service'
    )
}

rule MergeGateway2CombinedFragment{
    from
        gateway : CIM!Gateway(not gateway.isSourceOf.
            target>first())>oclIsTypeOf(CIM!
            ChoreographyActivity)
            and gateway.isTargetOf>size() > 1)
    using{
        connection : CIM!Link = gateway.isSourceOf;
        flow : Sequence(CIM!FlowObject) = connection.
            target;
    }
    to
        combinedFragment : UML!CombinedFragment(
            name < gateway.name,
            interactionOperator < thisModule.
                getInteractionOperator(gateway),
            operand < OrderedSet{seq1}
        ),
        =====
        seq1 : UML!CombinedFragment(
            name < 'SeqMerge',
            interactionOperator < #seq,
            operand < Sequence{thisModule>resolveTemp(flow
                >first(), 'combinedFragment')}
        )
}

```


VIII. Specification of a CS-CIM for BlueRose middleware

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/
  XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xmlns:cim="http://www.ugr.es/~
  carlosrodriguez/CIM" xsi:schemaLocation="http://www.
  ugr.es/~carlosrodriguez/CIM ../metamodels/CIM.ecore">
<cim:CommunicationSystem name="BlueRose">
  <parties name="Proxy" type="PEER" sendsTo="Servant
    Proxy" receivesFrom="Servant Proxy" exchanges="
    Information InformationPetition"/>
  <parties name="Servant" type="PASSIVE" sendsTo="Proxy
    Servant" receivesFrom="Proxy Servant" exchanges="
    Information InformationPetition"/>
  <channels name="BRBroker" transports="Information
    InformationPetition" conforms="BRProtocol"/>
  <protocols name="BRProtocol" rules="BRBroker"
    compiles="Information InformationPetition"/>
  <choreographies participants="Proxy Servant" messages
    ="Information InformationPetition" links="
    CloseConnectionGateway2EndParallelGateway
    EndProxyRequestsGateway2ParallelEndGateway
    ServantAdditionalRequestGateway2ServantServantRequest
    CloseConnectionGateway2ProxyServantRequest
    EndProxyRequestsGateway2ProxyProxyRequest
    ServantAdditionalRequestGateway2ServantProxyProvision
    EndParallelGateway2EndEvent Init2ParallelGateway
    Parallel2ProxyProxyRequest
    Parallel2ProxyServantRequest
    ProxyProxyRequest2EndProxyRequestsGateway
    ProxyServantRequest2ServantAdditionalRequestGateway
    ServantProxyProvision2CloseConnectionGateway
    ServantServantRequest2ServantAdditionalRequestGateway
    " name="MainChoreography" startingEvent="
    InitialEvent" endingEvents="EndEvent">
  <activities isSourceOf="
    ProxyServantRequest2ServantAdditionalRequestGateway
    " isTargetOf="Parallel2ProxyServantRequest
```

```

        CloseConnectionGateway2ProxyServantRequest" name
        ="Proxy2ServantRequest" subactivities="
        ProxyServantRequest" messages="
        InformationPetition" sourceParticipant="Proxy"
        targetParticipant="Servant"/>
    <activities isSourceOf="
        ServantServantRequest2ServantAdditionalRequestGateway
        " isTargetOf="
        ServantAdditionalRequestGateway2ServantServantRequest
        " name="Servant2ServantRequest" subactivities="
        ServantServantRequest" messages="Information
        InformationPetition" sourceParticipant="Servant"
        targetParticipant="Servant"/>
    <activities isSourceOf="
        ProxyProxyRequest2EndProxyRequestsGateway"
        isTargetOf="Parallel2ProxyProxyRequest
        EndProxyRequestsGateway2ProxyProxyRequest" name="
        "Proxy2ProxyRequest" subactivities="
        ProxyProxyRequest" messages="Information
        InformationPetition" sourceParticipant="Proxy"
        targetParticipant="Proxy"/>
    <activities isSourceOf="
        ServantProxyProvision2CloseConnectionGateway"
        isTargetOf="
        ServantAdditionalRequestGateway2ServantProxyProvision
        " name="Servant2ProxyProvision" subactivities="
        ServantProxyRequest" messages="Information"
        sourceParticipant="Servant" targetParticipant="
        Proxy"/>
</choreographies>
<messages name="InformationPetition" exchangedFrom="
    Proxy Servant" medium="BRBroker" conforms="
    BRProtocol"/>
<messages name="Information" exchangedFrom="Proxy
    Servant" medium="BRBroker" conforms="BRProtocol"/>
</cim:CommunicationSystem>
<cim:Parallel isSourceOf="Parallel2ProxyServantRequest
    Parallel2ProxyProxyRequest" isTargetOf="
    Init2ParallelGateway" name="ParallelStartGateway"/>
<cim:Parallel isSourceOf="EndParallelGateway2EndEvent"
    isTargetOf="

```

```
EndProxyRequestsGateway2ParallelEndGateway
CloseConnectionGateway2EndParallelGateway" name="
ParallelEndGateway"/>
<cim:DataBased isSourceOf="
ServantAdditionalRequestGateway2ServantServantRequest

ServantAdditionalRequestGateway2ServantProxyProvision
" isTargetOf="
ProxyServantRequest2ServantAdditionalRequestGateway
ServantServantRequest2ServantAdditionalRequestGateway
" name="ServantAdditionalRequestGateway"/>
<cim:DataBased isSourceOf="
CloseConnectionGateway2EndParallelGateway
CloseConnectionGateway2ProxyServantRequest"
isTargetOf="
ServantProxyProvision2CloseConnectionGateway" name="
CloseConnectionGateway"/>
<cim:DataBased isSourceOf="
EndProxyRequestsGateway2ParallelEndGateway
EndProxyRequestsGateway2ProxyProxyRequest"
isTargetOf="
ProxyProxyRequest2EndProxyRequestsGateway" name="
EndProxyRequestsGateway"/>
<cim:Sequential name="Init2ParallelGateway" source="
InitialEvent" target="ParallelStartGateway"
choreography="MainChoreography"/>
<cim:Sequential name="Parallel2ProxyServantRequest"
source="ParallelStartGateway" target="
Proxy2ServantRequest" choreography="MainChoreography
"/>
<cim:Sequential name="Parallel2ProxyProxyRequest"
source="ParallelStartGateway" target="
Proxy2ProxyRequest" choreography="MainChoreography"
/>
<cim:Sequential name="
ProxyProxyRequest2EndProxyRequestsGateway" source="
Proxy2ProxyRequest" target="EndProxyRequestsGateway"
choreography="MainChoreography"/>
<cim:Sequential name="
ProxyServantRequest2ServantAdditionalRequestGateway"
source="Proxy2ServantRequest" target="
```

```

    ServantAdditionalRequestGateway" choreography="
    MainChoreography"/>
<cim:Conditional name="
    ServantAdditionalRequestGateway2ServantServantRequest
    " source="ServantAdditionalRequestGateway" target="
    Servant2ServantRequest" choreography="
    MainChoreography"/>
<cim:Conditional name="
    EndProxyRequestsGateway2ParallelEndGateway" source="
    EndProxyRequestsGateway" target="ParallelEndGateway"
    choreography="MainChoreography"/>
<cim:Default name="
    ServantAdditionalRequestGateway2ServantProxyProvision
    " source="ServantAdditionalRequestGateway" target="
    Servant2ProxyProvision" choreography="
    MainChoreography"/>
<cim:Default name="
    EndProxyRequestsGateway2ProxyProxyRequest" source="
    EndProxyRequestsGateway" target="Proxy2ProxyRequest"
    choreography="MainChoreography"/>
<cim:Sequential name="
    ServantServantRequest2ServantAdditionalRequestGateway
    " source="Servant2ServantRequest" target="
    ServantAdditionalRequestGateway" choreography="
    MainChoreography"/>
<cim:Sequential name="
    ServantProxyProvision2CloseConnectionGateway" source
    ="Servant2ProxyProvision" target="
    CloseConnectionGateway" choreography="
    MainChoreography"/>
<cim:Conditional name="
    CloseConnectionGateway2EndParallelGateway" source="
    CloseConnectionGateway" target="ParallelEndGateway"
    choreography="MainChoreography"/>
<cim:Default name="
    CloseConnectionGateway2ProxyServantRequest" source="
    CloseConnectionGateway" target="Proxy2ServantRequest
    " choreography="MainChoreography"/>
<cim:Sequential name="EndParallelGateway2EndEvent"
    source="ParallelEndGateway" target="EndEvent"
    choreography="MainChoreography"/>

```

```
<cim:Initial isSourceOf="Init2ParallelGateway" name="
  InitialEvent" initiatedChoreographies="
  MainChoreography"/>
<cim:End isTargetOf="EndParallelGateway2EndEvent" name=
  "EndEvent" endedChoreographies="MainChoreography"/>
<cim:SubChoreographyActivity name="ProxyServantRequest"
  messages="InformationPetition" sourceParticipant="
  Proxy" targetParticipant="Servant"
  sourceParticipantMessage="InformationPetition"
  activity="Proxy2ServantRequest"/>
<cim:SubChoreographyActivity name="ProxyProxyRequest"
  messages="Information InformationPetition"
  sourceParticipant="Proxy" targetParticipant="Proxy"
  sourceParticipantMessage="InformationPetition"
  targetParticipantMessage="Information" activity="
  Proxy2ProxyRequest"/>
<cim:SubChoreographyActivity name="ServantProxyRequest"
  messages="Information" sourceParticipant="Servant"
  targetParticipant="Proxy" sourceParticipantMessage="
  Information" activity="Servant2ProxyProvision"/>
<cim:SubChoreographyActivity name="
  ServantServantRequest" messages="Information
  InformationPetition" sourceParticipant="Servant"
  targetParticipant="Servant" sourceParticipantMessage
  ="InformationPetition" targetParticipantMessage="
  Information" activity="Servant2ServantRequest"/>
</xmi:XMI>
```

IX. Specification of a US-PIM for BlueRose middleware, automatically obtained from the CS-CIM through the proposed QVT transformation rules

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/
  XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst
```

```

ance" xmlns:pim="http://www.ugr.es/~carlosrodriguez/P
IM" xsi:schemaLocation="http://www.ugr.es/~carlosrodr
iguez/PIM ../metamodels/PIM.ecore">
<pim:UbiquitousSystem name="BlueRose" channels="BRBroke
r" protocols="BRProtocol">
  <agents xsi:type="pim:Service" name="ServantService"
    eventhandler="ServantServiceEventHandler" exchange
s="InformationEvent InformationRequest Information
Response InformationPetitionEvent InformationPetit
ionRequest InformationPetitionResponse" discoverer
r="ServantServiceDiscoverer" isRequestedBy="ProxyA
pp" connectsTo="ProxyService ServantService" conne
ctedBy="ProxyService ServantService"/>
  <agents xsi:type="pim:Application" name="ProxyApp" ev
enthandler="ProxyAppEventHandler" exchanges="Infor
mationEvent InformationRequest InformationResponse
InformationPetitionEvent InformationPetitionReque
st InformationPetitionResponse" discoverer="ProxyA
ppDiscoverer" requestsServices="ServantService Pro
xyService"/>
  <agents xsi:type="pim:Service" name="ProxyService" ev
enthandler="ProxyServiceEventHandler" exchanges="I
nformationEvent InformationRequest InformationResp
onse InformationPetitionEvent InformationPetitionR
equest InformationPetitionResponse" discoverer="Pr
oxyServiceDiscoverer" isRequestedBy="ProxyApp" con
nectsTo="ServantService ProxyService" connectedBy=
"ProxyService ServantService"/>
  <choreographies participants="ProxyApp ProxyService S
ervantService" messages="InformationEvent Informat
ionRequest InformationResponse InformationPetition
Event InformationPetitionRequest InformationPetiti
onResponse ParallelEndGatewayDistributionActivityC
onditionalStartingEvent ParallelEndGatewayDistribu
tionActivityConditionalStartingEvent Servant2Serna
ntRequest0ConditionalStartingEvent Proxy2ServantRe
quest0DefaultStartingEvent Proxy2ServantRequest1De
faultStartingEvent Proxy2ProxyRequest2DefaultStart
ingEvent Proxy2ProxyRequest0DefaultStartingEvent P
roxy2ProxyRequest3DefaultStartingEvent Proxy2Proxy
Request1DefaultStartingEvent Servant2ProxyProvisio

```

```
n1DefaultStartingEvent Servant2ProxyProvision0DefaultStartingEvent EndEventDistributionActivityStartingEvent ParallelStartGatewayDistributionActivityStartingEvent Proxy2ProxyRequest0StartingEvent Proxy2ProxyRequest3StartingEvent Proxy2ProxyRequest1StartingEvent Proxy2ProxyRequest2StartingEvent Proxy2ServantRequest1StartingEvent Proxy2ServantRequest0StartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent CloseConnectionGatewayDistributionActivityStartingEvent CloseConnectionGatewayDistributionActivityStartingEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent" name="MainChoreography" startingEvent="InitialEvent" endingEvents="EndEvent">
<activities name="Proxy2ServantRequest0" messages="InformationPetitionEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent" participants="ProxyApp ServantService" startingMessages="Proxy2ServantRequest0DefaultStartingEvent Proxy2ServantRequest0StartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" name="ProxyApp2ServantServiceDiscovery" sourceParticipant="ProxyApp" targetParticipant="ServantService"/>
<transactions xsi:type="pim:EventDistributionActivity" name="ProxyApp2ServantServiceEventDistribution" sourceParticipant="ProxyApp" relatedEvent="InformationPetitionEvent"/>
<transactions xsi:type="pim:MessageExchangingActivity" name="ProxyApp2ServantServiceRequestMessageExchanging" sourceParticipant="ProxyApp" targetParticipant="ServantService" message="InformationPetitionRequest"/>
<transactions xsi:type="pim:EventDistributionActivity" name="ServantAdditionalRequestGatewayDist
```

```
        distributionActivityStartingEventDistributionActivity" sourceParticipant="ProxyApp" relatedEvent="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
    </activities>
    <activities name="Proxy2ServantRequest1" messages="InformationPetitionEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent" participants="ProxyService ServantService" startingMessages="Proxy2ServantRequest1DefaultStartingEvent Proxy2ServantRequest1StartingEvent">
        <transactions xsi:type="pim:DiscoveryActivity" name="ProxyService2ServantServiceDiscovery" sourceParticipant="ProxyService" targetParticipant="ServantService"/>
        <transactions xsi:type="pim:EventDistributionActivity" name="ProxyService2ServantServiceEventDistribution" sourceParticipant="ProxyService" relatedEvent="InformationPetitionEvent"/>
        <transactions xsi:type="pim:MessageExchangingActivity" name="ProxyService2ServantServiceRequestMessageExchanging" sourceParticipant="ProxyService" targetParticipant="ServantService" message="InformationPetitionRequest"/>
        <transactions xsi:type="pim:MessageExchangingActivity" name="ProxyService2ServantServiceResponseMessageExchanging" sourceParticipant="ProxyService" targetParticipant="ServantService" message="InformationPetitionResponse"/>
        <transactions xsi:type="pim:EventDistributionActivity" name="ServantAdditionalRequestGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ProxyService" relatedEvent="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
    </activities>
    <activities name="Servant2ServantRequest0" messages="InformationPetitionEvent InformationEvent ServantAdditionalRequestGatewayDistributionActivityStartingEvent" participants="ServantService" startingMessages="Servant2ServantRequest0Conditiona
```



```
    lStartingEvent">
    <transactions xsi:type="pim:DiscoveryActivity" name="ServantService2ServantServiceDiscovery" sourceParticipant="ServantService" targetParticipant="ServantService"/>
    <transactions xsi:type="pim:EventDistributionActivity" sourceParticipant="ServantService" relatedEvent="InformationPetitionEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" relatedEvent="InformationEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" name="ServantAdditionalRequestGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ServantService" relatedEvent="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
  </activities>
  <activities name="Proxy2ProxyRequest0" messages="InformationPetitionEvent InformationEvent EndProxyRequestsGatewayDistributionActivityStartingEvent" participants="ProxyApp" startingMessages="Proxy2ProxyRequest0DefaultStartingEvent Proxy2ProxyRequest0StartingEvent">
    <transactions xsi:type="pim:DiscoveryActivity" name="ProxyApp2ProxyAppDiscovery" sourceParticipant="ProxyApp" targetParticipant="ProxyApp"/>
    <transactions xsi:type="pim:EventDistributionActivity" sourceParticipant="ProxyApp" relatedEvent="InformationPetitionEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" relatedEvent="InformationEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" name="EndProxyRequestsGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ProxyApp" relatedEvent="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
  </activities>
  <activities name="Proxy2ProxyRequest1" messages="InformationPetitionEvent InformationEvent EndProxyRequestsGatewayDistributionActivityStartingEvent
```

```

    " participants="ProxyApp ProxyService" startingM
    essages="Proxy2ProxyRequest1DefaultStartingEvent
    Proxy2ProxyRequest1StartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" nam
    e="ProxyApp2ProxyServiceDiscovery" sourceParti
    cipant="ProxyApp" targetParticipant="ProxyServ
    ervice"/>
<transactions xsi:type="pim:EventDistributionActiv
    ity" name="ProxyApp2ProxyServiceEventDistribut
    ion" sourceParticipant="ProxyApp" relatedEven
    t="InformationPetitionEvent"/>
<transactions xsi:type="pim:MessageExchangingActiv
    ity" name="ProxyApp2ProxyServiceRequestMessage
    Exchanging" sourceParticipant="ProxyApp" targe
    tParticipant="ProxyService" message="Informati
    onPetitionRequest"/>
<transactions xsi:type="pim:EventDistributionActiv
    ity" name="ProxyService2ProxyAppEventDistribut
    ion" sourceParticipant="ProxyService" relatedE
    vent="InformationEvent"/>
<transactions xsi:type="pim:MessageExchangingActiv
    ity" name="ProxyService2ProxyAppResponseMessag
    eExchanging" sourceParticipant="ProxyService"
    targetParticipant="ProxyApp" message="Informat
    ionResponse"/>
<transactions xsi:type="pim:EventDistributionActiv
    ity" name="EndProxyRequestsGatewayDistribution
    ActivityStartingEventDistributionActivity" sou
    rceParticipant="ProxyApp" relatedEvent="EndPro
    xyRequestsGatewayDistributionActivityStartingE
    vent"/>
</activities>
<activities name="Proxy2ProxyRequest2" messages="In
    formationPetitionEvent InformationEvent EndProxy
    RequestsGatewayDistributionActivityStartingEvent
    " participants="ProxyService ProxyApp" startingM
    essages="Proxy2ProxyRequest2DefaultStartingEvent
    Proxy2ProxyRequest2StartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" nam
    e="ProxyService2ProxyAppDiscovery" sourceParti
    cipant="ProxyService" targetParticipant="Proxy

```

```
App"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="ProxyService2ProxyAppEventDistribut
ion" sourceParticipant="ProxyService" relatedE
vent="InformationPetitionEvent"/>
<transactions xsi:type="pim:MessageExchangingActiv
ity" name="ProxyService2ProxyAppResponseMessag
eExchanging" sourceParticipant="ProxyService"
targetParticipant="ProxyApp" message="Informat
ionPetitionResponse"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="ProxyApp2ProxyServiceEventDistribut
ion" sourceParticipant="ProxyApp" relatedEven
t="InformationEvent"/>
<transactions xsi:type="pim:MessageExchangingActiv
ity" name="ProxyApp2ProxyServiceRequestMessage
Exchanging" sourceParticipant="ProxyApp" targe
tParticipant="ProxyService" message="Informati
onRequest"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="EndProxyRequestsGatewayDistribution
ActivityStartingEventDistributionActivity" sou
rceParticipant="ProxyService" relatedEvent="En
dProxyRequestsGatewayDistributionActivityStart
ingEvent"/>
</activities>
<activities name="Proxy2ProxyRequest3" messages="In
formationPetitionEvent InformationEvent EndProxy
RequestsGatewayDistributionActivityStartingEvent
" participants="ProxyService" startingMessages="
Proxy2ProxyRequest3DefaultStartingEvent Proxy2Pr
oxyRequest3StartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" nam
e="ProxyService2ProxyServiceDiscovery" sourceP
articipant="ProxyService" targetParticipant="P
roxyService"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" sourceParticipant="ProxyService" relatedE
vent="InformationPetitionEvent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" relatedEvent="InformationEvent"/>
```

```
<transactions xsi:type="pim:EventDistributionActivity" name="EndProxyRequestsGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ProxyService" relatedEvent="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
</activities>
<activities name="Servant2ProxyProvision0" messages="InformationEvent CloseConnectionGatewayDistributionActivityStartingEvent" participants="ServantService ProxyApp" startingMessages="Servant2ProxyProvision0DefaultStartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" name="ServantService2ProxyAppDiscovery" sourceParticipant="ServantService" targetParticipant="ProxyApp"/>
<transactions xsi:type="pim:EventDistributionActivity" name="ServantService2ProxyAppEventDistribution" sourceParticipant="ServantService" relatedEvent="InformationEvent"/>
<transactions xsi:type="pim:MessageExchangingActivity" name="ServantService2ProxyAppResponseMessageExchanging" sourceParticipant="ServantService" targetParticipant="ProxyApp" message="InformationResponse"/>
<transactions xsi:type="pim:EventDistributionActivity" name="CloseConnectionGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ServantService" relatedEvent="CloseConnectionGatewayDistributionActivityStartingEvent"/>
</activities>
<activities name="Servant2ProxyProvision1" messages="InformationEvent CloseConnectionGatewayDistributionActivityStartingEvent" participants="ServantService ProxyService" startingMessages="Servant2ProxyProvision1DefaultStartingEvent">
<transactions xsi:type="pim:DiscoveryActivity" name="ServantService2ProxyServiceDiscovery" sourceParticipant="ServantService" targetParticipant="ProxyService"/>
```

```
<transactions xsi:type="pim:EventDistributionActivity" name="ServantService2ProxyServiceEventDistribution" sourceParticipant="ServantService" relatedEvent="InformationEvent"/>
<transactions xsi:type="pim:MessageExchangingActivity" name="ServantService2ProxyServiceRequestMessageExchanging" sourceParticipant="ServantService" targetParticipant="ProxyService" message="InformationRequest"/>
<transactions xsi:type="pim:MessageExchangingActivity" name="ServantService2ProxyServiceResponseMessageExchanging" sourceParticipant="ServantService" targetParticipant="ProxyService" message="InformationResponse"/>
<transactions xsi:type="pim:EventDistributionActivity" name="CloseConnectionGatewayDistributionActivityStartingEventDistributionActivity" sourceParticipant="ServantService" relatedEvent="CloseConnectionGatewayDistributionActivityStartingEvent"/>
</activities>
<activities name="CloseConnectionGatewayDistributionActivity" messages="ParallelEndGatewayDistributionActivityConditionalStartingEvent Proxy2ServantRequest1DefaultStartingEvent Proxy2ServantRequest0DefaultStartingEvent" startingMessages="CloseConnectionGatewayDistributionActivityStartingEvent CloseConnectionGatewayDistributionActivityStartingEvent">
<transactions xsi:type="pim:EventDistributionActivity" name="CloseConnectionGatewayDistributionTransactionActivity"/>
<transactions xsi:type="pim:EventDistributionActivity" name="ParallelEndGatewayDistributionActivityConditionalStartingEventDistributionActivity" relatedEvent="ParallelEndGatewayDistributionActivityConditionalStartingEvent"/>
<transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ServantRequest1DefaultStartingEventDistributionActivity" relatedEvent="Proxy2ServantRequest1DefaultStartingEvent"/>
```

```
<transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ServantRequest0DefaultStartingEventDistributionActivity" relatedEvent="Proxy2ServantRequest0DefaultStartingEvent"/>
</activities>
<activities name="ParallelEndGatewayDistributionActivity" messages="EndEventDistributionActivityStartingEvent" startingMessages="ParallelEndGatewayDistributionActivityConditionalStartingEvent ParallelEndGatewayDistributionActivityConditionalStartingEvent">
  <transactions xsi:type="pim:EventDistributionActivity" name="ParallelEndGatewayDistributionTransactionActivity"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="EndEventDistributionActivityStartingEventDistributionActivity" relatedEvent="EndEventDistributionActivityStartingEvent"/>
</activities>
<activities name="EndProxyRequestsGatewayDistributionActivity" messages="ParallelEndGatewayDistributionActivityConditionalStartingEvent Proxy2ProxyRequest2DefaultStartingEvent Proxy2ProxyRequest1DefaultStartingEvent Proxy2ProxyRequest3DefaultStartingEvent Proxy2ProxyRequest0DefaultStartingEvent" startingMessages="EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent EndProxyRequestsGatewayDistributionActivityStartingEvent">
  <transactions xsi:type="pim:EventDistributionActivity" name="EndProxyRequestsGatewayDistributionTransactionActivity"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="ParallelEndGatewayDistributionActivityConditionalStartingEventDistributionActivity" relatedEvent="ParallelEndGatewayDistributionActivityConditionalStartingEvent"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ProxyRequest2DefaultStartingE
```

```
    ventDistributionActivity" relatedEvent="Proxy2
    ProxyRequest2DefaultStartingEvent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Proxy2ProxyRequest1DefaultStartingE
ventDistributionActivity" relatedEvent="Proxy2
ProxyRequest1DefaultStartingEvent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Proxy2ProxyRequest3DefaultStartingE
ventDistributionActivity" relatedEvent="Proxy2
ProxyRequest3DefaultStartingEvent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Proxy2ProxyRequest0DefaultStartingE
ventDistributionActivity" relatedEvent="Proxy2
ProxyRequest0DefaultStartingEvent"/>
</activities>
<activities name="ServantAdditionalRequestGatewayDi
stributionActivity" messages="Servant2ServantReq
uest0ConditionalStartingEvent Servant2ProxyProvi
sion0DefaultStartingEvent Servant2ProxyProvisio
n1DefaultStartingEvent" startingMessages="Servan
tAdditionalRequestGatewayDistributionActivitySta
rtingEvent ServantAdditionalRequestGatewayDistri
butionActivityStartingEvent ServantAdditionalReq
uestGatewayDistributionActivityStartingEvent">
<transactions xsi:type="pim:EventDistributionActiv
ity" name="ServantAdditionalRequestGatewayDist
ributionTransactionActivity"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Servant2ServantRequest0Conditionals
tartingEventDistributionActivity" relatedEven
t="Servant2ServantRequest0ConditionalStartingE
vent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Servant2ProxyProvision0DefaultStart
ingEventDistributionActivity" relatedEvent="Se
rvant2ProxyProvision0DefaultStartingEvent"/>
<transactions xsi:type="pim:EventDistributionActiv
ity" name="Servant2ProxyProvision1DefaultStart
ingEventDistributionActivity" relatedEvent="Se
rvant2ProxyProvision1DefaultStartingEvent"/>
</activities>
```

```
<activities name="EndEventDistributionActivity" startingMessages="EndEventDistributionActivityStartingEvent">
  <transactions xsi:type="pim:EventDistributionActivity" name="EndEventDistributionTransactionActivity" relatedEvent="EndEvent"/>
</activities>
<activities name="InitialEventDistributionActivity" messages="ParallelStartGatewayDistributionActivityStartingEvent">
  <transactions xsi:type="pim:EventDistributionActivity" name="InitialEventDistributionTransactionActivity" relatedEvent="InitialEvent"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="ParallelStartGatewayDistributionActivityStartingEventDistributionActivity" relatedEvent="ParallelStartGatewayDistributionActivityStartingEvent"/>
</activities>
<activities name="ParallelStartGatewayDistributionActivity" messages="Proxy2ProxyRequest2StartingEvent Proxy2ProxyRequest1StartingEvent Proxy2ProxyRequest3StartingEvent Proxy2ProxyRequest0StartingEvent Proxy2ServantRequest1StartingEvent Proxy2ServantRequest0StartingEvent" startingMessages="ParallelStartGatewayDistributionActivityStartingEvent">
  <transactions xsi:type="pim:EventDistributionActivity" name="ParallelStartGatewayDistributionTransactionActivity"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ProxyRequest2StartingEventDistributionActivity" relatedEvent="Proxy2ProxyRequest2StartingEvent"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ProxyRequest1StartingEventDistributionActivity" relatedEvent="Proxy2ProxyRequest1StartingEvent"/>
  <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ProxyRequest3StartingEventDistributionActivity" relatedEvent="Proxy2ProxyRe
```



```
    quest3StartingEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ProxyRequest0StartingEventDistributionActivity" relatedEvent="Proxy2ProxyRequest0StartingEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ServantRequest1StartingEventDistributionActivity" relatedEvent="Proxy2ServantRequest1StartingEvent"/>
    <transactions xsi:type="pim:EventDistributionActivity" name="Proxy2ServantRequest0StartingEventDistributionActivity" relatedEvent="Proxy2ServantRequest0StartingEvent"/>
  </activities>
</choreographies>
</pim:UbiquitousSystem>
<pim:Topic name="InformationTopic" isTopicOf="InformationEvent"/>
<pim:Event name="InformationEvent" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" topic="InformationTopic" handledBy="ServantServiceEventHandler ProxyAppEventHandler ProxyServiceEventHandler" listenedBy="ServantServiceInformationEventListener ProxyAppInformationEventListener ProxyServiceInformationEventListener" acceptedBy="InformationPredicate"/>
<pim:Predicate name="InformationPredicate" accepts="InformationEvent" constraints="ServantServiceInformationEventListener ProxyAppInformationEventListener ProxyServiceInformationEventListener"/>
<pim:Request name="InformationRequest" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" reply="InformationResponse" targets="ProxyService"/>
<pim:Response name="InformationResponse" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" petition="InformationRequest"/>
<pim:Topic name="InformationPetitionTopic" isTopicOf="InformationPetitionEvent"/>
```

```
<pim:Event name="InformationPetitionEvent" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" topic="InformationPetitionTopic" handledBy="ServantServiceEventHandler ProxyAppEventHandler ProxyServiceEventHandler" listenedBy="ServantServiceInformationPetitionEventListener ProxyAppInformationPetitionEventListener ProxyServiceInformationPetitionEventListener" acceptedBy="InformationPetitionPredicate"/>
<pim:Predicate name="InformationPetitionPredicate" accepts="InformationPetitionEvent" constraints="ServantServiceInformationPetitionEventListener ProxyAppInformationPetitionEventListener ProxyServiceInformationPetitionEventListener"/>
<pim:Request name="InformationPetitionRequest" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" reply="InformationPetitionResponse" targets="ServantService ProxyService"/>
<pim:Response name="InformationPetitionResponse" exchangedFrom="ServantService ProxyApp ProxyService" medium="BRBroker" conforms="BRProtocol" petition="InformationPetitionRequest"/>
<pim:Discovery name="ServantServiceDiscoveryEvent" medium="BRBroker" conforms="BRProtocol" topic="ServantServiceDiscoveryTopic" discovers="ServantServiceDiscoveryListener"/>
<pim:Topic name="ServantServiceDiscoveryTopic" isTopicOf="ServantServiceDiscoveryEvent"/>
<pim:Discoverer name="ServantServiceDiscoverer" listeners="ServantServiceDiscoveryListener" callback="ServantServiceDiscoveryListener"/>
<pim:DiscoveryListener name="ServantServiceDiscoveryListener" constrainedBy="ServantServiceDiscoveryPredicate" seeker="ServantServiceDiscoverer" discoveredBy="ServantServiceDiscoveryEvent"/>
<pim:Predicate name="ServantServiceDiscoveryPredicate" constraints="ServantServiceDiscoveryListener"/>
<pim:EventHandler name="ServantServiceEventHandler" handles="InformationEvent InformationPetitionEvent" listeners="ServantServiceInformationEventListener Serva
```

```
ntServiceInformationPetitionEventListener"/>
<pim:EventListener name="ServantServiceInformationEvent
  Listener" listens="InformationEvent" constrainedBy="
  InformationPredicate"/>
<pim:EventListener name="ServantServiceInformationPetit
  ionEventListener" listens="InformationPetitionEvent"
  constrainedBy="InformationPetitionPredicate"/>
<pim:Discovery name="ProxyAppDiscoveryEvent" medium="BR
  Broker" conforms="BRProtocol" topic="ProxyAppDiscove
  ryTopic" discovers="ProxyAppDiscoveryListener"/>
<pim:Topic name="ProxyAppDiscoveryTopic" isTopicOf="Pro
  xyAppDiscoveryEvent"/>
<pim:Discoverer name="ProxyAppDiscoverer" listeners="Pr
  oxyAppDiscoveryListener" callback="ProxyAppDiscovery
  Listener"/>
<pim:DiscoveryListener name="ProxyAppDiscoveryListener"
  constrainedBy="ProxyAppDiscoveryPredicate" seeker="
  ProxyAppDiscoverer" discoveredBy="ProxyAppDiscoveryE
  vent"/>
<pim:Predicate name="ProxyAppDiscoveryPredicate" constr
  aints="ProxyAppDiscoveryListener"/>
<pim:EventHandler name="ProxyAppEventHandler" handles="
  InformationEvent InformationPetitionEvent" listener
  s="ProxyAppInformationEventListener ProxyAppInformat
  ionPetitionEventListener"/>
<pim:EventListener name="ProxyAppInformationEventListen
  er" listens="InformationEvent" constrainedBy="Inform
  ationPredicate"/>
<pim:EventListener name="ProxyAppInformationPetitionEve
  ntListener" listens="InformationPetitionEvent" const
  rainedBy="InformationPetitionPredicate"/>
<pim:Discovery name="ProxyServiceDiscoveryEvent" mediu
  m="BRBroker" conforms="BRProtocol" topic="ProxyServi
  ceDiscoveryTopic" discovers="ProxyServiceDiscoveryLi
  stener"/>
<pim:Topic name="ProxyServiceDiscoveryTopic" isTopicOf=
  "ProxyServiceDiscoveryEvent"/>
<pim:Discoverer name="ProxyServiceDiscoverer" listener
  s="ProxyServiceDiscoveryListener" callback="ProxySer
  viceDiscoveryListener"/>
```

```
<pim:DiscoveryListener name="ProxyServiceDiscoveryListener" constrainedBy="ProxyServiceDiscoveryPredicate"
  seeker="ProxyServiceDiscoverer" discoveredBy="ProxyServiceDiscoveryEvent"/>
<pim:Predicate name="ProxyServiceDiscoveryPredicate" constraints="ProxyServiceDiscoveryListener"/>
<pim:EventHandler name="ProxyServiceEventHandler" handles="InformationEvent InformationPetitionEvent" listeners="ProxyServiceInformationEventListener ProxyServiceInformationPetitionEventListener"/>
<pim:EventListener name="ProxyServiceInformationEventListener" listens="InformationEvent" constrainedBy="InformationPredicate"/>
<pim:EventListener name="ProxyServiceInformationPetitionEventListener" listens="InformationPetitionEvent" constrainedBy="InformationPetitionPredicate"/>
<pim:SoftwareProtocol name="BRProtocol" system="BlueRose" rules="BRBroker" compiles="InformationEvent InformationRequest InformationResponse InformationPetitionEvent InformationPetitionRequest InformationPetitionResponse ServantServiceDiscoveryEvent ProxyAppDiscoveryEvent ProxyServiceDiscoveryEvent"/>
<pim:NetworkingTechnology name="BRBroker" system="BlueRose" transports="InformationEvent InformationRequest InformationResponse InformationPetitionEvent InformationPetitionRequest InformationPetitionResponse ServantServiceDiscoveryEvent ProxyAppDiscoveryEvent ProxyServiceDiscoveryEvent" conforms="BRProtocol"/>
<pim:Initial name="InitialEvent" topic="InitialEventTopic" initiatedChoreographies="MainChoreography"/>
<pim:Topic name="InitialEventTopic" isTopicOf="InitialEvent"/>
<pim:End name="EndEvent" topic="EndEventTopic" endedChoreographies="MainChoreography"/>
<pim:Topic name="EndEventTopic" isTopicOf="EndEvent"/>
<pim:Conditional name="ParallelEndGatewayDistributionActivityConditionalStartingEvent" predicate="ParallelEndGatewayDistributionActivityConditionalStartingEventPredicate"/>
<pim:Predicate name="ParallelEndGatewayDistributionActivityConditionalStartingEventPredicate"/>
```

```
<pim:Conditional name="ParallelEndGatewayDistributionActivityConditionalStartingEvent" predicate="ParallelEndGatewayDistributionActivityConditionalStartingEventPredicate"/>
<pim:Predicate name="ParallelEndGatewayDistributionActivityConditionalStartingEventPredicate"/>
<pim:Conditional name="Servant2ServantRequest0ConditionalStartingEvent" predicate="Servant2ServantRequest0ConditionalStartingEventPredicate"/>
<pim:Predicate name="Servant2ServantRequest0ConditionalStartingEventPredicate"/>
<pim:Default name="Proxy2ServantRequest1DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ServantRequest1DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ServantRequest1DefaultStartingEventPredicate"/>
<pim:Default name="Proxy2ServantRequest0DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ServantRequest0DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ServantRequest0DefaultStartingEventPredicate"/>
<pim:Default name="Proxy2ProxyRequest2DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ProxyRequest2DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ProxyRequest2DefaultStartingEventPredicate"/>
<pim:Default name="Proxy2ProxyRequest1DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ProxyRequest1DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ProxyRequest1DefaultStartingEventPredicate"/>
<pim:Default name="Proxy2ProxyRequest3DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ProxyRequest3DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ProxyRequest3DefaultStartingEventPredicate"/>
<pim:Default name="Proxy2ProxyRequest0DefaultStartingEvent" nonSatisfiedPredicate="Proxy2ProxyRequest0DefaultStartingEventPredicate"/>
<pim:Predicate name="Proxy2ProxyRequest0DefaultStartingEventPredicate"/>
```

```
<pim:Default name="Servant2ProxyProvision0DefaultStartingEvent" nonSatisfiedPredicate="Servant2ProxyProvision0DefaultStartingEventPredicate"/>
<pim:Predicate name="Servant2ProxyProvision0DefaultStartingEventPredicate"/>
<pim:Default name="Servant2ProxyProvision1DefaultStartingEvent" nonSatisfiedPredicate="Servant2ProxyProvision1DefaultStartingEventPredicate"/>
<pim:Predicate name="Servant2ProxyProvision1DefaultStartingEventPredicate"/>
<pim:Sequential name="EndEventDistributionActivityStartingEvent"/>
<pim:Sequential name="ParallelStartGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="Proxy2ProxyRequest2StartingEvent"/>
<pim:Sequential name="Proxy2ProxyRequest1StartingEvent"/>
<pim:Sequential name="Proxy2ProxyRequest3StartingEvent"/>
<pim:Sequential name="Proxy2ProxyRequest0StartingEvent"/>
<pim:Sequential name="Proxy2ServantRequest1StartingEvent"/>
<pim:Sequential name="Proxy2ServantRequest0StartingEvent"/>
<pim:Sequential name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="EndProxyRequestsGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="ServantAdditionalRequestGatewayDistributionActivityStartingEvent"/>
<pim:Sequential name="CloseConnectionGatewayDistributionActivityStartingEvent"/>
```

```
<pim:Sequential name="CloseConnectionGatewayDistributio
nActivityStartingEvent"/>
<pim:Sequential name="ServantAdditionalRequestGatewayDi
stributionActivityStartingEvent"/>
</xmi:XMI>
```