Towards smart services with reusable and adaptable connected objects : An application to wearable non-invasive biomedical sensors

Arthur Gatouillat

N°d'ordre NNT : 2018LYSEI123

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
INSA-Lyon

Ecole Doctorale N° 512
InfoMaths

Spécialité/discipline de doctorat :
Informatique

Soutenue publiquement le 20/12/2018, par :
Arthur Yves Étienne GATOUILLAT

Towards Smart Services with Reusable and Adaptable Connected Objects:

An Application to Wearable Non-Invasive Biomedical Sensors.

Devant le jury composé de :

SONG, Ye-Qiong	Professeur des Universités, Université de Lorraine	Rapporteur
RIVENQ, Atika	Professeure des Universités, UPHF	Rapporteure
MALENFANT, Jacques	Professeur des Universités, Sorbonne Université	Examinateur
GUESSOUM, Zahia	Maître de Conférences HDR, Sorbonne Université	Examinatrice
BADR, Youakim	Maître de Conférences HDR, INSA-Lyon	Directeur de thèse
MASSOT, Bertrand	Maître de Conférences, INSA-Lyon	Co-encadrant de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage **secretariat@edchimie-lyon.fr** INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX **directeur@edchimie-lyon.fr**
E.E.A.	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Sec. : M.C. HAVGOUDOUKIAN **ecole-doctorale.eea@ec-lyon.fr**	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 **gerard.scorletti@ec-lyon.fr**
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http://e2m2.universite-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES **secretariat.e2m2@univ-lyon1.fr**	M. Philippe NORMAND UMR 5557 Lab. d'Ecologie Microbienne Université Claude Bernard Lyon 1 Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX **philippe.normand@univ-lyon1.fr**
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http://www.ediss-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE **secretariat.ediss@univ-lyon1.fr**	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne Tél : 04.72.68.49.09 Fax : 04.72.68.49.16 **emmanuelle.canet@univ-lyon1.fr**
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 Fax : 04.72.43.16.87 **infomaths@univ-lyon1.fr**	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX Tél : 04.26.23.45.52 **zamboni@maths.univ-lyon1.fr**
Matériaux	**MATÉRIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **ed.materiaux@insa-lyon.fr**	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 **jean-yves.buffiere@insa-lyon.fr**
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http://edmega.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **mega@insa-lyon.fr**	M. Jocelyn BONJOUR INSA de Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69 621 Villeurbanne CEDEX **jocelyn.bonjour@insa-lyon.fr**
ScSo	**ScSo*** **http://ed483.univ-lyon2.fr** Sec. : Viviane POLSINELLI Brigitte DUBOIS INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 **viviane.polsinelli@univ-lyon2.fr**	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 **christian.montes@univ-lyon2.fr**

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Abstract

The rapid growth of fixed and mobile smart objects raises the issue of their integration in everyday environment, e.g. in e-health or home-automation contexts. The main challenges of these objects are the interoperability, the handling of the massive amount of data that they generate, and their limited resources. Our goal is to take a bottom-up approach in order to improve the integration of smart devices to smart services. To ensure the efficient development of our approach, we start with the study of the design process of such devices regardless of specific hardware or software through the consideration of their cyber-physical properties. We thus develop two research directions: the specification of a service-oriented design method for smart devices with formal considerations in order to validate their behavior, and the proposal of a self-adaptation framework in order to handle changing operating context through self-reasoning and the definition of a declarative self-adaptation objectives specification language. The testing of these contributions will be realized through the development of a large-scale experimental framework based on a remote diagnostics case-study relying on non-invasive wearable biomedical sensors.

Keywords — Internet-of-Things ; Smart devices ; Design method ; Self-adaptation.

Résumé

La prolifération des objets communicants fixes et mobiles soulève la question de leur intégration dans les environnements quotidiens, par exemple dans le cadre de la e-santé ou de la domotique. Les principaux défis soulevés relèvent de l'interconnexion et de la gestion de la masse de donnée produite par ces objets intelligents. Notre premier objectif est d'adopter une démarche des couches basses vers les couches hautes pour faciliter l'intégration de ces objets à des services intelligents. Afin de développer celle-ci, il est nécessaire de d'étudier le processus de conception des objets intelligents indépendamment de considérations matérielles et logicielles, au travers de la considération de leur propriétés cyber-physiques. Pour mener à bien la réalisation de services intelligents à partir d'objets connectés, les deux axes de recherche suivant seront développés : la définition d'une méthode de conception orientée service pour les objets connectés intégrant une dimension formelle ainsi de valider le comportement de ceux-ci, l'auto-adaptation intelligente dans un contexte évolutif permettant aux objets de raisonner sur eux même au travers d'un langage déclaratif pour spécifier les stratégies d'adaptation. La validation de ces contributions s'effectuera par le biais du développement et de l'expérimentation à grandeur nature d'un service de diagnostic médical continu basé sur la collecte de données médicales en masse par des réseaux non-intrusifs de capteurs biomédicaux portables sur le corps humain.

Mot-clefs — Internet des objets ; Objets intelligents ; Méthode de conception ; Auto-adaptation.

To Yves Menguy and Étienne Gatouillat.
To my family.

Acknowledgements

I would like to thank Dr. Youakim BADR and Dr. Bertrand MASSOT for their guidance during my PhD. I am also deeply grateful to Dr. Ervin SEJDIĆ from University of Pittsburgh for his precious advice and collaborations. Their mentorship was of tremendous help during the three years of my PhD.
In addition, I would like to thank the members of the service-oriented computing team at LIRIS and the members of the biomedical sensors team at INL. More specifically, I express my gratitude towards Dr. Claudine GEHIN, Dr. Amalric MONTALIBET, Pr. Norbert NOURY and Pr. Eric MCADAMS, Arthur CLAUDE and Dr. Loïc SEVRIN. Our discussions were always extremely educational and rewarding. I would also like to thank Dr. Maroun ABI ASSAF and Xiaoyang ZHU, who were of valued help on numerous occasions.
Finally, I would like to thank my friends and family. First and foremost I most sincerely thank my parents, who gave me an excellent education and continue to give me continuous encouragement and support. I also want to mention my brother and sister, who are always by my side and represent a tremendous motivation. Finally, I thank all my friends and family for their great patience and listening during my times of doubts, and also for their understanding during my times of unavailability. I have a special thought for Pierre, Ingrid, Timothée, Chloé, Océane, Emmanuel, Howard, Clara, Jonathan, Hélène, and all the others I forgot to mention.
Last but not least, I can't thank Felipe DIAS enough for his continuous support, presence and company. We're a great team.

List of Figures

1.1 Layer-based representation of the IoT 2
1.2 Layer-based OSI stack 6
1.3 Dissertation outline diagram 11

2.1 Services orchestration vs choreography 18
2.2 Comparison of OSI, TCP/IP, Internet and CoAP-based stacks 20
2.3 Illustration of the FRACTAL component model 22
2.4 Illustration of a simple Node-RED program 23
2.5 Illustration of a timed automaton 25
2.6 Illustration of model checking 26
2.7 Exemplified taxonomy of typical smart devices digital hardware 32
2.8 Classical feedback loop architecture 36
2.9 MAPE-K feedback loop 39
2.10 DYNAMICO reference model 40
2.11 OSGi architecture 41
2.12 Asynchronous vs synchronous reactive systems 43

3.1 Smart devices detailed ontology with a class instance toy example 55
3.2 Smart device detailed resources ontology 56
3.3 Method lifecycle 57
3.4 Mutual constraints relationship 58
3.5 Modular architectural simple generic example 60
3.6 Cardiorespiratory sensor modular architecture 65
3.7 Specifications of heart rate detection, processing, and integration 66
3.8 Heart Rate service specification: PlusCal and automatically generated TLA+ 68
3.9 First version of the cardiorespiratory sensor prototype 69

4.1 Illustration of personalized healthcare for our case study 74
4.2 Simplified block diagram of the sensor 77
4.3 Mounted PCB (a.), body placement (b.) and encased sensor (c.) 78
4.4 Hardware vs software signal processing 80
4.5 Diagram of hardware-based RR interval detection 82
4.6 Cardiorespiratory sensor LTS 86
4.7 Screenshot of the developed Android companion application 87
4.8 Sensor evaluation using synthetic ECG 88
4.9 Impact of the BLE latency parameter on global power consumption 89
4.10 Electrodes' placement (a.) and nature (b.) evaluation 91
4.11 Bland-Altman diagram (a.) and comparative tachograms (b.) 93

4.12 Ambulatory HR and RWF 94
4.13 Ambulatory HRV parameters 94

5.1 Detailed QoS-driven self-adaptation framework 103
5.2 SLA ontology 105
5.3 Cardiorespiratory sensor LTS 107
5.4 Fine and gross position sensors LTSs 108
5.5 EDA sensor LTS (a.) and its Heptagon/BZR specification (b.) 109
5.6 Heptagon/BZR control contract example 110
5.7 Discrete controller synthesis workflow 110
5.8 Self-adaptation language Backus-Naur form grammar 114
5.9 Simplified sample adaptation goals using our rule-based language 115
5.10 Simplified hybrid self-adaptation framework 118

6.1 Basic Drools rule syntax example 127
6.2 Simplified implementation diagram 131
6.3 Implementation dependency graph 132
6.4 Objectives feedback loop class diagram 132
6.5 Objectives feedback loop activity diagram 133
6.6 Adaptation feedback loop class diagram 134
6.7 Adaptation feedback loop activity diagram 134
6.8 Monitoring feedback loop class diagram 135
6.9 Monitoring feedback loop activity diagram 136
6.10 Deployment life cycle 138
6.11 GUI welcome screen 139
6.12 Global view 140
6.13 Physiological data view 140
6.14 System administrator view 141
6.15 Objectives analysis duration as a function of number of rules 142
6.16 Objectives analysis duration as a function of number of rules 143

List of Tables

2.1 Summary of IoT challenges and solutions from SOA, MDE, CPS 28

3.1 Detailed integration test description 61

4.1 HRV paramters description 76
4.2 Static power characterization of IoT states 89
4.3 Precision and sensitivity of the sensor 92

6.1 Prototype dependencies names and versions 131

Contents

1 Introduction **1**
- 1.1 IoT and Smart Devices Generalities . . . 1
- 1.2 IoT Research Challenges . . . 3
  - 1.2.1 Unconventional Characteristics of Smart Devices . . . 3
  - 1.2.2 Global IoT Heterogeneity . . . 6
  - 1.2.3 The IoT from a Multidisciplinary Research Perspective . . . 7
- 1.3 Research Strategies . . . 8
  - 1.3.1 Research Statement . . . 8
  - 1.3.2 Improving Smart Devices Design to facilitate IoT Integration . . . 9
  - 1.3.3 Self-Adaptation: a Requirements of Smart Services . . . 10
- 1.4 Dissertation Outline . . . 11

2 State of the Art **13**
- 2.1 Introduction . . . 13
- 2.2 The IoT: Horizontal Solutions . . . 16
  - 2.2.1 Service-Oriented Architectures in the IoT . . . 17
  - 2.2.2 Model-Based and Graphical Development in the IoT . . . 20
  - 2.2.3 IoT Smart Devices as Cyber-Physical Systems . . . 24
  - 2.2.4 Summary of Horizontal Contributions . . . 27
- 2.3 Designing Embedded Systems . . . 28
  - 2.3.1 Systems Design Methods . . . 29
  - 2.3.2 Hardware Solutions for Smart Devices . . . 32
- 2.4 Self-Adaptation Solutions . . . 35
  - 2.4.1 Classical Control: Theory and Tools . . . 35
  - 2.4.2 Autonomic Computing: Self-Adaptive Software Systems . . . 38
  - 2.4.3 Reactive Systems and Discrete Controller Synthesis . . . 42
- 2.5 Summary and Conclusion . . . 46

3 Design Method for Service Oriented Smart Devices **49**
- 3.1 Introduction . . . 49
- 3.2 Method Overview . . . 52
  - 3.2.1 Generalities . . . 52
  - 3.2.2 Smart Devices Ontology . . . 54
- 3.3 Comprehensive Method Description . . . 56
  - 3.3.1 Smart Device Requirements Analysis . . . 58
  - 3.3.2 Smart Device Constraints Analysis . . . 58
  - 3.3.3 Modular Architecture Design . . . 59

3.3.4 Modules Formal Specification 60
3.3.5 Smart Component Formal Specification and Verification 60
3.3.6 Modules Implementation and Integration 61
3.3.7 Smart Device Testing 61
3.3.8 Smart Device Production 62
3.3.9 Adding Self-Adaptive Behavior as a System-Wide Requirement . 62
3.4 Medical Smart Device Design 63
3.4.1 Smart Device Analysis 63
3.4.2 Smart Device Specification 65
3.4.3 Smart Device Implementation: an Overview 67
3.5 Summary and Conclusion 69

4 Smart Device Hardware Design and Implementation 73
4.1 Introduction 73
4.2 Sensor Scope Statement 75
4.3 Sensor Design 76
4.3.1 Hardware Architecture 76
4.3.2 Embedded Hardware and Software Signal Processing 79
4.3.3 Integrating Self-Adaptive Behavior 84
4.3.4 Development of an Android Companion Application 86
4.4 Sensor Evaluation 87
4.4.1 Evaluation on Synthetic Signals 87
4.4.2 Energy Consumption 88
4.4.3 Sensor Precision Evaluation 90
4.4.4 Mobility Evaluation 93
4.5 Conclusion 95

5 Self-Adaptation Framework for Smart Devices 97
5.1 Introduction 97
5.2 Self-Adaptation Case-Study 100
5.3 Dynamic and Synchronous QoS-Driven Self-Adaptation for the IoT . . 101
5.3.1 Managing Dynamic Objectives and Monitoring Infrastructure . 102
5.3.2 Synchronous Programming Languages 106
5.4 Adaptation Objectives Specification 111
5.4.1 Declarative Self-Adaptation Specification 111
5.4.2 Specifying Adaptation Objectives with a Rule-Based Language . 113
5.5 From Synchronous to Hybrid Self-Adaptation 116
5.6 Conclusion 118

6 Implementing the Self-Adaptation Framework 121
6.1 Introduction 121
6.2 Selected Technical Solutions 123
6.2.1 The Data Distribution Service Standard 123
6.2.2 Asynchronous Reactive Systems Using Vert.x 125
6.2.3 The Drools Rule Engine 127
6.2.4 The Maven Software Management Tool 129
6.2.5 MongoDB Database 129

6.3 Implementation Architecture 130
6.3.1 Global Architecture 130
6.3.2 Implementation Model 131
6.3.3 Implementing the GUI 136
6.3.4 Deployment Life Cycle 137
6.4 Implementation Evaluation 137
6.4.1 Case Study 138
6.4.2 Experimental Results 139
6.5 Conclusion 143

7 Conclusion and Perspectives **145**
7.1 Summary of the Contributions 145
7.1.1 Service-Oriented Design Method for Smart Devices 146
7.1.2 Dynamic Self-Adaptation Framework 147
7.2 Future Research Directions 148

A Résumé Long en Français **151**
A.1 Introduction 151
A.2 Méthode de Conception Orientée Service pour Objets Intelligents . . . 156
A.2.1 Description Générale 156
A.2.2 Application de la Méthode à un Capteur Cardiorespiratoire . . . 160
A.3 Infrastructure d'Auto-Adaptation Dédiée aux Objets Intelligents 162
A.3.1 Présentation de l'Infrastructure Hybride 164
A.3.2 Implémentation de l'Infrastructure d'Auto-Adaptation 166
A.4 Conclusion 168

B Cardiorespiratory Sensor Schematic **171**

Bibliography **175**

CHAPTER 1

Introduction

Contents

1.1 IoT and Smart Devices Generalities 1
1.2 IoT Research Challenges 3
1.2.1 Unconventional Characteristics of Smart Devices 3
1.2.2 Global IoT Heterogeneity 6
1.2.3 The IoT from a Multidisciplinary Research Perspective 7
1.3 Research Strategies 8
1.3.1 Research Statement 8
1.3.2 Improving Smart Devices Design to facilitate IoT Integration . . 9
1.3.3 Self-Adaptation: a Requirements of Smart Services 10
1.4 Dissertation Outline 11

1.1 IoT and Smart Devices Generalities

The Internet-of-Things, also called the IoT, designates the interconnection of virtually any objects from the physical world, and their integration to wide-scale Internet-based frameworks. The IoT is rapidly growing, and 20 billions of connected devices are estimated to be available in 2020 [Cer+15]. This growth is also expected to have tremendous economic impact, as total hardware spending related to the IoT is expected to reach $2.9 trillion by 2020 [Gar17].

In addition to economic impacts, the IoT is also going to cause major societal changes. Indeed, IoT-based solutions can be used for a variety of applications and services, such as healthcare [BXA17], traffic control [CZ16], smart cities [Zan+14] just to mention a few. Because these applications and services are strongly related to the physical world, purely software solutions are not able to provide comprehensive representations of all the problems faced in this cyber-physical context. For instance, smart healthcare systems must be able to monitor a wide variety of physiological functions through sensors, or be able to change patients' physiological functions using drugs. This proximity to the physical world brings new requirements on smart devices and IoT-based systems (e.g., limited resources, mobility, human-in-the loop, etc.).

IoT-based applications come with very different challenges and requirements. However, they also share a common set of requirements [Sta14], especially in terms of system *reliability*, *safety* and *security*:

- *Reliability*: a system is considered reliable if functional requirements are verified at all times. More specifically, reliability implies that unexpected system failures must not occur under normal operating conditions. This requirement is particularly true for IoT-based systems as they are often used in critical applications (e.g., healthcare, traffic control, etc.), where system failures could have potential life-threatening consequences.
- *Safety*: a safe system is defined as not causing harm to its environment. This characteristic is extremely important in IoT-based systems, as the IoT is strongly related to the physical world in which damages can have disastrous consequences (e.g., misestimation of system's parameters could cause life-threatening accidents).
- *Security*: a secure system is robust to external and internal malicious attacks. In order to be secure, systems must either be designed with security features as functional requirements (security by design), or by enabling threat detection and resilience during systems' execution (security at run time). Security is thus a major concern of IoT-based systems, further emphasized by their often-critical nature.

Without loss of generality, the IoT relies on the use of numerous smart devices to build large scale intelligent and interconnected applications. Smart devices are physical objects embedded with wireless connectivity, computing capabilities, and smart behavior. The democratization of smart devices is principally driven by progress in various technological areas such as microcontrollers, low-energy wireless communication protocols, power management, etc. Consequently, smart devices are often designed using such hardware and protocols, and can be integrated into large scale frameworks through peer-to-peer or Internet-based networks.

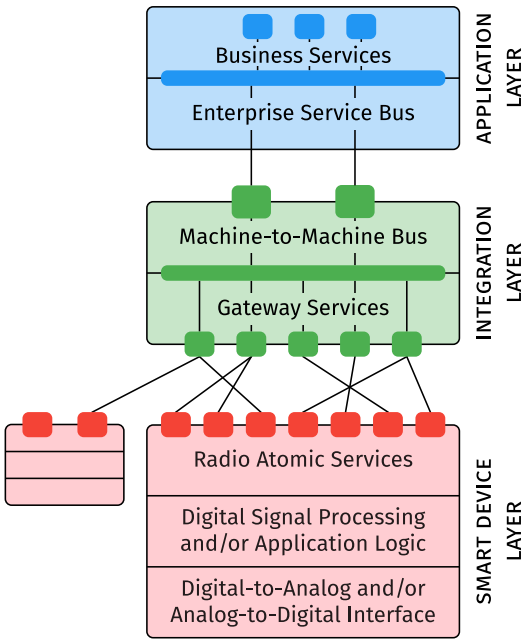


Figure 1.1 – Layer-based representation of the IoT

However, the IoT is more than a sum of smart devices interacting with each other. Indeed, it raises higher-level challenges such as its ability to scale-up, to manage the massive amount of data generated by smart devices, to perform advanced data analytics in order to make decisions and trigger actions in the physical world, etc. In addition, IoT-based systems often interact with Internet components such as Web services, Web servers, or the Cloud in order to provide end-users with interfaces to manage or retrieve information about smart devices through high-level applications.

In this dissertation, we illustrate the IoT architecture with respect to three layers, as depicted in Figure 1.1. The smart devices layer focuses on smart devices design and implementation, whereas the integration layer describes communication capabilities between smart devices and high-level software components. It also allows their integration to build wider-scale frameworks. Eventually, the application layer focuses on business logic to develop application-based IoT services. This layer is typically based on Internet technologies, in particularly, the service-oriented architectural paradigm and Web services in order to build reusable and interoperable IoT applicaitons and services.

1.2 IoT Research Challenges

Even though layer-based representations are valid models of the IoT [LXZ15], one must realize that it does not preclude IoT-based systems to be considered as a whole, and that holistic visions of the IoT improve safety, reliability and security. Indeed, each layer comes with its own set of characteristics and requirements, which must be carefully studied and understood in order to provide comprehensive solutions for IoT-based systems, but also in order to have a comprehensive vision of such systems.

As above mentioned, the entire IoT paradigm relies on the use of smart devices to interact with the physical world. These devices exhibit unconventional characteristics, that are comprehensively discussed in the remaining sections of the Introduction. Additionally, the IoT is extremely heterogeneous in terms of hardware, software (both embedded and high level) and communication protocols. This heterogeneity remains a bottleneck challenge as it prevents large-scale and system-wide integration of smart devices, and it may cause interoperability issues. Furthermore, IoT heterogeneity requires the collaboration of several research communities, namely the computer science community to tackle challenges at the higher IoT layers and the electrical engineering community to investigate challenges in the lower level layers at the edge between physical and digital world, which makes it a multidisciplinary issue. Without loss of generality, many additional communities such as signal processing, telecommunications or social and business considerations can contribute to build holistic and interoperable IoT-based systems.

1.2.1 Unconventional Characteristics of Smart Devices

The definition of smart devices is wide and encompasses extremely heterogeneous devices. For instance, an industrial robot can be considered as a smart device, but this is also the case of a wearable medical sensors. Even though those devices both fall under the smart devices, they are very different in terms of available computing capabilities, energy consumption and their potential impact on the physical world. By focusing on the most constrained cases (i.e., devices with strong resources limitations)

through the integration of resources management as a requirement, we are able to generalize IoT-based frameworks to all kinds of capable and more constrained smart devices, effectively resulting in comprehensive solutions for the design and study of IoT-based systems.

In many IoT scenarios, smart devices must simultaneously tackle the following challenges:

- *Limited resources*: Even though the scope of what can be considered a smart device is broad, and ranges from small button-cell powered sensors to continuously-powered small computers, available resources are often limited and represent a major challenge. These limitations can be observed in terms of computational power (i.e., microcontroller clock speed limited to the megahertz range), storage (i.e., flash memory not exceeding a few hundreds of kilobytes), Random Access Memory (RAM) (i.e., memory rarely surpasses a few tens of kilobytes), or battery (i.e., from few tens of mAh to a few thousands mAh). These limitations impact the devices' processing power, their autonomy, as well as their connectivity. Indeed, the processing power is correlated with the microcontroller clock frequency, and higher clock speed causes higher energy consumption. A trade-off between computing capabilities and energy consumption must thus be achieved while designing smart devices. Moreover, the processing power has also direct impact on the complexity of wireless communication protocols used by smart devices. Sophisticated Internet Protocol (IP) -based communication protocols such as TCP introduce communication overhead because of their complex message structures, and thus requiring more computational resources than non-IP communication protocols, such as Zigbee, Bluetooth Low Energy (BLE) or Near Field Communication (NFC). Consequently, resources limitation must be taken into consideration in order to improve the optimization of smart devices.
- *Hardware heterogeneity*: the wide variety of hardware used to implement smart devices remains an open research challenge. The democratization of low-cost development boards (e.g., Arduino boards, Intel Edison, Raspberry Pi) has opened the electronics development to numerous hobbyists and Do-It-Yourself (DIY) makers [Raz+16]. Even though such devices are not representative of microprocessors used in massively manufactured IoT devices, they are already heterogeneous (i.e., the Arduino Uno is based on a 8 bit Atmel AVR microcontroller, while the Intel Edison board is based on a 32 bit x86 dual core microprocessor, while the latest Raspberry Pi is built around a 64 bit Cortex-A53 core). A wide variety of programming languages and technologies are also available for low-cost development boards, ranging from the C-based Arduino programming language, to more advanced languages such as Python, Java or Node.js on Raspberry Pi and Intel Edison. The hardware heterogeneity imply challenges in terms of programming languages (e.g., object oriented programming for Java, event-driven programming for Node.js, etc.). Despite various development boards bring used in numerous DIY and research projects, they are still not suited for wide-scale production of hundred of thousands of units because of their size (i.e., the minimal size of for the Intel Edison board is $35.5 \times 25 \times 3.9$ mm) or their greedy power consumption of at least a few hundreds of milliwatts in average [DST16; TVL17; KGH14]. In general, if we consider all IoT technologies that we can use to build distributed and interconnected smart devices, the heterogeneity becomes even bigger, and

represents a major challenge for the design of IoT-based systems. It is worth noting that there is a wide variety of microcontrollers available on the market, which includes 8 bit Intel 8051 and Atmel AVR microcontrollers, 16 bit power efficient MSP430 from Texas Instruments, 32 bit Cortex Core-M*x* from Silicon Labs, STMicroelectronics, etc. Every microcontroller comes with its own characteristics and configurations parameters, which makes generalization in this field very difficult. Additionally, one should realize that microcontrollers are only one of numerous Integrated Circuits (IC) involved in the design of smart objects. Indeed, they are often equipped with application-specific peripherals such as Analog Front-Ends (AFE), which handle signal conditioning, Analog-to-Digital Converters (ADC), or Digital Signal Processors (DSP), etc. All these peripherals have their own communication protocols and configurations parameters, which increases the smart devices hardware heterogeneity. As a result, smart device designers have to chose the best hardware for their projects, while simultaneously keeping in mind the overall power consumption, processing capabilities and manufacturing cost without relying on any kind of decision-making assistance tools.

In addition to ICs heterogeneity, there is also a wide variety of hardware buses that can be implemented for cross-IC communications. Serial buses, for example, are the preferred low-level protocols, and hardware manufacturers often embed them in their ICs. This adds another hardware heterogeneity issue that must be addressed at the lowest layer of smart devices development. Smart devices designers must thus pay a particular attention at the communication protocols used by the ICs embedded in their prototypes. In sum, hardware heterogeneity is due to several causes, ranging from microcontrollers to communication buses, but also supply voltage or data format. Hardware heterogeneity thus prevents the use of common tools that could make smart devices development easier.

- *Hardware-software design methods*: The hardware heterogeneity described herein above require appropriate design methods in order to improve smart devices reliability, safety and security. Design methods (sometimes also called design methodologies) are traditionally defined as organized approaches used for system development [WR97]. They provide guidelines, methodological steps and tools to assist system development, including formal verification and validation techniques into design methods can provide additional guarantees on system properties (e.g., data type compatibility, reachability analysis, etc.), and helps to match specifications. However, design methods must be easy to use and practical, especially for smart devices development because of short time-to-market constraints. As smart devices make use of rapidly changing technologies, design methods must be generic enough in order to handle such changes without the need for extensive revisions. Reliability, safety and security requirements of smart devices can be considered by integrating appropriate logical and mathematical formalisms into the design methods, thus leading to smart devices being reliable, safe and secure by-design.

In sum, smart devices are faced with challenges and unconventional characteristics. Constrained resources, hardware heterogeneity and the lack of design methods for smart devices raise drastic challenges as how to build reliable, safe and secure smart

devices. In addition to these low-level concerns of the IoT, researchers are also faced with higher level challenges and research roadblocks.

1.2.2 Global IoT Heterogeneity

Similar to hardware heterogeneity for smart devices as defined herein-above, the IoT also faces a global heterogeneity challenge. The integration and application layers of the IoT are subject to the proliferation of protocols, frameworks, standard initiatives. This leads to heterogeneity from software (e.g., different architectural and programming paradigms, variety in the resources requirements of the frameworks, etc.) and larger-scale hardware perspectives.

Indeed, numerous high level communication protocols are developed for IoT applications. One basic way to categorize those protocols is whether they are dependent or not on the IP stack, as numerous non-IP protocol are emerging. For instance, Bluetooth, Zigbee, ANT protocols are not based on the traditional IP stack. Other examples are the ZWave or EnOcean protocols, which are both targeted at home automation applications and have been developed for integration in application-specific products. In general, the energy consumption of such protocols is low (i.e., few tens of milliwatts [Dem+13]), and their range is relatively short (i.e., up to a few tens of meters [LSS07]). Consequently, such protocols can be used to build short-range Body Area Networks (BAN) or Personal Area Networks (PAN). BANs are defined as wearable and small scale (e.g., maximum of 8 devices for BLE) computer networks, while PANs designate computer networks covering a range of a few tens of meters (i.e., at a single room scale or small building scale). It is worth noting that low-power wide-area network protocols such as LoRaWAN or Sigfox, provide a wide range connectivity up to a few tens of kilometers [RKS17] while keeping a low power consumption. Nevertheless, these protocols are not compatible with each other, and interconnectivity must be provided using dedicated proxies to handle packet translations. In addition, non IP-based protocols usually implement a full customized version of the OSI stack [Zim80] (displayed in Figure 1.2).

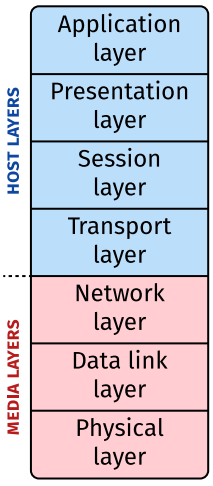


Figure 1.2 – Layer-based OSI stack

IP-based protocols are also widely available and follow the IP network layer. These protocols are typically implementing IPv4 or IPv6 over IEEE 802 -based data link and physical layers, such as 802.11 (known as Wi-Fi). Most smart devices use IP-based

protocols in order to be easily integrated in already-existing Internet-based infrastructures. Typical application layer IP-based protocols used in IoT devices are the Message Queuing Telemetry Transport (MQTT) protocol, the Constrained Application Protocol (CoAP) protocol and the Hypertext Transfer Protocol (HTTP) protocol. MQTT is a publish-subscribe protocol based on the Transmission Control Protocol (TCP) transport, while CoAP and HTTP are RESTful protocols respectively based on the User Datagram Protocol (UDP) and the TCP transport in their typical implementations. The main difference between publish-subscribe and RESTful protocols is the way data is accessed. In publish-subscribe protocols, publishers send data to subscribers asynchronously, which makes them ideal for event-driven frameworks, as all data transfers occur as messaging events. In RESTful protocols, data is accessed and modified synchronously using Uniform Resource Identifiers (URI), and they assume client-server architectures [PZL08]. Difference regarding data accesses (publisher-subscriber vs RESTful) is a source of heterogeneity, and software gateways must be implemented if both protocols coexist in real-world applications.

As a result, the global IoT heterogeneity is a major research challenge because it prevents the development of IoT-based systems at a large scale. Frameworks for the implementation of IoT-based systems must be generic, and should not rest on assumptions such as the adoption of a single communication protocol for cross smart devices interactions. Combing challenges at the high-level and the lower-level, described in the previous section, make cross-disciplinary tools, models, and methods appealing as they provide comprehensive solutions to tackle IoT-related research challenges.

1.2.3 The IoT from a Multidisciplinary Research Perspective

IoT research problems are a combination of low-level and high-level challenges related to the layer-based IoT architecture depicted in Figure 1.1. Indeed, the design and development of smart devices require the collaboration of electrical engineers, signal processing experts or even mechanical engineers (in cases where complex physical actuation is necessary). In addition, the networking of smart devices is studied by computer scientists while data analytics experts can provide insights on collected data generated by large scale IoT-based systems. They also study the integration of devices into smart services, in order to offer full-stack smart applications, solving complex real-world problems.

However, most communities consider IoT-related problems from a horizontal point of view, meaning IoT challenges are traditionally considered within a single layer. For example, electrical engineers focus on smart devices components such as System-on-Chips (SoC) or smart devices design, while computer scientists focus on smart devices networking and orchestration.

In order to simultaneously tackle both low-level and high-level IoT challenges, we promote the adoption of a multidisciplinary and holistic approach to design and implement smart devices. Indeed, the main problem caused by disciplinary partitioning is that researchers dealing with low-level layers of the IoT do not necessarily address concerns of researchers dealing with higher level layers. Typically, they deal with improvements and hardware optimizations as priorities, while system integration and services are not considered as a major concern. Nevertheless, researchers who are

working on the lower layers of the IoT have a good understanding of smart devices constrained resources (i.e., energy consumption or computational capabilities).

In opposition, researchers dealing with high-level challenges of the IoT often consider smart devices as black-boxes, which are commonly abstracted using interfaces without accurate resources modeling and understanding. They however have major contributions when it comes to smart devices integration, and the handling a massive amount of devices (i.e., in the range of hundreds or thousands of devices for a single IoT-based system).

Researchers who have a good understanding of high level technologies commonly used in the design of real-world IoT-based systems. Such technologies can be Internet-based such as cloud, fog and edge computing. Additionally, IoT-based systems become smarter if techniques such as autonomic computing and data analytics are embedded in such systems. Indeed, one of the many definitions of system intelligence is its ability to handle changing environments, and autonomic computing provides a paradigm which can be used to build self-adaptable and robust systems. Many other fields of computer science can be used in the IoT context: artificial intelligence, data analytics, distributed systems, etc. However, these fields only provide software solutions, and thus do not accurately represent the hybrid nature of the IoT.

Consequently, our research strategy attempts to combine concerns from all IoT layers in order to provide comprehensive solutions to build full-stack IoT-based systems.

1.3 Research Strategies

In our brief introduction of IoT research challenges, we identify key research challenges and directions that must be investigated to provide full-stack solutions when designing and building a smarter IoT. Our research strategy seeks to design interoperable smart devices and self-adaptable to changes.

1.3.1 Research Statement

We summarize our research problem as follows: *How to design* reusable, self-adaptable *smart devices and how to integrate these devices in smart services taking into account IoT characteristics?* Our research question includes several IoT concerns and challenges, namely through *reusability* and *self-adaptability*. Indeed, *reusability* tackles interoperability between resource-constrained smart devices and illustrates the need for better design methods targeted at the construction smart devices by considering interoperability as a design requirement. In addition, *self-adaptability* is an enabling step towards the construction of smarter devices and systems. By being able to adapt devices to environmental changes, IoT systems can operate under a wider range of execution contexts and are thus more robust, and be easily integrated to smart services.

In our research strategy, we attempt to combine advanced techniques from the computer science community and contributions from the electrical engineering community to improve lower layer interoperability and overall system integration. Our strategy seeks to:

- *Improve smart devices design*, which subsequently improves their integration into smart services. Indeed, by considering interoperability by-design, we are able to tackle hardware and software heterogeneity by emphasizing the importance of

uniform smart devices data and configuration access. This simultaneously tackle the hardware and software heterogeneity challenges, and we promote the use of computer science methods such as Service-Oriented Architectures (SOA) as means of achieving better interoperability though a multidisciplinary approach. Additionally, because we consider smart device design as a low-level problem, we are still able to capture resources limitations and constraints.

- *Provide self-adaptation mechanism to smart devices*, which improves IoT-based systems intelligence, but also their reliability and safety. Indeed, since such systems are in constant interaction with the physical world, which is in turn continually evolving, they must be robust to a wide variety of changes and operating conditions. Reliability and safety can be achieved though the consideration of self-adaptive behavior, which aims at adapting smart devices in reaction to physical alterations in order to preserve expected system behavior.

1.3.2 Improving Smart Devices Design to facilitate IoT Integration

Our first contribution is the proposition of a design method for service-oriented smart devices. Our design method aims at improving the reliability and safety of smart devices, while being generic and easy-to-use to promote its adoption by system designers.

Because smart devices are typically used in critical contexts such as healthcare or traffic control, designers must be able to provide guarantees about their devices' safety and reliability under a variety of nominal use-cases. Our design method first focuses on precise specifications of smart devices functional and Non-Functional Properties (NFP), with a particular focus on NFPs as they are the true factors of safety and reliability.

From smart device specifications, we derive a hybrid hardware-software modular architecture, which provides an overall perspective on the design of reusable and adaptable devices. This modular architecture is based on hybrid hardware-software modules, each of which is formally specified and verified, before being integrated into a full smart device specification that is also verified.

In addition, interoperability is enabled by the adoption of a service-oriented hardware description. Using this approach, we are able to abstract hardware components as technology-agnostic services, thus improving overall interoperability of smart devices. By integrating formal verification and validation as mandatory methodological steps in the design method, we provide additional guarantees on smart devices reliability and safety. We thus propose extensive testing methods based on smart devices category (i.e., sensor, actuator or gateway). Both unit testing and integration testing are provided by the method. In addition, we also introduce the concept of real-life testing on small batches of production-level prototypes in order to validate smart devices behavior in real-life conditions. This is particularly important for wearable smart devices, as the operating conditions during daily activities is very different from operating condition in laboratory settings. Finally, our design method, because it emphasizes the importance of the NFPs of smart devices, can be used to build devices with globally optimal Quality of Service (QoS).

In order to showcase the practicality of our design method, we develop a medical-grade cardiorespiratory smart sensor. Our sensor is built with an early emphasis on NFPs in the design process. Non-functional requirements include extended battery life (i.e., at least 24 hours of battery life), and medical grade accuracy and precision. The

latter characteristics are key components when developing sensor, and respectively designate the sensor ability to provide measurements that accurately represent the physical phenomenon being measured, with a controlled standard deviation. This sensor was extensively tested and characterized in accordance with the design method steps. We also developed our sensor with a particular focus on its non-functional characteristics and its ability to self-adapt with respect to both internal or external contexts. We also support interoperability by using of common and acknowledged standards (namely, BLE) so that our sensor can easily be integrated into already-existing IoT-based infrastructures.

1.3.3 Self-Adaptation: a Requirements of Smart Services

As mentioned above, self-adaptation is an enabling characteristic of full-stack IoT smart systems. Indeed, the definition of "*smart*" is wide, but we consider that the ability for a system to be able to appropriately react to external and internal changes is a first step toward global IoT-based intelligent systems. In addition, self-adaptation maximizes the use of smart devices in a wide variety of contexts and applications, making them more versatile when new use-cases are considered.

In this dissertation, we consider a vertical approach to deal self-adaptation problems. We start by bringing self-adaptability to smart devices with a focus on NFPs and QoS preservation. Practically, we propose the adoption of Labeled Transition Systems (LTS) to model smart device behaviors. This discrete model is generic enough to accurately represent smart devices heterogeneity. We also propose a rule-based language for self-adaptation objectives specification. Our main motivation for such language is the declarative nature of rule-based specifications. Indeed, the potentially massive size of IoT-based systems requires maintainability, and the fact that end-users of such systems are generally not experts compels the development of easy-to-use languages. These concerns are answered by our rule-based language, because end-users only need to specify *what* the system should achieve (in opposition to the specification of *how* to achieve such objectives in imperative languages). The combination of our rule-based language and LTS-based models of smart devices are the basis of automatic discrete controller synthesis, which are then used to enforce self-adaptive behaviors in distributed environments.

At the IoT higher layers, we propose a framework specifically addressing the dynamic nature of the IoT. Indeed, because IoT-based systems are intrinsically subject to numerous changes (e.g., changes in the physical world, dynamic addition of sensors and actuators, etc.), static self-adaptation solutions are not appropriate. Using separation of concerns between self-adaptation strategies, adaptation goals management and monitoring infrastructure dynamism, we are able to provide a comprehensive solution to the IoT-based system self-adaptation.

As a Proof-of-Concept (PoC), we developed a prototype and applied it to a healthcare case-study. We considered a wide-scale remote patient monitoring with consistent QoS which is based on a network of smart homes. We also focus on their health status, which should be robustly monitored in order to trigger appropriate medical response if deemed necessary. Our prototype features fully distributed self-adaptation policies, with the ability to deal with dynamic adaptation objectives and monitoring probes, while preserving smart devices interoperability and taking into account their limited resources.

Discrete controller synthesis was implemented using a Synchronous Programming Language (SPL), along with a basic parser to translate rule-based control objectives into synchronous control contracts.

Consequently, we present a vertical solution to IoT-based systems self-adaptation. Advantages of our contributions consist of declarative and distributed formalisms under the form of LTS-based models of smart devices, and the use of SPLs.

1.4 Dissertation Outline

The remaining of this dissertation is organized as follows: In Chapter 2, we present an extensive state of the art with respect to the challenges identified in this chapter. We thus investigate how different communities deal with IoT research problems towards design, integration and interoperability. In Chapter 3, we introduce our first contribution: a design method for service oriented smart devices.

The practicality of our design method is then illustrated in Chapter 4, where we describe its application during the design and implementation of a medical-grade smart cardiorespiratory sensor. A main component of this smart device is its ability to be integrated into wider-scale scale systems, embedded with self-adaptation capabilities, and a dynamic self-adaptation system for smart devices is described in Chapter 5. Our sensor deals with changing control objectives and monitoring infrastructure, and dynamically updates adaptation strategies in response to changes. A comprehensive implementation of our dynamic self-adaptation system is presented in Chapter 6. Eventually, we summarize the work described in this dissertation in Chapter 7, and we also discuss promising research directions related to self-adaptation, formalism and unified IoT-related modeling framework. Figure 1.3 illustrates a diagram of relationships and dependencies between chapters.

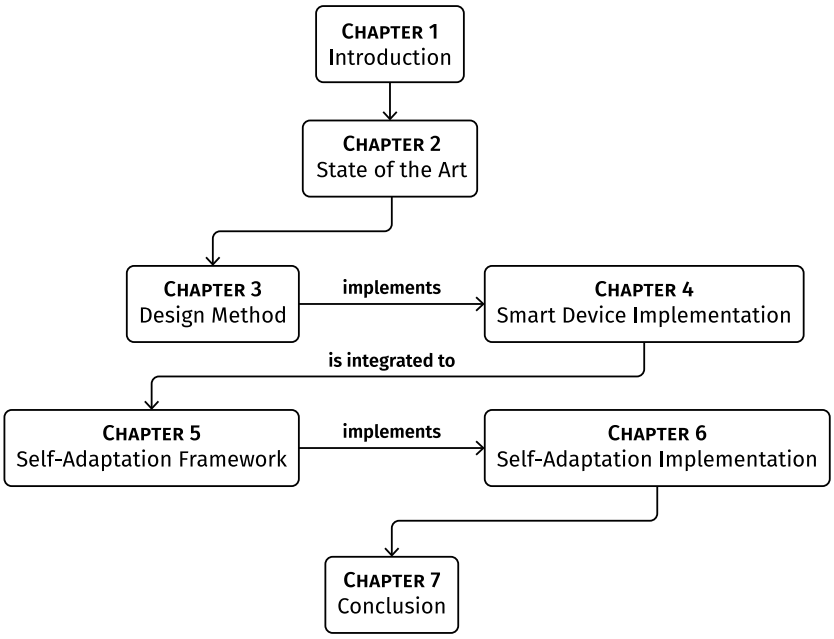


Figure 1.3 – Dissertation outline diagram

Contributions introduced in this dissertation were disseminated in the following conferences and journals, and these documents will be used and referenced in this dissertation:

- **A. Gatouillat**, Y. Badr, "Verifiable and Resource-Aware Component Model for IoT Devices." In *Proceedings of the 9th International Conference on Management of Digital EcoSystems (MEDES)*, November 2017, Bangkok, Thailand, pp. 235–242. *Invited Paper*.
- **A. Gatouillat**, Y. Badr, B. Massot, "QoS-Driven Self-Adaptation for Critical IoT-Based Systems." *Proceedings of the 2nd Workshop on Adaptive Service-oriented and Cloud Applications (ASOCA)*, November 2017, Màlaga, Spain, pp. 93–105.
- **A. Gatouillat**, B. Massot, Y. Badr, E. Sejdić and C. Gehin, "Building IoT-Enabled Wearable Medical Devices: An Application to a Wearable, Multiparametric, Cardiorespiratory Sensor." *Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC) – Volume 1: BIODEVICES*, January 2018, Funchal, Portugal, pp. 109–118.
- **A. Gatouillat**, Y. Badr, B. Massot, "Hybrid Controller Synthesis for the IoT." *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing (ACM SAC)*, April 2018, Pau, France, pp. 778–785.
- **A. Gatouillat**, B. Massot, Y. Badr, E. Sejdić, C. Gehin, "Evaluation of a real-time low-power cardiorespiratory sensor for the IoT," *40th International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, July 2018, Honolulu, USA, pp. 1–4.
- **A. Gatouillat**, Y. Badr, B. Massot, "Smart and Safe Self-Adaption of Connected Devices Based on Discrete Controllers." *IET Software*, Special Issue based on Adaptive Service-Oriented and Cloud Applications, 2018, pp 1–11.
- **A. Gatouillat**, B. Massot, Y. Badr, E. Sejdić, "Internet of Medical Things: A Review of Recent Contributions Dealing with Cyber-Physical Systems in Medicine." *IEEE Internet of Things Journal*, vol. 5, no. 5, 2018, pp. 3810–3822.

CHAPTER 2

State of the Art

Contents

2.1 Introduction 13
2.2 The IoT: Horizontal Solutions 16
2.2.1 Service-Oriented Architectures in the IoT 17
2.2.2 Model-Based and Graphical Development in the IoT 20
2.2.3 IoT Smart Devices as Cyber-Physical Systems 24
2.2.4 Summary of Horizontal Contributions 27
2.3 Designing Embedded Systems 28
2.3.1 Systems Design Methods 29
2.3.2 Hardware Solutions for Smart Devices 32
2.4 Self-Adaptation Solutions 35
2.4.1 Classical Control: Theory and Tools 35
2.4.2 Autonomic Computing: Self-Adaptive Software Systems 38
2.4.3 Reactive Systems and Discrete Controller Synthesis 42
2.5 Summary and Conclusion 46

2.1 Introduction

As mentioned in the Introduction chapter, the IoT is a broad field of study, with contributions from many research communities such as computer science, electrical engineering, signal processing, control theory, social sciences, etc. These communities have very different views of the IoT. For instance, computer scientists typically consider the IoT problems and services at the networking scale or at the application scale. More specifically, they study the networking of numerous IoT devices in order to improve global reliability, safety or energy consumption of distributed systems and services. In addition, they also study the integration of the IoT into traditional application frameworks in order to be used in real-world use cases. Nevertheless, the electrical engineering community considers the IoT from device-oriented and hardware perspectives. Typical concerns of this community are the optimization of energy consumption, networking capabilities, or processing capabilities of IoT-based smart devices. Electrical engineering IoT-related contributions are wide, ranging from Integrated Circuit (IC) design to IC integration

to Printed Circuit Board (PCB) in order to build advanced smart devices. Yet another concern of electrical engineering researchers is the investigation of design methods in order to improve hardware safety and reliability while reducing production cost and time-to-market of these devices.

In our state of the art, we thus focus on disciplinary-specific approaches to IoT problems, with a strong emphasis on horizontal contributions (i.e., contribution focusing on a single layer of the layer-based representation of the IoT, thus leading to lack of holistic approaches and technological silos). We start with a top-down approach to identify IoT-related research challenges. This approach starts with the high layer of the IoT (namely the application layer), as displayed in Figure 1.1. At this layer, computer scientists focus on smart devices integration, their coordination and collaboration in order to build real-world IoT systems. They also focus on reliability and safety characterizations of such systems to guarantee global system requirementsQuality of Service (QoS). The application layer strongly relies on purely software solutions to implement IoT solutions, which leads to a proliferation of standards, tools and techniques. Among software solutions, we have identified a few key concepts and technologies that provide interesting insights on IoT-related problems. Namely, Service-Oriented Architectures (SOA) which provides solutions for some IoT challenges. Indeed, the SOA provide an architectural style based on self-contained and loosely coupled software components [Bel08], thus improving system-wide interoperability. Another recent trend in the SOA community is the adoption of microservices architectures, as they provide improvements over traditional SOAs. In fact, microservices are defined as self-contained, functionally-limited components interacting with external tiers through messages [Dra+17]. A microservice architecture is thus characterized as a messaging-based choreography of numerous microservices. Their functionalities can consequently improve the modularity and flexibility of IoT-based systems, as they are based on self-contained, specialized, software components.

Another contribution is Model Driven Engineering (MDE), which emphasizes on the use of advanced models to improve software quality. The two fundamental contributions of MDE are the development of domain-specific modeling languages and the use of transformation or execution engines and code generators [Sch06]. Domain-specific modeling languages formally describe application requirements and behaviors, which can be validated and verified. They are typically provided as meta-models to accurately represent constraints and semantics of a specific application domain. These domain-specific models are then processed by transformation or execution engines, but also code generators, in order to provide correct-by-construction solutions for complex software problems. Contributions from the MDE community can be used for the IoT as they share common concerns, namely the validation and verification of complex systems, and beause they provide guarantees in terms of expected system behavior. However, MDE stays strongly software-oriented and lacks typical concerns of the IoT such as resources limitation and hybrid hardware/software components.

In contrast, contributions related to the low layer of the IoT architecture strongly emphasizes the modeling and simulation of hardware used in smart devices. Indeed, contributions range from ICs simulation and manufacturing to cross-IC communication modeling and simulation. Researchers also focus on the representation of hybrid systems, more particularly under the scope of Cyber-Physical Systems (CPS). CPS are defined as systems simultaneously integrating cyber (i.e., digital) and physical prop-

erties [Lee08]. The goal of the CPS community is to provide comprehensive modeling and simulation frameworks of such hybrid systems. By being able to capture both the digital and physical nature of CPS, exhaustive and quantitative system characterizations can be performed. This information can then be used not only to build accurate digital controllers for physical processes, but also to guarantee global system-wide QoS. Because smart devices are computationally augmented physical devices, they fall under the CPS definition and can be studied under this perspective. However, the scope of systems traditionally considered as CPS are relatively small, and is typically constrained to standalone single smart devices. As result, it limits the adoption of this research strategy to large IoT-based systems. Methods and tools developed by this community are however relatively theoretical, and do not emphasize on real-life smart devices design and implementations.

In addition to horizontal solutions mentioned above, we also investigated design methods for smart devices. Indeed, we consider the IoT under a practical perspective, which mandates the study of how smart devices are traditionally implemented. For that purpose, we present traditional embedded systems design methods, along with hardware solutions available for the construction of smart devices. In terms of design methods for embedded systems, numerous contributions dealing with both low level and high level aspects can be found in the literature. For instance, the hardware/software codesign community focuses on the lower layer of embedded systems, and aims at optimizing system-level objectives by simultaneously designing software and hardware components [DG97].

This field investigates compromises between extremely energy efficient, computationally powerful hardware-only implementations and sub-optimal but generic and flexible software-only solutions for digital systems. Practically, hardware/system codesign proposes a balance between Application-Specific Integrated Circuits (ASIC) implemented using Hardware Description Languages (HDL) (e.g., Verilog, VHDL, etc.) and programs running on microprocessors implemented in high-level programming languages (e.g., C, C++, etc.) [Sch10c].

Such design methods are relevant in a smart device context, as they must compromise between application-specific and energy efficient or being generic in order to increase interoperability. However, tools and techniques developed by the hardware/-software codesign community are mostly targeted at very low-level applications such as the design and implementation of application-specific ICs such as System-on-Chips (SoC). The SoC is defined as a single IC embedding several high-level hardware functions, such as a central microcontroller and several peripherals (e.g., encryption accelerators, serial buses interfaces, etc.) [Sch10b]. We also study more generic high-level system design solutions taking roots in the systems engineering community, which provide modeling frameworks for software-intensive systems.

In order to clarify the wide variety of hardware platforms available for smart devices development, we realize an extensive state of the art study of development boards, microcontrollers, or SoCs that designers traditionally use to implement their devices. As an outcome, we specify the limitations of each hardware platform in terms of scaling to mass production, ease-of-use and available resources.

We conclude with the investigation of contributions dealing with self-adaptation of smart devices. In order to accurately capture all self-adaptation solutions and see how existing contributions can be of help in the IoT context, we voluntarily kept the definition

of self-adaptation wide. We thus start with the root of self-adaptive systems: classical control. The classical control community was the first to deal with the study of system reactions to external disturbances caused by changes in the physical world. Typically, systems are integrated in closed feedback loops which integrate a tuned *controller* to provide predictable system behavior under a wide range of operating conditions [Oga10]. Classical control researchers usually focus on continuous time systems controlled by either continuous or discrete time controllers, and they use state-space system models, transfer functions and differential equations.

In the upcoming sections, we explore how classical control standard architectures can be applied to software systems, and we make a transition to contributions on self-adaptive software systems, also known as autonomic computing. Autonomic computing provides solutions for automatic system self-configuration, self-optimization, self-protection and self-healing [KC03]. Practically, autonomic elements are organized as MAPE-K feedback loops (Monitor, Analyze, Plan, Execute *over shared* Knowledge). Autonomic computing is thus the result of interactions between several MAPE-K loops. However, it does not provide a comprehensive solution to self-adaptation in the context of IoT-based systems as MAPE-K loops are often high level software components and do not capture the cyber-physical nature of smart devices. Finally, the survey of IoT-based self-adaptation leads us to the study of reactive systems, which are defined as systems that continuously react to events from the external world [HP85]. Typically, they can exhibit either synchronous or asynchronous behaviors.

The reactive systems community offers various tools such as Synchronous Programming Languages (SPL), which are considered as the basis of discrete controller synthesis. With controller synthesis, controllers for reactive discrete systems are automatically generated from system models and sets of control objectives. Such approaches are of particular interest for the IoT as smart devices tend to be strongly event-based, and must react to changes in the physical world while presenting digital (and thus discrete) interfaces to external tiers.

In the following sections, we first cover mono-disciplinary horizontal solutions for the IoT. We then consider design methods for embedded systems as they can help to improve smart devices safety, reliability and security. In addition, we explore self-adaptation contributions and see how they can play a role in the establishment of full-stack self-adaptive smart devices and IoT-based systems. Finally, we identify limitations of existing contributions with respect to our research statement on how to design reusable, self-adaptable smart devices, and show how our research strategy contribute towards these limitations.

2.2 The IoT: Horizontal Solutions

Since the apparition of the IoT, many research contributions considered IoT-related problems from a horizontal perspective (i.e., mono-disciplinary and dispersed approaches). It is worth noting that our contributions through this dissertation are at the intersection of electrical engineering and computer science. For this reason, we investigate horizontal solutions with respect to these two disciplines. More particularly, we focus on the adoption of computing paradigms such as SOAs and model-driven engineering in the context of the IoT. As a matter of fact, SOAs and model-driven engineering

are typically software-only solutions, and do not accurately deal the cyber-physical nature of smart devices. To this end, we investigate contributions related to CPS, which use hybrid models to represent both digital and physical aspects of smart devices. We however show that these approaches, while presenting solutions to some IoT problems, often lack cross-cutting considerations (i.e. the integration of concerns from other layers of the IoT stack).

2.2.1 Service-Oriented Architectures in the IoT

SOAs emerged at the beginning of the 2000s, and have been widely used by the computer science community ever since. They are defined as an architectural style providing design principles, and they are based on loosely-coupled, self-contained, technology-agnostic, reusable and contract-based *services* [Bel08]. The definition of a *service* is not trivial and can be seen under two perspectives: a business perspective (i.e., what does a service provide to potential buyers) and a technical perspective (i.e., what should be implemented in order to respect clients' requirements) [PL03]. In this section, we focus on the technical aspect of services and how they are applied to the IoT. The design principles of SOAs are particularly relevant for Web-based applications, which resulted in the rise of Web services [Alo+04]. Web services are defined as self-contained modular applications exposed through Internet-based standards [UDD01]. This definition puts a particular emphasis on respecting Web standards, which thus becomes integral parts of Web services. The success of this architectural style is due to the ability of Web services to automate software integration through a process called *composition*. Services composition simply describes the combination of Web services in order to achieve a high-level application or business logic [Alo+04]. Web services implementations are based on numerous standard technologies:

- The Extensible Markup Language (XML), which is often used as message formats cross-services;
- The Simple Object Access Protocol (SOAP), which is used as a messaging protocol;
- The Web Services Description Language (WSDL) which is typically used as a service description language, specifying Web services invocation mechanisms.

Finally, service descriptions are stored in Universal Description, Discovery, and Integration (UDDI) registries in order to be discovered and selected by external tiers.

When it comes to Web services composition, two main approaches were developed, and are illustrated in Figure 2.1 [Pel03]:

- *Services orchestration*: In this approach, Web service composition is seen as an executable process. Practically, this means services orchestration relies on remote procedure call to invoke remote services. Orchestration flow is typically specified using standards such as Business Process Execution Language for Web Services (BPEL4WS) or Business Process Model and Notation (BPMN). Web service composition is thus described as an execution flow with classical components such as sequential or parallel executions. It uses various independent services in order to achieve high-level application logic.
- *Services choreography*: This approach of cross-service interactions relies on a collaborative message-driven behavior. Essentially, services choreography is implemented through the observation of message sequences between services, that are used to trigger business-oriented actions. Message sequences can be specified

using standards such as Web Service Choreography Interface (WSCI), which associates *actions* to particular messages (or sequences of messages). It is worth noting that communications between services occur through an Enterprise Service Bus (ESB), which integrates heterogeneous services thus providing interoperability.

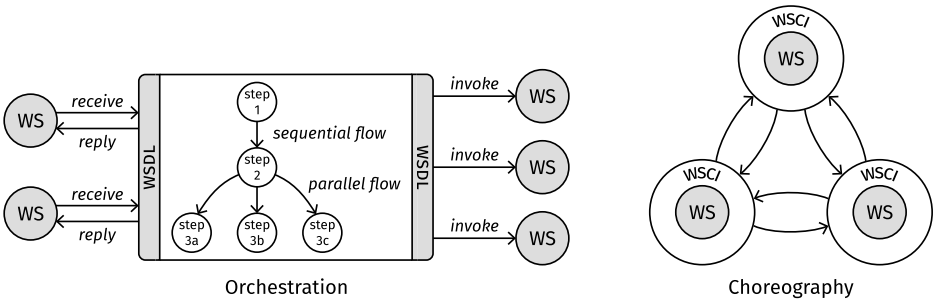


Figure 2.1 – Services orchestration vs choreography, adapted from [Pelo3]

In addition to service compositions, the SOA community focuses on problems such as Web services discovery (i.e., remotely discover services based on their functional capabilities) [Ben+05] and Web services selection (which describes the step where discovered services are filtered based on their non-functional capabilities in order to meet expected global QoS) [MS04]. However, Web service architectures based on XML, SOAP and WSDL are considered to be heavy and complex [Gar+16]. In fact, the deployment of classical SOAs based on those protocols is monolithic because of potential contract-based cross-services dependencies [CDP17].

In order to reduce the potential monolithic nature of some SOAs, microservices architectures were proposed. They are presented as an improvement over traditional SOAs [Dra+17]. In this paradigm, services are reduced to a message-based software components answering to single and simple functional concerns [Dra+17; CDP17].

Because they are message-based, microservices are preferably choreographed rather than orchestrated [CDP17]. In addition, microservices architecture typically rely on technologies such as software containers (e.g., Docker) without the need for a global and commonly-shared middleware. Microservices tend to drop XML for other languages such as JavaScript Object Notation (JSON) or Protocol buffers (also called protobuf) [SCL16] for messages, while they generally use Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST) for synchronous communication [SCL16]. In conclusion, microservices promote finer granularity than traditional SOAs, and adopt a practical approach driven by real world requirements and practices [SCL16].

SOAs offer interesting solutions to IoT-related problems. More particularly, the promotion of concepts such as loose coupling or self-containment are relevant in an IoT context, as smart devices can easily be considered as loosely coupled and self-contained entities. Additionally, SOAs can handle potential interoperability issues considering the various technologies that can be used to implement services and the variety of communication protocols utilized for communications (e.g., SOAP vs REST for synchronous protocols). This is the main motivation to the development of service-oriented IoT.

A first approach of service-oriented IoT is the use of service-oriented middleware to integrate device services. In such approach, smart device functionalities are abstracted as services [Iss+16] that interact with each other through middleware. Nu-

merous service-oriented middleware for the IoT were developed. For instance, the SOCRADES middleware proposes a smart devices integration architecture based on Web services technologies (i.e., SOAP, WSDL, etc.) [Spi+09; Gui+10]. It implements devices discovery, selection and composition. It is particularly aimed at the integration of smart devices to traditional business processes. Other Web services -inspired middleware include ubiSOAP [CRI12], which takes into account QoS during network selection, and HYDRA [ERA10], which implements ontology-based discovery. Cloud-based middleware using virtual representations of sensors were also investigated. For example, virtual sensors in [Per+14a] are implemented using a stream-based Domain-Specific Language (DSL). More advanced middleware, such as MobIoT, use ontologies and probability theory to implement devices discovery [HPI14; Iss+16]. They can be augmented with advanced graph-based composition algorithms and service buses [Iss+16]. It is worth noting that these solutions rely on smart devices orchestrations, which present drawbacks such as being monolithic and lacking flexibility. In opposition, middleware targeting large scale multi-services choreography were also developed [Vin+10; Tei+11]. In these solutions, choreography is based on distributed services buses using the Devices Profile for Web Services (DPWS) standard, which features a resources-constraints aware description of smart devices capabilities.

Another approach of service-oriented IoT is the use of service-oriented devices. In such contributions, services are integrated at the device level, and smart devices directly expose their services to external tiers. Such idea was first exposed in [dDeu+06], where authors describe the modeling of devices as services with the ability to be integrated to ESBs. With this approach, devices can directly interact with other traditional and purely software services. Authors in [dDeu+06] point out the need for devices adapters, which are provided by devices suppliers according to their work, to seamlessly integrate to generic ESBs.

However, this work remains theoretical and implementations are not provided. This is a drawback since the IoT is strongly driven by real-world practical problems. The proliferation of low-cost and easy-to-use development boards shifted research work from theoretical SOAs to service-oriented devices implementations. For instance, [Kyu+13] implement a full SOAP stack on resources constrained devices (based on a 16-bit microcontroller), and quantify resources needed for their implementation (namely, about 5 kB of Random Access Memory (RAM) and 120 kB of program memory). In addition to traditional Web services protocols, implementations of more recent RESTful or publish-subscribe protocols for constrained devices were also proposed. Typically, [Cas+11; DTD17; Raz+13] implement the Constrained Application Protocol (CoAP) RESTful protocol on various microcontrollers (please refer to Figure 2.2 for an illustration of a typical CoAP-based constrained environment IP stack). The wide adoption of this protocol in resources-constrained devices is mainly caused by its reliance on User Datagram Protocol (UDP) transport, which does not require constant connection between clients and servers (in opposition to Transmission Control Protocol (TCP) transport used in Message Queuing Telemetry Transport (MQTT)). Services choreography between CoAP-based smart devices was also proposed [BIT17], and authors define *observers* in CoAP. After registration, observers are notified of every state changes in smart devices. Please note that the list of observed streams are stored on the devices.

In conclusion, SOAs provide solutions dealing with some IoT-related challenges. Namely, challenges such as interoperability and heterogeneity are familiar topics to the

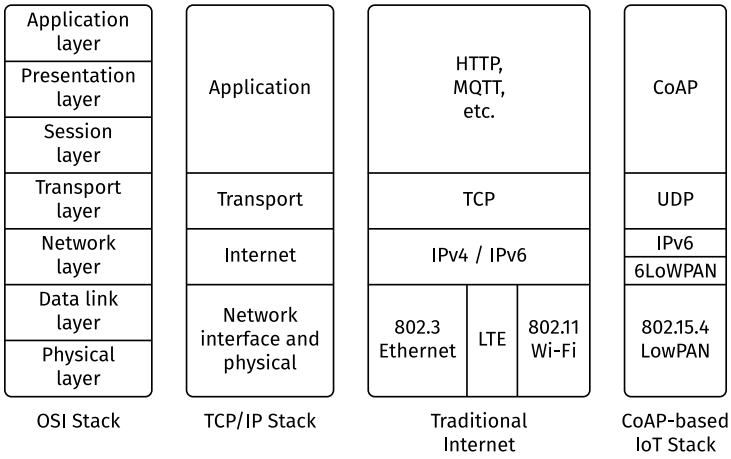


Figure 2.2 – Comparison of OSI, TCP/IP, Internet and CoAP-based stacks

SOAs community. These research problems are addressed by considering services as self-contained, loosely-coupled and contract-based entities, and can thus be applied to smart devices. Additionally, lightweight Web-based protocols such as CoAP are used in smart devices in order to provide Internet connectivity even in the lower layer of the IoT. However, SOAs are still heavily software-oriented, consequently making the integration of hybrid components challenging, especially if components are resource-constrained or based on non-IP technologies. Finally, SOAs only provide an architectural framework describing the organization of services to achieve high level application objectives, and services implementations are not particularly discussed except for the fact that they must rely on standard Web technologies. The exploration of tools targeted at safe and reliable software implementation are the concern of the MDE community, and will be detailed in the next section.

2.2.2 Model-Based and Graphical Development in the IoT

In this section, we focus on MDE and graphical programming languages, and see how they can be used in an IoT-related context. MDE aims at improving software developers productivity by promoting the use of models and code generators. This field relies on the use of meta-models in order to simultaneously represent system objectives and implementations.

Historically, MDE is the extension of Computer-Aided Software Engineering (CASE), which appeared in the 1980s as a solution to improve software quality [Sch06]. CASE uses graphical-based representation in computer systems. They are then analyzed and verified to provide insights on various system properties (e.g., complexity, non-blocking nature, etc.). Even though it was a popular research topic, it was not widely used in practical applications because of numerous limitations, such as poor code generation capabilities, poor scalability and difficulty to represent complex systems, or the lack of support for concurrent computing [Sch06], etc.

These drawbacks along with the increasing diversity of programming languages and deployment infrastructures were the main motivations for the development of MDE. Principal modeling languages supporting the MDE community include, but are not limited to, the Unified Modeling Language (UML) [Obj18e] and BPMN [Obj18a], both

initiated by Object Management Group (OMG). UML is a generic software modeling language, while BPMN focuses on the modeling of business processes [Rod15], and they both are the basis of comprehensive MDE frameworks, such as Model Driven Architecture (MDA) [Obj18c] or Eclipse Modeling Framework (EMF) [Ecl18]. MDA is a reference architecture providing a set of guidelines for model-based software specifications, and its aims at providing technology agnosticism by separating business and application logic and lower level implementation technology. EMF provides a full-fledged framework for modeling and code generation of any defined modeling language. Practically, modeling languages are specified as meta-models using the XML Metadata Interchange (XMI) standard, and the specification is used to build code generators that can then be used by end-users. The main limitation of UML stems form strongly software-oriented aspects, which make it heavy and difficult to use at its full extent. In order to tackle this limitation, and to model a wider variety of problems, Systems Modeling Language (SysML) was introduced. This language uses seven of UML diagrams, and defines additional requirements and parametric diagrams [Obj18d].

The use of modeling languages in the IoT is common, and numerous contributions promote their use in order to improve smart devices reusability. For instance, authors of [HLR17] use an IoT-dedicated SysML extension called SysML4IoT [Cos+16], which is a SysML representation of the IoT-A reference architecture [Bau+13]. In both this modeling language and architecture, smart devices and IoT-based frameworks are modeled in terms of their software, hardware and hybrid components. In particular, [HLR17] details the development of a model-based self-adaptive IoT infrastructure, which makes use of SysML4IoT models of the target system to perform automatic Java code generation, ensuring the verification of adaptation goals. In [TC16], authors define a UML profile based on the LWM2M protocol targeted at the integration of smart devices to industrial-grade IoT-based systems. Authors proceed with a similar approach as in [TC16], and use their UML profile to automatically generate code which handles the integration of IoT-ready devices in the considered industrial framework. Authors of [CS17] adopt a similar approach and propose a MDA framework, which is then utilized in combination with the Modeling and Analysis of Real-Time and Embedded systems (MARTE) UML profile to implement a smart lightning system. In addition, because MDE provides advanced tools for model checking and automatic code generation, they can be used to design mission-critical IoT-based systems [Cic+17], where systems properties such as safety and reliability must be verified both at design time and at run time.

In order to handle the proliferation of modeling languages in the IoT, initiatives such as GEMOC were developed. This initiative aims at improving heterogeneous modeling languages integration and coordination [Com+13; Bou+16]. Practically, an open-source tool called GEMOC Studio is used to represent various software-oriented or hardware-oriented modeling languages. Additionally, this tool provides a framework for the coordination and execution of heterogeneous models. When it comes to the IoT, GEMOC studio was used to model the Arduino Uno development board [GEM18]. The instantiation of such models are then used to execute Arduino programs without the need for the target physical platform. However, extensive modeling work must be performed in order to integrate systems into this framework. In our opinion, this can be a drawback in the IoT-related context considering hardware and software heterogeneity featured by smart devices and IoT-based systems.

Another challenge handled by the MDE community is the modeling of smart devices life cycle. Tools such as the PauWare engine [Bar06] use state chart XML and UML states machines in order to model and simulate devices life cycle. Such modeling languages can be used to represent software systems' reactions to internal and external events. Nevertheless, the PauWare engine is a software-oriented tool and cannot be used to precisely model software and hardware resources consumption through the object life cycle.

In the context of MDE, it is worth noting that Component-Based Software Engineering (CBSE) also contributes to the design of software using already-existing components enabled by separation of concerns [Cai+00], and it can be applied to the IoT. In fact, a component is defined as software entity implementing a set of functionalities accessed through interfaces. Many component models were defined, some of the most common being FRACTAL [Bru+06] (depicted in Figure 2.3), Palladio [BKR09], the Common Object Request Broker Architecture (CORBA) component model [Obj18b] or the Component Object Model (COM) [Mic18]. All these components share similar characteristics, such as the enclosure of their code (called *content* in Figure 2.3) in a container (described as *membrane* in Figure 2.3).

In Figure 2.3, interfaces are typed (types I, J and K) and present a client-server topology. In fact, a server interface (represented as a T-shaped sign) is responsible for the transmission of relevant data to the appropriate client interface (represented as a dashed T-shaped sign). CBSE was applied to IoT-related contexts: for instance, authors of [KHS10] have developed a component-based framework called EIToolkit in order to provide cross-device interoperability. In [AWB14; Fou+12], authors describe Kevoree and μ-Kevoree, which are dynamic component model targeted at self-adaptive systems. They demonstrated the practical use of their adaptation framework on resource-constrained Arduino boards, which proves component-based architectures can be used in smart devices with limited resources. Additionally, authors of [Rom+13] describe a component-based home-automation infrastructure. The infrastructure is implemented using the FraSCAti middleware, which uses a Service Component Architecture (SCA) with FRACTAL components. Finally, [Nai+10] used a component based approach for the integration of IoT devices to traditional Internet infrastructure.

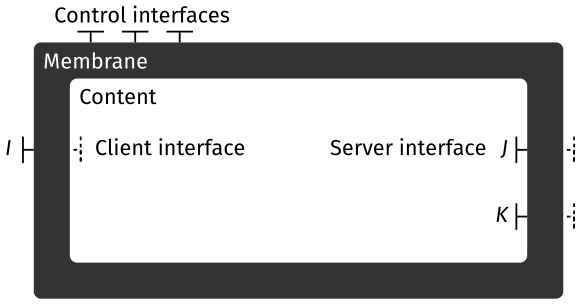


Figure 2.3 – Illustration of the FRACTAL component model, adapted from [Bru+06]

In extension to contributions of the CBSE field to IoT-related problems, we discuss the use of graphical programming languages in an IoT context. Graphical programming languages for the IoT are relatively recent, and numerous tools were developed in order to graphically generate executable code for either the coordination of smart devices, or smart device firmware execution [Ray17]. Programming the coordination or interactions

between smart devices rely on graphical flow-based languages. Node-RED [Nod18] (supported by the JS Foundation), NetLab ToolKit [vAll18] or Octoblu Flows [Oct18] are typical examples of graphical programming languages for the IoT. However, we establish a distinction between languages such as NetLab ToolKit which focuses on resources-constrained development boards such as Arduino, while Node-RED and Octoblu aim at higher level smart devices coordination. In addition, Node-RED only offers flow-based coordination of smart devices through the support of standard Internet protocols such as MQTT or HTTP, while Octoblu promotes the use of their dedicated messaging platform called Meshblu (even though they provide bindings to traditional protocols). A simple example of a Node-RED program is given in Figure 2.4. This program subscribes to two MQTT topics, on which are published JSON data containing temperature values from two sensors in the same room. The temperature is then averaged, compared to a threshold and a heating actuation is published on the relevant MQTT topic if necessary.

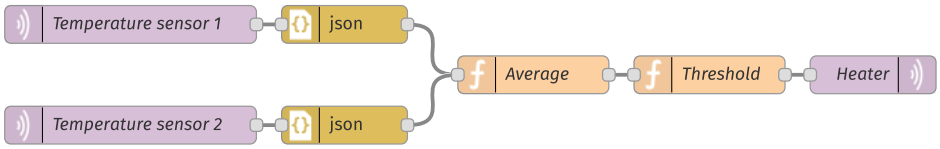


Figure 2.4 – Illustration of a simple Node-RED program

Another category of graphical programming languages undertakes a different approach by offering imperative programming capabilities for smart devices. Typical examples include ArduBlock [LH18], Modkit [Mod18] or Wyliodrin [Wyl18]. In such languages, a block-based approach is considered for the development of smart devices. Yet another interesting feature of these graphical programming languages comes from the fact they typically target lower layers of the IoT, in opposition to flow-based programming languages. For example, most of them deploy their code on Arduino development boards, and even on simple microcontrollers (for instance, Modkit supports some microcontrollers of the MSP430 family). In this block-based approach, the entire flow of execution is imperatively specified using various block structures to implement loops, branching or input/output operation.

While such initiatives are welcome as solutions for rapid prototyping or for educational purposes, they are not ready yet for large scale deployment in the industry. Indeed, graphical approaches do not easily scale because diagrammatic representation of systems can become intricate if they comprise an important amount of components. Considering IoT-based systems typically account for hundreds, or even thousands of devices, graphical languages cannot be used in this context as it would cause poor system maintainability. Another strong limitation is the limited compatibility of such languages regarding the thousands of different microcontrollers available for the implementation of smart devices. In addition, even though the graphical programming languages described above are easy to use, they are not equipped with formalism, and as a result they lack guarantees related to the verification on functional or non-functional properties of the programmed systems.

In conclusion, the study of MDE, CBSE and graphical programming languages in IoT-related context performed in this section demonstrated that they are still heavily focused on software-only architectures, and that they do not account for smart devices' limited resources and physical properties. This leads us to the study of the CPS field,

which typically aims at modeling both the digital and physical aspects of smart devices. In the following sections, we provide details on how this field contributes to the IoT.

2.2.3 IoT Smart Devices as Cyber-Physical Systems

Considering smart objects are physical objects embedded with computational and communication capabilities, they can be described as CPS. In fact, smart devices are often used in various environments to realize a range of heterogeneous applications, including telemedicine, traffic control, or smart cities [Lee10; Lee08].

In such applications, cyber systems (i.e., digital systems) are used to measure and control physical phenomenons, which results in a continuous interaction between the digital and physical worlds [Lee10]. Such systems thus fall under the CPS category. As emphasized in the Introduction chapter, smart devices are typically used in critical applications (e.g., healthcare, traffic control, etc.), and they must satisfy three properties both at design time and at run time [Raj+10]:

- Smart devices must be *robust*, *reliable* and *secure*. In particular, the continuously evolving nature of the physical world requires CPS to be able to maintain a safe behavior by appropriately reacting to physical changes and maintaining a predefined global QoS. Since CPS are typically used to regulate applications in domains where partial or total system failures can have life-threatening consequences, their security must be guaranteed. They must consequently be able to resist a variety of internal and external malicious attacks [Car+09; Wan+10].
- CPS rely on comprehensive models of hybrid systems. Considering CPS are at the intersection of the digital and physical worlds, they require models that simultaneously represent characteristics from both these worlds. In addition, the use of model-based design methodologies offers capabilities such as simulation, which improves systems reliability and safety. The difficulty of hybrid system modeling comes from the fact that both continuous and discrete behavior must be simultaneously considered in representative models. Indeed, physical systems models rely on a continuous time representations, while digital systems are typically represented using discrete time representations. CPS modeling framework must thus be able to simultaneously model both continuous and discrete time representations.
- Finally, CPS must be equipped with validation and verification mechanisms, which can be used to provide guarantees on final smart devices' functional and non-functional characteristics.

Numerous IoT-related applications can benefit from the CPS approach, and they have been used to model and verify a wide variety of systems. This is for instance the case of smart grids, which offer an extensive example of how the previously-identified CPS properties apply to real-world situations [Raj+10]. Indeed, smart grids should be able to handle a wide variety of perturbations such as power surges or power lines damages (typically caused by storms or other weather-related events). Additionally, they must continue to distribute power to consumers even with partial system failures or under extreme external disturbances. This mandates the study of global system robustness and reliability, and it can typically be achieved using advanced hybrid models such as in [LLP12]. However, modeling of such large-scale distributed systems is a challenging task, which mandates the use of experimental test beds to validate the hybrid

models [Sta+13]. Smart grids security can also be studied under a CPS perspective. For examples, authors of [SHG12] have developed a comprehensive risk evaluation framework for power distribution infrastructures, while authors of [Yil+12] used numerical attack models to detect and counteract cyber-attacks, thus protecting the power grid system.

Another example of a critical CPS can found in Internet-operated surgical robots, on which surgeons can remotely connect in order to operate on patients. For obvious reasons, designers of such systems must be able to provide guarantees in terms of reliability, robustness and security. Precise models of the robots' physical characteristics must be developed in order to design advanced command laws, which are then used to improve surgical safety. These models can also be used to study overall system robustness to a wide range of perturbations. Eventually, verification and validation can be used to prove formal system properties ensuring global system behavior.

In order to improve CPS' reliability, robustness and security, numerous design methods have been developed in the literature. Such methods heavily rely on hybrid modeling approaches to provide validation, verification and simulation frameworks that guarantee appropriate system behavior [JCL11]. In order to accurately model hybrid systems, one must start with precise continuous time systems modeling [LS17]. In such systems, designers typically use differential equations to model system dynamics since the input/output relationship of a given system can be expressed as a differential equation. Continuous time systems can then be integrated into feedback loops described by classical control theory to ensure desired system behaviors. In contrast, discrete systems can be represented using state machines [LS17]. State machine -based representations are the foundation of numerous models of hybrid systems, such as timed automata, which simply mix real-valued clocks with classical state-based automata.

Typically, the physical plant (i.e., the physical system being controlled) evolves in a continuous state space, while its controller presents a discrete behavior [ASL93]. Figure 2.5 illustrates a simple thermostat modeled as a timed automaton. A hysteresis is introduced in the system through the condition on $s(t)$ associated to transitions in order to represent a real-valued clock. In addition to this timing condition, transitions are also associated with conditions related to the temperature, $\tau(t)$. The controller then switches between two discrete states, which altogether describes a cooling or heating actuation.

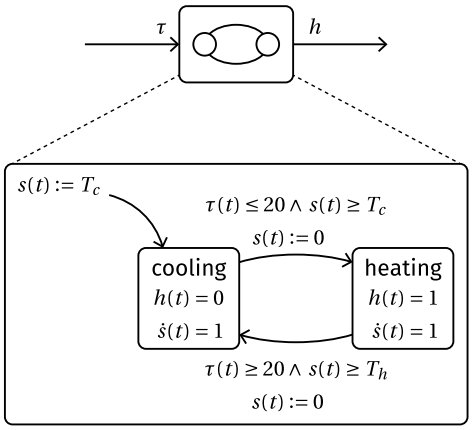


Figure 2.5 – Illustration of a timed automaton, adapted from [LS17]

Numerous tools were developed for modeling, simulating and verifying hybrid systems. Typical examples are the Ptolemy II project [Pto14], KeYmaera [Pla10] and its newer implementation KeYmaera X [Pla18], or UPPAAL [LPY97]. The Ptolemy II focuses on the design of embedded systems though modeling and simulation, particularly by the interconnection of concurrent components. This framework accounts for numerous modeling languages, ranging from continuous time models to hybrid models, along with a variety of models of computation. Because of its flexibility, the Ptolemy II framework is used to verify embedded CPS in miscellaneous application fields, such as healthcare [Sil+15] or traffic control [Bag+17].

The KeYmaera theorem prover takes another approach on CPS modeling, and promotes the use of a single language called differential dynamic logic [Pla10]. This logic specifies systems with simultaneous continuous and discrete dynamics, and the prover provides certified proofs of systems' properties. For example, the differential dynamic logic was used to prove the various safety properties of autonomous ground robots [Mit+17b] or train braking systems [Mit+17a].

UPPAAL is a toolbox for modeling and verification of real-time CPS. It features advanced model checking capabilities, which can be used to verify system properties and iteratively improve system behavior, as displayed in Figure 2.6. It uses networked time automata extended with advanced data structures (such as arrays) in order to model and analyze CPS. The UPPAAL toolbox is used in several applications such as medical devices [Jee+10; Lee+12] and aircraft landing management systems [JM12].

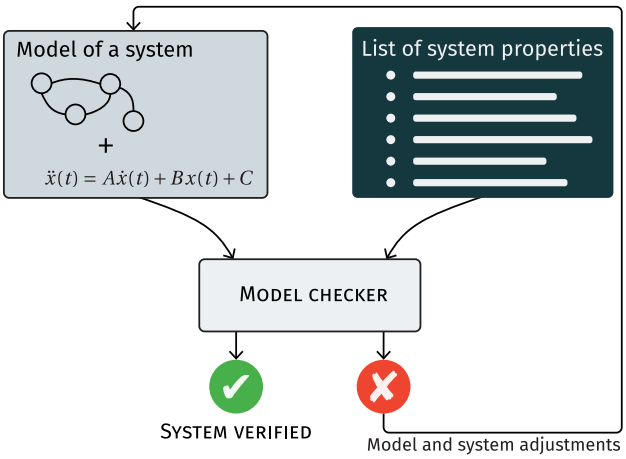


Figure 2.6 – Illustration of model checking

Even though Ptolemy II, KeYmaera and UPPAAL aim at modeling and verifying CPS, they are not equivalent, and they can provide advantages for certain situations in different ways. Typically, Ptolemy II is able to simulate a variety of heterogeneous models of computation, and is consequently a tool of choice for modeling and verifying systems with numerous computing components. In contrast, KeYmaera and its differential dynamic logic are oriented at the modeling of physical devices with advanced dynamics (i.e., moving physical devices) with some mild discrete behaviors (typically a continuous controller switching strategy). KeYmaera is thus a relevant framework for modeling of robots or any moving devices. The UPPAAL relies on advanced formalism to model hybrid systems and is thus a good fit if formal specification and validation is expected (for example, the development of industrial solutions in which safety must be proved).

These tools and frameworks have been extended and applied to IoT-based applications. For instance, Lee et al. extended Ptolemy II to build an actor-based solution for the IoT [Bro+18; LL15]. They introduce two important notions: the IoT space is referred as a *swarmlet* and *accessors*. A swarmlet represents a composition of components named *actors*. Accessors, which are a special category of actors, are seen as proxies between virtual components and implemented services or smart devices. Ptolemy II is then used as a simulation and execution environment for the coordination between *actors*. Such solution present a clear advantage over purely graphical languages such as Node-RED since advanced Ptolemy II tools ensure system validation and verification. It is worth noting that accessors generate requests attached to callbacks to perform service invocation (i.e., request-based invocation). As a consequence, they cannot be used to capture flow-based approaches of the IoT where smart devices are intelligent enough to decide when data should be sent. Indeed, flow-based approaches rely on subscription-based mechanisms instead of request-based solutions as promoted by accessors.

In conclusion, CPS provide a comprehensive theoretical framework for smart devices because they can model both physical and digital characteristics. Nevertheless, the CPS approach shows some drawbacks when higher-level problems are considered. Indeed, models developed in the CPS context lack genericity and reusability, which becomes a problem when large-scale IoT-based systems are under study. More particularly, IoT systems can use hundreds or even thousands of extremely heterogeneous devices, which makes extensive and comprehensive modeling of every smart devices an extremely time-consuming task. Additionally, CPS tools provide highly specific models of low level systems, and the modeling process must be reiterated at every system change. The lack of genericity and reusability is a strong drawback when considering a full-stack IoT-based approach because they do not provide a generic description of what is a smart object, and thus prevent higher-level analysis of entire IoT-based systems.

2.2.4 Summary of Horizontal Contributions

In summary, we focused on contributions from three communities with respect to IoT-related challenges such as smart devices reusability and interoperability along with their hybrid hardware/software nature and limited resources. In addition, we covered internal and external self-adaptation of devices with respect to the physical world. These challenges are discussed from three different perspectives as summarized in Table 2.1. Researchers who are focusing on SOAs are able to provide service-based solutions simultaneously tackling smart devices reusability, interoperability and adaptation. The use of services is particularly relevant in the context of the IoT as services are loosely coupled, self-contained, and reusable entities. This make them appealing to build large-scale and distributed IoT-based systems.

However, SOAs are typically software-centric architectures, and they do not account for resources limitation or hybrid hardware/software constraints. Similar observations can be drawn from MDE-based solutions when they are applied to IoT. Indeed, they remain limited to software development and neglect hardware aspects. Technologies such as component-based software and graphical programming languages provide interesting solutions regarding IoT-based system design since they facilitate the coordination of smart devices. Nevertheless, they present the same lacks as SOAs. In addition,

graphical approaches have the drawback of being poorly maintainable when IoT-based systems grow beyond a dozen of devices.

Conversely, CPS-based systems are modeling frameworks of choice because they can simultaneously model both digital and physical system aspects. Because physical models can be established, systems designer can simulate and verify resources consumption, which is a clear advantage given the fact that smart devices are often resources constrained. However, CPS modeling is a time-consuming task. The design of accurate physical models usually derives from a set of practical measurements of a physical phenomenon. It thus requires a deep understanding of the physical processes driving system dynamics. For example, simple electromechanical system models can be derived from various well-known continuous-time equations. Additionally, models developed for CPS lack genericity, and cannot be easily reused in different applications. These aspects are clear barriers for the development of the IoT, and urge solutions improving modularity and flexibility while taking into account software and hardware heterogeneity.

Table 2.1 – Summary of IoT challenges and solutions from SOA, MDE, CPS

IoT Challenges	SOA	MDE	CPS
Reusability	✓	✓	
Hybrid HW/SW			✓
Limited resources			✓
Interoperability	✓		
Adaptation	✓	✓	

In conclusion horizontal solutions provide interesting-but-partial answers to IoT-related challenges. Indeed, such solutions commonly lack one or several IoT-related characteristics (e.g., genericity, heterogeneity, modularity, etc.). This is our main motivation for the integration of more vertical methods and frameworks for the IoT. Improving solutions verticality' can be achieved by multidisciplinary approaches, where all the IoT layers integrate characteristics and requirements of the other layers, thus resulting in better vertical integration.

2.3 Designing Embedded Systems

Embedded systems can be defined as the embedding of microcontroller or microprocessor based computer systems into electrical or mechanical systems [Hea02]. They can range from very complex, computationally powerful and interconnected systems (e.g., avionics systems, smart cars, etc.) to very simple self-contained systems (e.g., television remote, smart temperature sensor, etc.). By this definition, embedded systems can be seen as systems controlling the physical world using digital processing and control strategies. Consequently, smart IoT devices can be seen as a sub-category of embedded systems, as they are in close relationship with the physical world (i.e., they can sense or change the physical world through sensor or actuators).

The design of smart devices can thus be studied under an embedded systems perspective. In this section, we will detail traditional embedded systems design methods and discuss their relevance with respect to smart devices design. We will also introduce traditional hardware platforms available for smart IoT devices design.

2.3.1 Systems Design Methods

In this section, we discuss systems design methods from a top-down perspective with a particular focus on embedded systems design. First, we introduce our service-oriented vision of smart devices, and we thus present design methods from the SOA domain. Then, we consider traditional embedded systems, regardless of their size, and discuss low-level software and hardware design methods. Indeed, smart devices are commonly built around a central processing unit, which coordinates interactions between its associated peripherals. These peripherals implement functions such as analog-to-digital and digital-to-analog conversions, and/or radio communication. Practically, this architectural organization can be considered as an orchestration of peripherals achieved by a central microcontroller [ZW06; TA05; Arm+05]. Because this approach is monolithic, it features reduced modularity and reusability. The microcontroller load is usually higher considering its constant orchestration all of its internal and external peripherals. There is thus a need for more modular design methods for smart devices, and we will thus position existing methods regarding their modularity. Important parameters such as formalism and the ability to describe functional and non-functional parameters on the systems are also discussed.

As reported in the previous section, SOAs are architectures of choice in terms of flexibility and modularity. For this reason, we start by exploring contributions related to the development of service-oriented systems. Numerous service-oriented design methods were developed [GL11], including SOMA [Ars+08], SOAD [ZKG04], just to mention a few. Other examples can be found in [PH06] and [CK07], where authors focus on methods for designing and implementing Web services -based SOAs [PH06] and adaptive SOAs based on the SOAD methodology [CK07]. All these methods are strongly oriented at software aspects and neglect hardware considerations. Additionally, even though authors mention the importance of non-functional requirements in the design of such system, most do not detail their practical integration within the methodological frameworks [GL11]. Furthermore, none of these methodologies integrate formalism [GL11], which is restricting for a use in often-critical IoT-based applications.

However, SOAs provide modularity and flexibility, which is particularly interesting for the IoT, and service-oriented design methods targeted specifically at the IoT were developed. For example, authors of [PA16] propose a design method for RESTful interfaces for IoT-based applications. In [Fan+14], authors introduce an ontology-based automated design method for the IoT, and they apply it to a specific IoT-based rehabilitation case study. However, the discussed ontologies are domain-specific, and this design methodology is thus confined at small-scale and application-specific IoT-based application.

Regarding general systems engineering (e.g., without the reliance on a service-oriented paradigm), several approaches have been proposed such as Harmony [Hof14; Est07], Object-Oriented Systems Engineering Method (OOSEM) [FIM07; Est07], and Rational Unified Process for Systems Engineering (RUP SE) [Rat03; Est07] just to mention a few. All these methods are not expressly directed at embedded systems development. They often rely on the use of SysML or UML, and are thus suited for the software development of embedded systems. We however particularly emphasize on the difference between modeling tools such as SysML or UML, and their use in design methods. Indeed, such tools by themselves only provide a modeling framework for systems de-

velopment, and are not associated with any set of guidelines and design steps usually found in design methods. Contributions such as [dNOW07; Mhe+14] have used SysML and UML meta-models for the development of electromechanical systems. However, all the aforementioned design methods adopt a monolithic view of systems implementation, which is a limitation in the smart devices context as they call for modularity in order to decrease development time. In addition, they commonly rely on a functional description of system goals, and non-functional properties are often not taken into account during the design and implementation phases. Authors of [JP12] propose a design method to simultaneously deal with functional and non-functional requirements by using annotated graphs (i.e., the class diagram is augmented with a series of notes describing the impact of each class on the overall QoS), but the described approach is still monolithic. A software-only vision is acceptable for wide-scale embedded systems, where computing components can abstract most of the hardware-related challenges. However, for small-scale systems such as smart devices, a lower-level approach must be also considered. Consequently, we study design methods related to the systems' hardware.

Design methods for low-level embedded systems remain a wide field of study with numerous explored approaches. One of the principal design method of embedded systems is the *hardware/software codesign* method. This method emerged in the late 1980s as an attempt to bring more automation to the development of embedded systems in order to handle increased systems complexity and the need for shorter development life cycle [Wol94; Wol03]. The principal concern of the codesign community is the optimization of functions with respect to their hardware or software implementations, also called hardware/software partitioning. The motivation behind such division comes from the fact that hardware-only implementations are commonly more energy-efficient and faster than software-only solutions [Sch10a]. Hardware implementations are however less flexible, more expensive, and less generic than software approaches [Sch10a].

Early contributions in the field of hardware/software codesign principally focused on the development of algorithms and tools, and lead to the emergence of two approaches. Namely, [GD93] consider hardware-only solutions, and incrementally add software components to decrease the development cost, while [EHB93] start with a software-only implementation and progressively add hardware components in order to increase overall system performance. In [KL93], authors describe a digital signal processing -specific Ptolemy-based design method, and emphasize the need for simulation and automatic code generation as means of reducing design time.

Other initiatives such as SystemCoDesigner [Kei+09] or [Döm+08] highlight the relevance of hardware/software system synthesis. Authors in [Kei+09] rely on SystemC models of the target systems to perform design space exploration and hardware/software synthesis. SystemC [1] is a C-like system modeling language often used in numerous design methodologies related to the electronic system-level design and verification methodology [BMP07]. This methodology is generic and targeted at the verification of low-level digital hardware systems. However, it lacks hardware synthesis capabilities, which is provided by SystemCoDesigner [Kei+09]. Authors in [Döm+08] adopt a different C-based system modeling language called SpecC. Both design methods feature a similar

1. SystemC specification: `http://www.accellera.org/downloads/standards/systemc` (visited on 06/27/2018)

life cycle, where C-based system models are used to generate transaction-level models, which are in turn used to synthesize both hardware and software. Transaction-level models designate system descriptions by which separation of concerns is performed between communicating system modules and the actual module implementation. As a result, they focus on a transactional modeling of embedded systems. This leads to hardware architectures based on several blocks communicating through various buses. In terms of hardware synthesis, both [Döm+08] and [Kei+09] rely on a register-transfer level representation of digital circuits, which models systems in terms of hardware registers and logical gates. Practical hardware implementations can then be synthesized directly from the system representation. The PeaCE Ptolemy extension [Ha+07] gives another example of a low-level design method for embedded systems, emphasizing on the simulation of generated hardware and software.

However, these design methods, even though they aim at being used for the design of any kind of embedded systems, are typically targeted at the design of ICs and SoCs. Higher-level applications of hardware/software codesign have been proposed in an IoT-related context: authors of [Bab+11] propose a design framework integrating security concerns at every design step, while [Hsi+14] describe a Field-Programmable Gate Arrays (FPGA)-based mote for wireless sensor networks. Both these methods have limitations when it comes to their use in the smart devices design. For example, the method described in [Bab+11] is extremely generic and practical implementations of the method are not discussed. Additionally, [Hsi+14] is located in the area of wireless sensor networks, which promotes the deployment of identical sensors and is consequently a poor fit considering the heterogeneity of IoT-based applications.

Furthermore, hardware/software codesign assumes that both hardware and software can be automatically generated. While this is true in the case of ICs development, it is not the case for smart devices hardware development. Indeed, the objectives of ICs manufacturers is to design and implement self-contained components that can be sold to various companies that are integrating these ICs in their design. Smart devices design and manufacturing should thus be considered, from a computer scientist perspective, as the integration of several ICs in order to meet functional and non-functional objectives. Practically, the integration of ICs is realized through a PCB, which interconnects all the electronic circuits according to a schematic specified by the smart device designers. While automation techniques such as hardware and software synthesis exist for the development of ICs, such tools are not common for ICs integration. First, while ICs manufacturers can rely on numerous hardware libraries (i.e., hardware components specified using HDLs), the variety of components available for higher-level system design represent a challenge (for instance, hundreds or even thousands of ARM Cortex-M microcontrollers can be purchased from numerous manufacturers). The proliferation of components makes their selection very difficult, which is the main reason why component selection is still typically performed by human experts. While PCB design automation tools such as autorouting exist, their use is not recommended for full PCB routing, and they should only be used to route non-critical parts of the boards [Nat17].

In conclusion, this section discusses system design methods using a top-down approach. Since flexibility and modularity are crucial requirements for smart-devices, we focused on the study of design methods applied to SOAs. Then, because SOAs are mainly software-oriented architectures, we investigated generic system design methods which can be utilized for embedded software development. We also investigated design

methods targeted at low-level hardware/software codesign. These design methods are either only dedicated to the software development or relying on hardware and software synthesis, which are not applicable to smart devices design. In fact, smart devices design relies on ICs integration rather than actual ICs design, and numerous electronic components can be used for their implementations. Such components are discussed and classified in the next section.

2.3.2 Hardware Solutions for Smart Devices

In this section, we study existing hardware solutions for implementing smart devices, and we consequently focus on technical solutions rather research contributions. We start by clarifying the vocabulary dealing with hardware peripherals typically used in the IoT context, and we then discuss their limitations or domain of application.

Recently, the proliferation of manufacturers such as Arduino [2], Raspberry Pi [3] or Beagleboard [4] offering a variety of development platforms helped to lower the costs and difficulty of electronics development. However, the variety of this ecosystem is often the source of confusion regarding their capacity, both in terms of technical characteristics and usage. A taxonomy of traditional hardware used in smart devices is given in Figure 2.7. As depicted in the figure, the four main available computing circuits are Digital Signal Processors (DSP), microcontrollers, FPGAs or microprocessors [EK16; MY11].

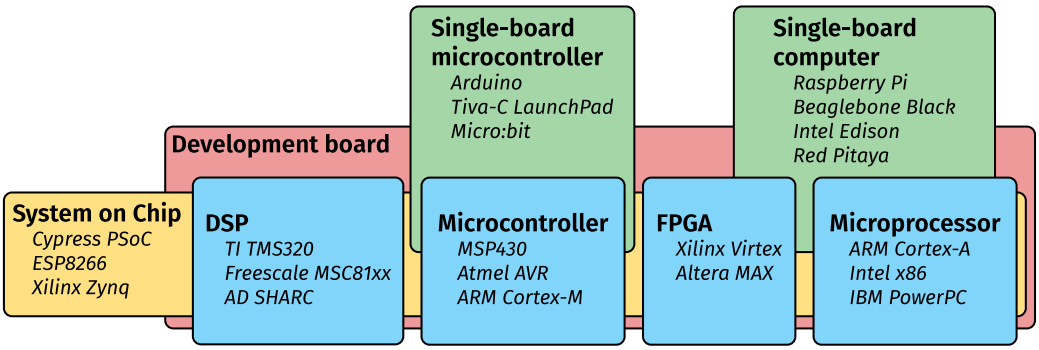


Figure 2.7 – Exemplified taxonomy of typical smart devices digital hardware

Microcontrollers are the most often used, and they designate ICs combining at least a central processing unit and a limited amount of RAM and flash memory. They also usually feature various peripherals such as timers, digital buses, or clock generation and management circuitry. Typical microcontrollers families include the MSP430 family from Texas Instruments, the AVR family from Microchip and the Cortex-M family from ARM. Microcontrollers are cheap (usually a few euros, at most a few tens of euros) and have a low power consumption (from a few milliwatts to a few hundreds of milliwatts). Nevertheless, they have reduced computing and memory capabilities. They are usually programmed using Assembly, C or more rarely C++ with development kits provided by the microcontrollers' manufacturers. Two kind of programming philosophy can be applied to microcontrollers: bare metal programming, which designates the direct programming of the hardware using low-level libraries provided by manufacturers; or

2. Arduino website: `https://www.arduino.cc/` (visited on 06/22/2018)
3. Raspberry Pi website: `https://www.raspberrypi.org/` (visited on 06/22/2018)
4. Beagleboard: `https://beagleboard.org/` (visited on 06/22/2018)

the use of a Real-Time Operating System (RTOS), such as the open source initiatives Zephyr [5], Mbed OS [6], FreeRTOS [7], or RIOT [8]). RTOS provide higher-level programming and concurrent execution capabilities to microcontrollers [Alt14; Sim15]. RTOS typically feature a small code footprint (a few tens of kilobytes), and are consequently able to run on resources-constrained microcontrollers.

Microprocessors can be used when the considered smart device does not require low energy consumption, or when more processing power is necessary. In opposition with microcontrollers, microprocessors only embed one or several central processing units, without any RAM or flash memory except low-level cache. They are however more computationally powerful than microcontrollers, and can thus be used in applications requiring advanced computational resources. Typical examples of microprocessors are the ARM Cortex-A family, the Intel x86 family or the IBM PowerPC family. They are typically integrated into full-fledged computers, and are usually operated though operating systems such as Windows, macOS or Linux. Microprocessors are however more expensive than microcontrollers, and they consume more energy, which can make them unsuitable in highly mobile applications, or for applications with limited energy availability.

DSPs can be used for signal processing -intensive applications. They are application-specific microcontrollers optimized for the operations performed in digital signal processing. Even though such ICs can be programmed using C, they are often programmed in Assembly for optimization purposes. They are used for the implementation of a wide range of functionalities related to real-time signal processing, such as filtering, compression or time-to-frequency domain conversions. The usage of DSPs in smart devices is however not as widespread as classical microcontrollers and microprocessors, mainly because advanced signal processing operations are not always required. In addition, their programming is more difficult and must usually be performed by expert programmers with extensive signal processing knowledge [Kar+09].

FPGAs are also relevant for smart devices implementation. They consist of an array of logic gates that can be programmatically interconnected to implement a desired function [MY11]. Practically, FPGAs are programmed using HDLs such as VHDL or Verilog, which are used to specify and configure logical functions. They can implement any computing function, especially in applications mandating highly parallel computations. FPGAs are however not suited for applications with limited energy supply, as they still feature high power consumptions [Vie+03; TGD14]. For examples of FPGAs, please refer to Figure 2.7.

Nowadays, hardware components are often packaged with several other peripherals in what is called a SoC. A SoC is defined as an IC which embeds one or several computing units along with several analog and digital peripherals [Fur00]. Because integration is improved through packaging of several hardware functions into a single IC, such components are characterized with smaller PCB size, energy optimization or better time to market. Typical examples of SoC are the Cypress PSoC family, where the PSoC5LP [9]

5. Zephyr website: `https://www.zephyrproject.org/` (visited on 06/26/2018)
6. Mbed website: `https://os.mbed.com/` (visited on 06/26/2018)
7. FreeRTOS website: `https://www.freertos.org/` (visited on 06/26/2018)
8. RIOT website: `https://www.riot-os.org/` (visited on 06/26/2018)
9. PSoC5 website: `http://www.cypress.com/products/32-bit-arm-cortex-m3-psoc-5lp` (visited on 09/01/2018)

embeds a microcontroller, a small FPGA and a small DSP in a single package. Other examples are the ESP8266, a popular low-cost SoC embedding a full Wi-Fi stack, or the Xilinx Zynq family, which embeds powerful FPGAs and Cortex-A microprocessors.

To improve the adoption of their hardware peripherals, manufacturers often offer development boards for all these components. Boards contains required hardware to utilize the ICs, such as power management, ICs hardware programming interface, etc. Additionally, development boards often implement a wide variety of functions showcasing the full capabilities of their main ICs. Typical examples are a set of screens, LEDs, various input mechanisms, etc. The motivation for implementing these functionalities is to demonstrate the entire spectrum of ICs capabilities. However, development boards are only for rapid prototyping and are never used in the mass production of smart devices .

We then consider single-board microcontrollers (i.e., Arduino or Micro:bit boards). Such platform have gained a lot of visibility in the recent years, partly because they are easy to use and cheap. Such boards are based on a central microcontroller (e.g., ATmega328P for Arduino Uno, SAMD21 for Arduino M0, or nRF51822 for Micro:bit), and implement all necessary hardware for power management, input/output operation or microcontroller programming. They usually do not only provide hardware, but also a full fledged development infrastructure with various software libraries and Integrated Development Environments (IDE). For instance, the Arduino ecosystem offers a development environment along with a variety of library which makes the integration of external peripherals easier. Additionally, the success of Arduino boards promoted it as a "standard" in the Do-It-Yourself (DIY) community, and numerous manufacturers now advertise on the compatibility of their platforms with the Arduino pinouts (i.e., the spatial organization of the inputs/outputs on the board).

This is further accentuated by the numerous Arduino *shields*, which are plugged on Arduino boards to provide additional functionalities (such as Wi-Fi, Bluetooth, motor drivers, etc.). Even though single-board microcontrollers have been presented as an enabler of the IoT, such as in [Sol+13; Per+14b; AlF+15], they lack the ability to scale up to billions of devices as expected by 2020 [Gar17]. Indeed, while we acknowledge that such boards are good solutions for the rapid prototyping of smart devices, they are confined to the DIY community as they do not represent nor space effective or cost effective solutions for the wide scale production of such devices. For instance, an Arduino Uno board cost Around 20 €, while the microprocessor it uses cost less than 2 €. In addition, Arduino boards embed hardware that might not be relevant for a specific implementation of a smart devices, and thus represent a suboptimal solution in terms of energy efficiency or PCB surface.

Eventually, we discuss the recent trend of single-board computers such as Raspberry Pi or Beaglebone families. Such devices embed powerful microprocessors and are powered by full-fledged operating systems, such as Linux distributions (e.g., Raspbian, Ubuntu Core, Yocto Linux, etc.), Android Things, or Windows IoT Core. They are commonly more powerful than single-board microcontrollers, as they embed powerful microprocessors (e.g., a 64-bit quad-core Broadcom BCM2837B0 for Raspberry Pi 3 B+ or a 32-bit single core TI AM3352 for the Beaglebone Black) and a large amount of RAM (from a few hundreds of megabytes to a few gigabytes). Single-board computers are flexible as their operating systems can easily changed or updated, and they can be interfaced with the physical world without difficulty thanks to a wide range of input/outputs

present on the boards. However, they do not accurately capture resources constraints of some IoT devices as they are full-fledged computers with advanced microprocessors, a large amount of RAM (in comparison with microcontrollers), and powerful operating systems. Single-board computers can however be used as low-cost gateways in an IoT context, as they often embed wireless communication capabilities and can easily be extended with additional protocols with the use of shields or USB dongles.

In conclusion, this section discussed the variety of computing components available for the implementation of smart devices. We presented the hardware-related vocabulary, and established clear boundaries between actual ICs such as microcontrollers and microprocessors, and their inclusion in single-board microcontrollers or single-board (i.e, Arduino boards or Raspberry Pis). Indeed, even though Arduino boards and similar products provide solutions for rapid prototyping of smart devices, they do not scale up to an industrial production. Even though the DIY community is extremely active, DIY devices are too complex for end-users both in their implementation and their usage. In our opinion, the IoT will be driven by industrial products, which can directly be integrated into wider-scale systems with effortless configurations and without requiring advanced technical knowledge.

2.4 Self-Adaptation Solutions

In this section, we detail contributions with respect to the self-adaptation of systems. We explore the classical control theory, which describes the adaptation of physical systems and discuss contributions related to autonomic computing. This research area focuses on the self-adaptation of software systems in order to improve their robustness and safety with respect to changes in their external environments. Eventually, we study the literature related to reactive systems, which defines a new category of systems and provides solutions for their specification and verification. Contributions dealing with reactive systems have developed interesting tools such as Discrete Controller Synthesis, which is typically used for the automatic generation of controllers ensuring predefined control objectives.

2.4.1 Classical Control: Theory and Tools

Control theory is a well-studied research area with contributions dating back to the beginning of the 20th century [Oga10]. Important elements of control theory are: *plant*, which designates a physical system to be controlled; *controlled variable*, effectively describing the physical phenomenon being acquired and controlled; *control signal*, which defines the variable being changed by the controller to meet control objectives with respect to the controlled variable; and eventually *feedback control*, designating the use of closed feedback loops as the architectural basis of controlled systems. A generic example of a feedback control loop is given in Figure 2.8. Traditionally, all components of this loop are continuous and described using transfer functions. A typical example of the use of classical closed feedback loop would be the regulation of the rotational speed of an electrical motor, where a target speed is provided as an input. The objective of closed loop control is to ensure that the desired speed is achieved. To do so, a differential equation based on electromechanical models of electrical motors is established and is used for the design of controllers. Controllers can be implemented in order to produce

a desired control signal (typically a voltage or a current) with the measured error, which is defined as the difference between the measured and desired rotational speeds.

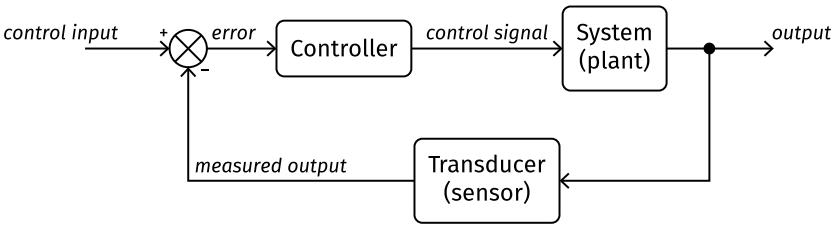


Figure 2.8 – Classical feedback loop architecture

The first class of controllers described by classical control theory are Proportional Integral Derivative (PID) controllers. Such controllers feature a continuous dynamic based on proportional, and integral and derivative components. They are consequently associated with three parameters (one for each component), which must be adjusted in order to verify the desired behavior. Typical adjustment criteria are closed-loop system stability, response time, overshoot, [Oga10] etc. Numerous techniques for PID controller tuning were developed, usually based on a set of rules such as in [Sko03] or using advanced algorithms such as in [KMS08]. It is worth noting that numerous controllers can be built on components defined in the PID architecture, such as proportional derivate (PD) or proportional integral (PI) controllers. A major limitation of such controllers stems from their static nature, meaning that they are not able to self-adapt to system changes. This is a strong drawback as systems might change for a wide variety of reasons. Such examples can be found in the aviation field, where the mass of plane vary greatly during long-haul flights because of fuel consumption.

In order to solve such problems, adaptive control was introduced and aims at providing guarantees on global behavior despite changes in the controlled system. Adaptive control is commonly implemented using Model Reference Adaptive Control (MRAC) or Model Identification Adaptive Control (MIAC). In both approaches, a reference model of the system is established, and the system is continuously monitored in order to decide if the controller needs to be adapted [Bru+09]. Other categories of adaptive control have used self-tuning feedback control architectures, such as in [CG79] or [Ath99], where systems parameters have been estimated through general purpose algorithms.

Even though such controllers were traditionally implemented using analog electronics (i.e., amplifiers, capacitors or resistors), technological advances in microcontrollers in terms of processing speed and cost made discrete controller implementations a practical reality [PNC15]. In order to implement a digitally-controlled physical system, developers typically rely on well-studied continuous-time control architectures (such as classical feedback loops) along with appropriate analog-to-digital and digital-to-analog conversions [PNC15]. Systems designer can then rely on discrete system theoretical frameworks, which provide tools such as the Z-transform, to study a wide range of system property (e.g., stability) and parameters (e.g., sampling rate of the analog-to-digital conversion).

Numerous other control methods have been proposed by the classical control community. For instance, optimal control introduces techniques such as model predictive control (which is able to handle multivariate systems which are difficult to control using simple controllers such as PID controllers) or linear quadratic regulators (which opti-

mizes system adaptation using a quadratic cost function) [Bem+02]. Please note that optimal control still requires detailed mathematical models of controlled systems, which can be challenging if systems are extremely heterogeneous. Additionally, intelligent control is introduced and relies on artificial intelligence tools such as fuzzy logic, neural networks or genetic algorithms [Ste93] in order to implement control systems. The application of such fields to the control of systems improves complex controlled systems characteristics such as robustness or reliability by the introduction of formally-derived controllers in the feedback control loop.

Traditional control infrastructures tend to be monolithic systems: even though controlled systems might be large, they are commonly driven by a single and centralized controller. The extremely distributed nature of the IoT represents a severe limitation regarding the use of classical control techniques for IoT-related applications. In order to tackle this issue, distributed control was developed, and it targets large-scale industrial systems. In fact, distributed control assumes that local regulators share partial information about the controlled systems that are then used to implement local adaptive behavior [Sca09; NM14]. In contrast to hierarchical control, distributed control does not rely on an upper control layer in charge of computing and transmitting local adaptation goals to the distributed regulators [Sca09; NM14]. Distributed control is heavily oriented towards model predictive control, principally because it features the capability to be used in large scale systems with numerous inputs and numerous outputs, in opposition to other control techniques [May14].

Traditionally, tools and methods described above are applied to the control of physical systems. However, classical control was also applied to software self-adaptation. The principal difficulty when utilizing such strategies becomes the accurate modeling of software systems within a constrained framework. Examples of software self-adaptation using classical control can be found in [Ang+16], where authors applied model predictive control for the self adaptation of a meeting scheduling software. Authors in [Pat+12] also used model predictive control for QoS and resources adaptation in software systems. More particularly, they used a non-linear model of a software flight reservation system along with a model predictive control strategy in order to ensure constant system response time under varying workloads. In [Pen+12], authors describe the use of a traditional PID controller for the self-adaptation of a course registration system. Self-tuning of controllers targeted at software systems was also explored, such as in [Fil+12] where authors used relay-based auto-tuning of a proportional integral controller for software load balancing. Contributions mentioned in this paragraph are only software-centric and do not take into consideration system physical aspects.

Classical control is used widely in smart devices. For instance, authors of [Kou+16] implement an Arduino-based connected drone using PID flight controllers. At a wider scale, [Dom+16b] describes the use of a PID-inspired network routing algorithm. Similarly, [RL15] use a feedback control loop for the control of smart power grids using Internet-related technologies. Building automation also relies on control theory, particularly in the field of heating, ventilation, and air conditioning. Numerous contributions can be found and typically rely on predictive controls (i.e., model predictive control and its variations) in order to implement intelligent temperature management systems [Thi+17]. Additionally, most continuous-time actuators need controllers in order to ensure expected output.

Eventually, researchers at the intersection of the control community and the IoT emphasize the importance of the human element in the design of IoT-based systems through the concept of *human in the loop*. In this concept, humans are considered as parts of the feedback control loop, and systems are built by considering human intents and actions [SZS15]. Humans can be integrated into every element of the feedback control loop (e.g., humans can be considered as controllers if they have direct impact on the actuations over a system, or they can be considered as sensors if they are used as a data source, etc.), and this has a direct impact on they way IoT-based systems are designed. For instance, authors of [Dom+16a] describe *soft actuations* applied in a smart home environment. In this concept, control systems only present humans suggestions of actions on their living environments (e.g., turn off the light if external sunlight can illuminate the room, close the window if the heat is turned on, etc.). Humans are thus considered as controllers in the feedback control loop. Additionally, this system can be expanded with habits-learning capabilities in order to offer personalized and more relevant suggestions. In [Guo+15], authors describe a comprehensive framework for mobile crowd sensing, where mobile phones carried by human users are used as an information source. Mobile crowd sensing is particularly useful in urban contexts, and they can be used as an information source for various advanced applications such as traffic prediction, itinerary recommendations, etc. In this framework, humans are considered as an information source and are thus representing the sensing element of the feedback control loop.

In conclusion, classical control theory provides well studied tool for the adaptation of physical systems. The objectives of control theory aim at ensuring controlled systems behave in accordance with a given control input, which designates desired functional metrics on the system outputs. Typical examples of control techniques are based on feedback loops, which can be extended to provide adaptive control, optimal control or intelligent control. Traditional control is also used in software-centric contexts to preserve overall QoS over varying execution environments.

However, techniques described above commonly rely on extended mathematical models of the controlled systems, which represent a limitation for their use in the IoT. Indeed, the IoT heterogeneity in terms of protocol or available resources makes the comprehensive modeling of IoT-based system challenging. Additionally, control techniques lack modularity as they typically define static control architectures with non-standard elements, which can be constraining for IoT-related applications considering IoT-based control systems must be able to scale up in order to handle numerous smart devices. This leads us to study the field of autonomic computing, which defines elements for the self-adaptation of software systems.

2.4.2 Autonomic Computing: Self-Adaptive Software Systems

MAPE-K feedback loops are considered as a standard reference model for self-adaptive software systems [ARS15; Vil+13; WMA10], and a graphical representation is depicted in Figure 2.9. MAPE-K stands for Monitor, Analyzer, Planner, Executor and Knowledge, as they constitute the architectural elements of the feedback loop. This reference model was initially developed by autonomic computing researchers in order to implement systems that could react to their external executing environment [KC03]. A MAPE-K loop consists of four components realizing the following actions:

- *Monitor*: To perform this action, context monitors are implemented in the system. Typically, they come as software probes measuring relevant indicators in a specific adaptive context (e.g., if the system should self-adapt to user load, the number simultaneous connection should be measured).
- *Analyze*: This action analyzes the measured context and outputs a set of adaptive actions if deemed necessary. Analyzers can be implemented in a variety of ways, from simple thresholds to complex probabilistic models. They can also be used to analyze information coming from numerous monitors in order to compute the best adaptive action with respect to a complex and multi-variable executing environment.
- *Plan*: The planning action implements the actual self-adaptive process. Practically, it defines technical ways of meeting the self-adaptation goals by specifying how the system must be changed in order to meet these goals.
- *Execute*: The execution action describes the actual deployment of the self adaptation through a set of software executors. For instance, executions can be system reconfiguration, changes in resources allocation, partial system redeployment, etc.

The four actions described above operate over shared *knowledge* (depicted in Figure 2.9), which designates system information relevant to self-adaptive strategies. For example, when self-adaptation occurs through a system reconfiguration, all possible system configurations must be known in order to provide relevant self-adaptive behavior and prevent forbidden system configurations.

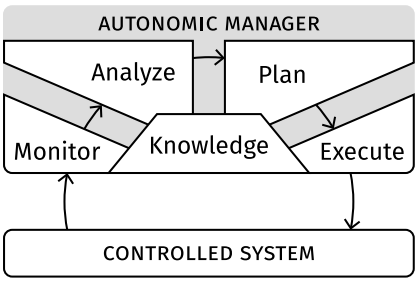


Figure 2.9 – MAPE-K feedback loop, adapted from [KC03]

MAPE-K feedback loops are particularly relevant for the IoT because of the modularity and the generic nature of their components. Indeed, MAPE-K loops are architectural models depicting the basic components of autonomic computing along with their organization, without any constraints on their actual implementations. In opposition, feedback loops from the classical control community are architecturally strict, and they define specific components architecture and implementation.

Several implementations of MAPE-K feedback loops include agent-based implementations in which components of the feedback loop are self-contained agents and interactions between agents occur through standard interfaces [ARS15]. Another implementation of MAPE-K feedback loops can be found in the FORMS framework [WMA10]. This framework introduces a formal reference model for self-adaptive software systems, and it was applied to MAPE-K loops. The FORMS structure is formally verified through the specification and verification of generic self-adaptive modules in the standard Z specification language. This framework was improved in [IW14] under the name ActivFORMS, and improvements consisted in offering model-checking of self adaptation

at run time. ActivFORMS was practically used for IoT self-adaptation in [Ift+17]. Nevertheless, these contributions propose imperative adaptation strategies specification, thus limiting their scalability and maintainability. In order to provide better scalability, declarative approaches are required because of their smaller code-base for large-scale systems, resulting in a better maintainability.

In order to handle changes in the monitoring infrastructure or in the control objectives, the DYNAMICO reference model was specified in [Vil+13]. This model applies separation of concerns between control objectives, adaptation strategies and monitoring infrastructure by implementing three interacting MAPE-K loops, as illustrated in Figure 2.10. The DYNAMICO model consequently combines the flexibility of MAPE-K loops and architectural integrity of traditional feedback control loops. Indeed, this model interconnects three independent MAPE-K loops through classical control feedback loops, thus providing an architectural solution to manage changing system objectives, changing monitoring framework and system adaptation. Nevertheless, the DYNAMICO model is typically oriented at software self-adaptive systems, and needs to be adapted in order to be used for self-adaptation of the IoT.

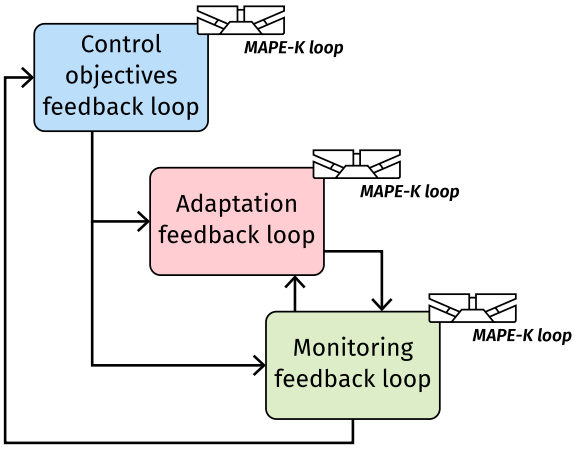


Figure 2.10 – DYNAMICO reference model, adapted from [Vil+13]

In addition, MAPE-K feedback loops are confined in the high levels of the IoT architecture, as illustrated in contributions such as [Vou+14], which uses MAPE-K components for ontology-based adaptation of flow-based social IoT. Likewise, MAPE-K derived self-adaptation in workflow-based IoT was described in [SHA17], which handle self-healing properties of traditional workflows used for the specification of IoT-based systems. Authors of [Ala+14] implement a MAPE-K based plugin for the self-adaptation of IoT architectures. A similar plugin-based integration of MAPE-K loops into semantic-based IoT frameworks is exposed in [Sey+16], and authors propose ontology-based self-adaptation (i.e., ontologies are used by every module of the MAPE-K loop in order to provide self-adaptation strategies). In [Car+17], authors describe a framework based on MAPE-K loops and Business Process Execution Language (BPEL) for the execution of self-adaptive strategies. Finally, MAPE-K feedback loops are purely-software solutions, and are consequently not appropriate for the self-adaptation of hybrid software-hardware systems. They are thus a poor fit for IoT-based systems. MAPE-K loops, even though their use is widespread, are not the only solution for self-adaptive software systems.

In order to tackle self-adaptation in a SOA context, self-adaptive service-oriented software systems were studied. In these systems, SOAs and their properties such as self-containment or loose coupling are utilized as enablers of self-adaptation in software systems. Self-adaptive service-oriented software systems were introduced in 2006 by [Den+06], and authors propose a Finite-State Machine (FSM) -based algorithm to replace defected services with other services sharing similar functionalities. In this work, adaptation is discussed both at the functional level and at the protocol level (meaning the overall SOA should be able to replace a service with another even if they do not use the same protocols). Following this preliminary work, comprehensive self-adaptive service-oriented framework were developed. Typical examples of such frameworks are SASSY (Self-Architecting Software SYtems) [Men+11] or MUSIC [Rou+09]. In SASSY, a model-driven approach is considered in order to create service-oriented self-adaptive systems. Nevertheless, this framework relies on the use of IP-based networks and technologies, which are not always available in the IoT. Indeed, non-IP networks are sometimes used for energy-saving purposes in smart devices. Similar observations can be drawn for the MUSIC self-adaptive middleware [Rou+09], which is implemented using the Open Service Gateway initiative (OSGi) service platform. OSGi is a specification of a modular service-based platform relying on Java technologies, and its architecture is given in Figure 2.11. The OSGi framework is represented by the blue and red blocks, while the other components describe parts and processes of the considered system. Because of its genericity and modularity, this framework was used in different of IoT-related projects. For instance, an OSGi-compliant middleware called HYDRA was described in [ERA10], while [GTD12] used the OSGi architecture to create an IoT platform relying on radio-frequency identification technologies. However, even though OSGi is an enabler of ubiquitous connectivity between smart devices, it relies on the Java Virtual Machine (JVM), which is not deployable in resources constrained environments.

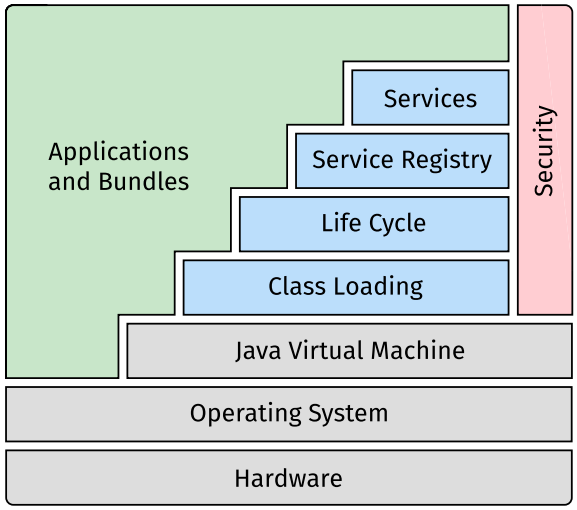


Figure 2.11 – OSGi architecture, adapted from [OSG07]

Models at run time (sometimes also called models@run.time or models@runtime) provide self-adaptation capabilities for software systems. This approach is related to the MDE community, and researchers define models@runtime as self-representing model of a given system with a particular emphasis on system's goals, behavior and structure [BBF09]. These models can then be integrated to already-existing MDE tools

to provide self-adaptive behaviors to the systems. This philosophy was used in miscellaneous contexts: for instance, authors of [Mor+09] describe a models@runtime-based architecture for dynamic software product lines through the use of complex event processing, goal-based reasoning and a configuration manager to implement their solution. Additionally, models@runtime were also used in IoT-related contexts such as wireless sensor networks, where authors of [GFA11] implemented a self-adaptive framework targeted at the reconfiguration of identical motes used in such network. Their solution is based on a MAPE-K loop, and models@runtime are used in the planning component of the loop to compute the appropriate reconfiguration of the network. However, it is worth noting that wireless sensor networks are only a subset of the IoT, and they often assume that interconnected devices present similar hardware and software architectures (i.e., a set of distributed identical devices). This assumption is not applicable for IoT-based applications since smart devices are usually heterogeneous. As discussed earlier in the state of the art, authors in [Fou+12; AWB14] proposed the models@runtime-inspired Kevoree framework to compute appropriate reconfiguration of resources-limited components. In conclusion, models@runtime are a MDE-based take on self-adaptation. Researchers using models@runtime rely on accurate models of their system in order to automatically compute appropriate self-adaptive actions. Considering models@runtime are embedded into MDE, they present same limitations with respect to the IoT, namely their software-orientation and lack of hardware modeling.

In conclusion, the autonomic computing community provides modular and flexible solutions for software system self-adaptation: the MAPE-K feedback loop, self-adaptive service-oriented systems and models@runtime. The MAPE-K loop defines the necessary components to implement self-adaptive behavior in software systems. These components are independent and loosely coupled, which makes the MAPE-K loop modular and flexible. Because such loops are only reference architectures, practical implementations are not restricted to any specific technology.

We also reviewed contributions from the MDE community to deal with self-adaptation problem, namely the models@runtime approach. In this approach, systems self-representations are used to automatically decide on self-adaptive actions. However, the contributions described above are strongly software-oriented, and usually target high-level application layers. Consequently, they are ill-suited in a resources-constrained smart devices context. Considering smart devices are in continuous interactions with the physical world, they must react to a wide variety of external events, and reactions must be integrated to self-adaptation strategies. Following this idea, we studied reactive systems and discrete controller synthesis, and their applications to the IoT as explained in the following section.

2.4.3 Reactive Systems and Discrete Controller Synthesis

Reactive systems are defined as being faced with continuous solicitations from the external world and by their need to *react* to such solicitations [HP85]. They are the opposite of transformational systems, which designate simple input/output-based systems (e.g., systems associated with a transfer function such as filters, electromechanical systems, etc.). Smart devices are systems in constant bi-directional interactions with the physical world (through the acts of sensing and actuating), and can thus be considered as reactive systems. Such systems have received a constant research effort since their

first mention three decades ago [HP85]. Researchers put a strong emphasis on the verification of reactive systems, and have proposed the use of various tools such as temporal logic [Pnu86], or SPL as exhaustive models of the systems associated with compilers acting as verification components [BB91]. Researchers dealing with reactive systems came to the conclusion that a synchronous representation of such systems presents a good theoretical framework for their formal study [BB91]. Practically, synchronous reactive systems are represented using *statecharts* [HP85; Har+90], mode-automata [MR98], or simple deterministic [MR01] or probabilistic (meaning transitions are associated with probabilities) [Seg06] Labeled Transition Systems (LTS).

A system can be defined as synchronous if the computation time of a reaction is negligible in comparison with the rate of events [Gam10], and this property is called the *synchrony hypothesis*. Practically, this means that the system must be powerful enough to compute the output between two input acquisitions. A comparative illustration of synchronous and asynchronous systems is given in Figure 2.12. In contrast to asynchronous systems, which are based on physical discrete time, synchronous systems use logical time, which means the only important timing notion in such systems is the input acquisition rate. In addition, Figure 2.12 illustrates the fact that in asynchronous systems, events can occur at any instant, and the reaction time to those events can vary. For instance, a factor which can influence the response time is the overall system load, meaning that if a system is concurrently executing several processes, the response time might be delayed if the system load is important. In opposition, in synchronous reactive systems, reaction to inputs occurs instantaneously. The synchrony hypothesis is strong constraint, but many small-sized microcontroller-based systems fall under this category, considering the microcontroller clock frequency is much higher than the sampling rate used to sense the physical world, and that microcontroller are usually dedicated at the implementation of real-time applications. It is worth noting that this hypothesis is also a good representation of the usual implementation of simple embedded systems, where the firmware typically consist of a main infinite loop that coordinates peripherals at a specific rate (e.g., [Ben12; Fer+10; WK07]). The main advantage of using synchronous systems is that they have smaller state space than asynchronous systems, which makes state space exploration for verification possible [BB91].

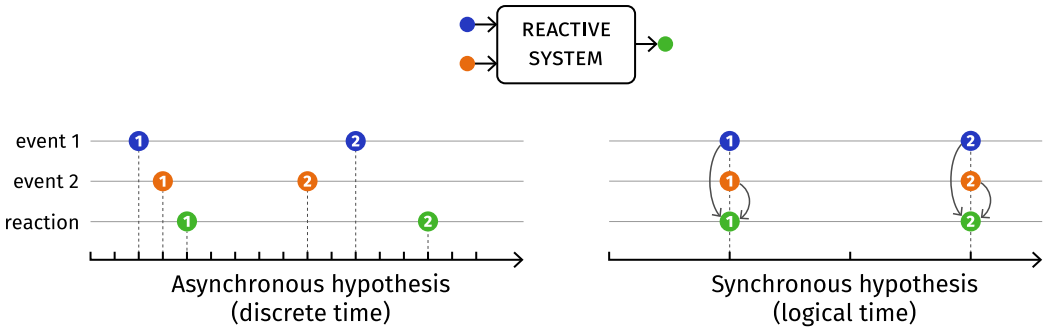


Figure 2.12 – Asynchronous vs synchronous reactive systems, adapted from [Gam10]

As specified above, synchronous systems can be represented as a variety of graphical models, such as statecharts or LTSs. However, graphical approaches feature limited maintainability (they are suitable for simple system but rapidly become extremely complex as systems grow bigger), and textual SPLs were developed. The motivating idea

behind SPLs is the ability to textually describe synchronous reactive systems in order to generate either code or executable binaries. Numerous SPLs are available, with the most common historical solutions being ESTEREL [BdS91], LUSTRE [Hal+91] or SIGNAL [BLJ91]. ESTEREL takes an imperative approach to reactive programming, while both LUSTRE and SIGNAL are dataflow languages (i.e., programs specify operations on data and connections between elements) [Hal+91; Poi+98]. The principal difference between LUSTRE and SIGNAL is that the first uses a more functional-based view of reactive programming, in opposition to the second which is constraints-based. More recent initiatives are the imperative Quartz language [SB17] integrated into the Averest [RGS13] MDE framework or Heptagon/BZR [DR10], which extends dataflow programming languages such as LUSTRE with imperative capabilities [CPP05]. All these SPLs can describe reactive synchronous systems, and generate code structures (either typical C or Java software code, or HDL code such as VHDL or Verilog). Because smart devices are traditionally designed as a microcontroller which coordinates various peripherals, it is possible to use SPLs as models for devices, and such solutions should be considered in an IoT context.

In addition to the specification of individual synchronous devices, the rigorous coordination of several devices is also challenging and can be solved using Discrete Controller Synthesis (DCS). It is related to the idea of distributed reactive systems [PR90] and more generally to the control of discrete systems [RW89]. The objectives of DCS is to use synchronous models of individual systems to generate correct-by-construction discrete controllers [Mar+00]. In other words, provided with models of a system and a set of adaptation goals (also called control objectives), DCS must compute the controller which will provide the desired closed-loop behavior. DCS was originally defined for any discrete systems [AMP95] (authors used a game-theory derived algorithm to solve the synthesis problem for discrete and timed systems), and was extended to the special case of synchronous reactive systems [Mar+00; DMR10; DRM13] (both contributions implement DCS with a tool called Sigali). The controller synthesizer Sigali is integrated with both the Signal and Heptagon/BZR SPLs. It relies on Boolean symbolic transition system encoding of LTSs to avoid the need for comprehensive state space exploration when computing the controller in order to prevent state space explosion [Mar+00; DRM13]. This Boolean representation is however constraining, as systems may need an infinite variable domain (meaning variables cannot be enumerated during state space exploration). ReaX, a tool for DCS of infinite synchronous reactive systems based on arithmetic symbolic transition system was introduced in [BM14], and is integrated to the Heptagon/BZR toolbox.

Considering smart objects in the IoT can be considered as synchronous systems typically coordinated by a central gateway, and that this gateway is powerful enough to handle the streams of events generated by the objects it controls synchronously, DCS can be used. Contributions such as [Zha+13; Zha+14; SLR17; Syl+17] use discrete controllers in home or building automation. In particular, authors of [Zha+13; Zha+14] use Heptagon/BZR for self-adaptation of home appliances, with a strong focus on functional adaptation (e.g. if a presence is detected in a room, lights are switched on, etc.) and a minor discussion of non-functional adaptation (particularly with respect to global energy consumption). Overall QoS is not considered. In addition, they also propose ontology-based hierarchical relationships between appliances, which are used to map physical properties (such as the lightning state of a room) to physical actuation (such

as the action to switch a lamp on or off). By such, the system can automatically infer the actuation of a particular physical parameter (typically, the need to actuate lamps in order to change the lightning). Authors of [SLR17; Syl+17] take a different approach to the use of DCS in an IoT-related context by integrating it into MAPE-K feedback loops for the management of smart buildings. In summary, analysis and planning blocks are replaced by discrete controllers, and the implementation relies on LINC (an IoT-oriented transactional middleware), and PUTUTU (a device abstraction framework). These contributions also discuss the implementation of systems handling changing adaptation goals using a controller switching-based approach. Controllers are replaced by others if changes in the control objectives mandating a controller adjustment are detected by the system. The coordination and composition of several MAPE-K loops is also discussed, and is presented as a solution for the up-scaling of the control system in response to an increase in controlled devices number. Similar to authors of [Zha+13; Zha+14], authors of [SLR17; Syl+17] only consider functional properties and QoS preservation is not discussed. Please note that adaptation goals in Heptagon/BZR are contract-based, and are expressed though a series of declarative first-order logic statements specifying desired global system behavior. In addition, DCS in Heptagon/BZR requires the distinction between controllable and non-controllable transitions in LTSs. Practically, controllable transitions can be triggered by the external controller in order to achieve adaptive behavior, while non-controllable transitions describe internal synchronous module behaviors. DCS thus computes actuation on controllable transitions to guarantee the verification of controlled objectives. Event-Condition-Action (ECA) rules-based DCS was also explored through the translation of a ECA rules-based high-level to BZR contracts in order to perform controller synthesis [CDR14; Can+14].

Even though the synchronous hypothesis provides a theoretical framework for the study of reactive systems, it is not suitable for all systems falling under that category, and some systems must be studied with an asynchronous perspective. Synthesis of asynchronous finite reactive modules is presented in [PR89a], and automated reasoning of temporal logic can be used to verify such systems [Eme96]. Advanced toolboxes such as CADP [Gar+13] were developed, along with tools such as LNT [GLS17], a high level specification language for distributed system. Capabilities of the CADP toolbox are wide, as it offers distributed model checking of concurrent asynchronous systems. CADP was used in combination with LNT for DCS of asynchronous systems in [ASD17] and applied to the coordination of autonomic managers.

In this section, we demonstrated that smart devices can be modeled as reactive systems, as they continuously interact with the physical world. The reactive systems community provides formal tools to both verify and synthesize such systems, but also to coordinate several reactive systems. Because synchronous reactive systems are more studied than asynchronous ones, and because advanced tools such as SPLs are available for the specification and design of such systems, we promote solutions which verify the synchrony hypothesis when developing smart devices. In addition, SPLs and DCS provide a comprehensive formal framework which guarantees global system behavior, which is necessary considering the critical nature of most IoT-based systems.

2.5 Summary and Conclusion

In this chapter, we presented a broad view of the fields, tools and techniques from various communities related to the design and adaptation of smart devices in an IoT context. First, we explored IoT horizontal contributions from the computer science and electrical engineering communities. In particular, we demonstrated that SOAs, which organize software systems around services, provide promising solutions to handle IoT challenges related to modularity and interoperability. Indeed, because SOAs emphasize on the use of modular, self-contained and loosely coupled services, they can improve system-wide interoperability even when faced with heterogeneous technologies. However, SOAs are purely software solutions, which do not take into consideration resources limitation and physical characteristics of smart devices.

Additionally, contributions from the MDEs community have shed light on the IoT development lifecycle. Indeed, the critical nature of IoT-based system mandates the presence of formalism in their development life cycle in order to provide guarantees regarding their functional and non-functional properties. MDE proposes modeling languages along with analytical and execution frameworks which can help producing such guarantees. Furthermore, the formal definition of software-oriented components by the CBSE community improves smart devices flexibility and modularity as they can be seen as a set of interacting and self-contained software entities. The main difference between *components* defined by CBSE and *services* described in SOAs lies in the fact that components deal with implementations, whereas services deal with higher-level and technology-agnostic specifications.

To conclude our review of horizontal solutions for the IoT, we explored the field of CPS and studied hybrid physical and digital systems through modeling, simulation and verification. To achieve such objectives, advanced hybrid modeling languages were developed, focusing on low-level characteristics of smart devices. Such model can then be simulated in order to verify various digital and physical system properties. However, physical modeling is time-consuming because it commonly relies on practical experimentation for the computation of accurate models. This is a strong limitation when we attempt to scale them by considering hundreds or even thousands of devices.

In sum, horizontal solutions are not able to provide comprehensive solutions to the IoT-related challenges defined in the Introduction chapter. This motivates our vision of a vertical approach to IoT-based systems, by which we can improve integration between different IoT layers. In the remaining of this dissertation, we thus adopt a bottom-up perspective to build our solutions for adaptable and reusable smart devices. This is why we start with the hardware layer to design and build devices with strong constraints on their functional and non-functional requirements. Indeed, smart devices are usually resources constrained, and their hybrid nature combined with their connectivity represent a design challenge, which can benefit from a service-oriented approach. We thus attempt to apply service-oriented concepts to the design of smart devices through new design principles.

Contributions related to systems design methods were also explored. We demonstrated that even though they provide insights for the design of modular and flexible systems, SOAs design methods are heavily software-oriented, and do not capture accurately the hybrid nature of smart devices. We also focused on generic systems design methods based on tools such as UML or SysML because they can be used for embedded

systems software development. However, these methods are targeted at monolithic systems, which do not support the modular and flexible nature of smart devices. Eventually, methods targeted as low-level hardware/software codesign were investigated. Such methods provide tools and frameworks for the specification, simulation and implementation of hybrid embedded systems. Nevertheless, they are oriented at the design and manufacturing of ICs, and are consequently not suited at smart devices development. Indeed, the design of smart devices typically focuses on the integration of several ICs rather than the development of application-specific SoCs. We consequently demonstrated that there are opportunities for the development of a new design methodology focusing on ICs integration using a service oriented perspective. By such, we aim at providing both formal guarantees on final smart devices behavior while preserving acceptable modularity and flexibility.

We then continue our state of the art with a bottom-up approach to self-adaptation of smart devices and of the systems built around them. In this context, we consider that self-adaptation is an enabler of vertical system intelligence, and that the construction of smart services should rely on self-adaptive devices. This motivated our study of self-adaptation strategies and classical control theory, which hinges on the adaptation of physical systems using continuous-time models and controllers. However, most classical control solutions rely on detailed models of physical systems, which are not always available as the design of such models represent a time-consuming task due to their experimental nature. We then discussed contributions from the autonomic computing community, which defines a standard architecture for feedback control loops known as MAPE-K. This architecture is generic, and can be adapted to a wild variety of use cases, as proved by the numerous practical implementations detailed above. The genericity of MAPE-K feedback loops can then be combined with formal methods to provide feedback control with guarantees on adaptation objectives. We thus focused on formal methods for the control of discrete systems, and discussed contributions from the reactive systems community. These systems are in continuous interaction with their external environment and are consequently appealing for the representation of adaptable smart devices. Formal models of reactive systems can then be used to automatically synthesize discrete controllers, in order to ensure the verification of their adaptation goals. We consequently chose to combine the versatility of MAPE-K feedback loops with the formal guarantees provided by discrete controller synthesizers.

Since our contributions consider a service-oriented approach as low as the device layer of the IoT, we are able to improve smart devices interoperability while considering limited resources and hardware/software aspects with an appropriate smart device design method. In addition, we propose to embed internal and external self-adaptive behaviors in smart devices, as self-adaptation is a step towards building smart services, with a particular emphasis on QoS preservation because of the often-critical nature of IoT-based systems. In order to provide formal guarantees about the verification of systems adaptation goals, we rely on theoretical and practical tools developed by the reactive systems community.

CHAPTER 3

Design Method for Service Oriented Smart Devices

Contents

3.1 Introduction 49
3.2 Method Overview 52
3.2.1 Generalities 52
3.2.2 Smart Devices Ontology 54
3.3 Comprehensive Method Description 56
3.3.1 Smart Device Requirements Analysis 58
3.3.2 Smart Device Constraints Analysis 58
3.3.3 Modular Architecture Design 59
3.3.4 Modules Formal Specification 60
3.3.5 Smart Component Formal Specification and Verification 60
3.3.6 Modules Implementation and Integration 61
3.3.7 Smart Device Testing 61
3.3.8 Smart Device Production 62
3.3.9 Adding Self-Adaptive Behavior as a System-Wide Requirement . 62
3.4 Medical Smart Device Design 63
3.4.1 Smart Device Analysis 63
3.4.2 Smart Device Specification 65
3.4.3 Smart Device Implementation: an Overview 67
3.5 Summary and Conclusion 69

3.1 Introduction

Smart devices are building blocks of the IoT, and all IoT-based systems rely on the use of such devices to interact with the physical world. They cover a wide spectrum of sensors, actuators, robots, unmanned vehicles or even wearable devices, which come with specific requirements and characteristics, namely:

- *Limited resources*: even though smart devices are diverse in nature, they are usually resources constrained. More specifically, microcontrollers typically found in

smart devices only feature minimal amounts of Random Access Memory (RAM) and ROM (i.e., from a few tens of kilobytes to a few megabytes), but they also have reduced clock speed (usually a few tens of megahertz), which limits their computational power. The use of resources-constrained hardware is driven by energy savings motivations (less powerful microcontrollers are less power-hungry) as well as cost saving reasons since constrained microcontrollers tend to be cheaper than more powerful solutions. Furthermore, smart devices are often battery-powered, and a compromise must be achieved between computational power, size of the device, and battery life.

- *Hardware and software heterogeneity*: The IoT ecosystem is broad, and a wide variety of protocols can be used for the implementation of smart devices. For instance, they can use various communication protocols such as Wi-Fi, Bluetooth Low Energy (BLE), NFC, Zigbee, implemented on a variety of microcontrollers (e.g., controllers from the MSP430 or ARM Cortex-M families, Intel 8051-based System-on-Chips (SoC), etc.). All these components come with their own programming tools and development environment (usually provided by the manufacturers), and they are not inter-compatible. Additionally, regardless recent standardization efforts, IoT standards remain highly fragmented, which leads to divergence in vocabularies, methods and models (e.g., OneM2M, IoT reference architecture, etc.) [Bau+13; Sic+15].

As a result, the design of smart devices must simultaneously account for all these specific requirements and characteristics, with an additional emphasis on the development of flexible and modular solutions, promoting the reusability of devices architectures and capacities in a wide variety of applications. Furthermore, the critical nature of most IoT-based system requires the integration of formal verification and validation methods and techniques within their design lifecycle, in order to be able to provide guarantees on final smart devices behavior.

When it comes to interoperability challenges, Service-Oriented Architectures (SOA) provide solutions by using self-contained, loosely couple and technology-agnostic *services*. Indeed, SOAs provide a theoretical framework and design concepts for the implementation of distributed software systems. They tackle interoperability issues by promoting the use of standard technologies such as Web services -related standards and the use of *adaptors* to integrate legacy applications [PvdH07] to service-oriented systems. It turns out that the service model is a good fit for smart devices, as several similarities can be established between them, such as their self-contained and independent nature, but also the fact that their integration to wider scale framework should not depend on their implementation (i.e., technology agnostic integration).

However, even if some similarities can be drawn between SOAs and smart devices, the former can not directly be applied to the latter because of a variety of reasons. First, traditional SOAs design methods such as SOMA or SOUP [Est07] adopt top-down design lifecycles and are not able to accurately model and take into account smart devices limited resources and hardware constrains. Additionally, such methods are often heavily software-oriented, and hardware is considered as a black box providing all needed tasks through low level Application Programming Interfaces (APIs). Finally, SOAs usually rely on heavy and verbose protocols such as Web Services Description Language (WSDL), and require a full TCP/IP stack, which is not always available for resources-constrained smart devices.

Conversely, lower-level design methods such as hardware/software codesign can accurately model, simulate and synthesize hybrid systems. Methods developed in this context, such as [Kei+09] or [Döm+08], offer a comprehensive design lifecycle for the creation of hardware/software systems. However, these methods hypothesize that both hardware and software can be automatically generated. If this is the case for Integrated Circuits (IC) development, the hypothesis does not hold in the context of smart devices' design. Indeed, smart devices are commonly implemented through the integration of several ICs on a single Printed Circuit Board (PCB), where automation is not widespread and man-crafted design is still preferred by system creators rather than the use assisted tools and automation processes.

In order to simultaneously tackle limitations of design methods defined for both hardware/software codesign and traditional SOAs, we propose a new design method with key concepts originating from SOAs and applied to the development of hardware/software smart devices. Our design method emphasizes on the need for better interoperability, while still being applicable in resources-constrained smart devices in order to improve their integration to wide-scale IoT-based infrastructures. Consequently, we propose a comprehensive SOA-based design method to build service-oriented smart devices. This method brings all the key concepts of SOAs at the smart device level in terms of modularity, interoperability and reusability; while preserving accurate representations of smart devices' limited resources and conditions of operations. The outcome of this method leads to the design and manufacturing of interoperable smart devices, which expose low-level hardware-software services (that we also call hardware services). Our method seeks to improve smart devices' integration into a wide variety of IoT-based applications through middleware, and their aggregation with businesses services for the construction of smart services.

More specifically, our design method for service-oriented smart devices comprises methodological steps to:

1. Build an internal modular architecture by decoupling hardware/software components into self-contained modules with well-defined interfaces. By such, each module can be easily developed and tested separately or later substituted without the need for a complete architectural redesign.
2. Apply the service concept to smart devices in order to expose their functionalities as hardware-level services, ensuring external modularity and interoperability, or to manage their constrained resources and capabilities.
3. Apply a V-shaped development lifecycle by which testing activities like requirement analysis, design tests and system integration are validated and corrected well before hardware implementation and software programming. This ensures a higher chance of design success over waterfall lifecycle models, which are often used in SOA design methods and hardware/software codesign methods.
4. Integrate formal hardware and software specifications at the design phase, followed by validation and verification through model checking to ensure desired smart devices properties (i.e., safety and liveness).
5. Refer to a full-fledged ontology for smart devices in order to share common concepts among all participants from different disciplines in the design process. Practically, shared ontology instances are used when expressing functional requirements, Quality of Service (QoS) constraints, or verification and validation

statements. By such, ontologies reduce ambiguity and ensures global consistency at each phase of our design method.

The remaining of this chapter is organized as follows: Section 3.2 gives a broad overview of guidelines and methodological concepts used in our design method, but also introduces a smart devices ontology which is then instantiated and used in the remaining methodological steps. Section 3.3 generically details each phase our method, which is then applied to a specific medical smart device use case in section 3.4. Finally, we conclude our work in Section 3.5, and we identify limitations and possible research directions.

3.2 Method Overview

In this section, we further present our motivations for the foundation of our design method. Additionally, we introduce simple ontologies that can be used within our method to assist smart devices designers with some methodological steps. We also introduce a set of guidelines supporting the main idea behind our design method, namely hardware/software module compartmentalization for better smart devices modularity and flexibility.

3.2.1 Generalities

The rise of the IoT and the democratization of various boards, sensors and actuators available in the market are a strong driver of the Do-It-Yourself (DIY) movement. As mentioned in the state of the art, single-board microcontrollers and computers such as Arduino boards or Raspberry Pis represent low-cost options for the development of smart devices by hobbyist developers. Nevertheless, smart devices development based on such boards is empirical, and lacks formalism ensuring devices' safety and reliability. Additionally, the DIY approach to smart devices design does not scale to mass production (i.e., manufacturers usually manufacture thousands of devices), neither is suitable for industrial deployment. As a result, the design of smart devices should be subject to a rigorous design method, accounting for both functional and non-functional requirements, along with constraints on limited resources, formal specification and verification, and eventually hardware and software testing. These methodological steps are preliminaries to the mass production of smart devices and help to provide guarantees on smart devices behavior in a wide variety of applications.

To meet all these requirements, the design of smart devices (e.g., medical sensors or actuators, home automation devices, etc.) comprises two stages. The first stage deals with smart devices prototyping, which relies on the use of various SoCs and ICs, and defines verifiable hardware-software services. The second stage is defined with respect to the mass production smart devices, which are in fact PCBs mounted with ICs organized into assembly lines.

Our design method implements these two stages, and introduces a V-shaped design and development life-cycle. It also advocates that different hardware and software components should be organized as a choreography in order to provide a service-oriented modular hybrid architecture. We define different architectural hardware elements as a

tuple:

$$\text{HW architecture} = < \text{DAC}, \text{ADC}, \mu\text{C}, \text{DSP}, \text{Radio}, \text{Memory}, \text{Energy}, \text{Bus} > \quad (3.1)$$

where DAC designates Digital-to-Analog Converters (DAC), which convert digital domain information to physical domain actions, while ADC labels Analog-to-Digital Converters (ADC), which provide the opposite conversion. DSP and μC respectively represent dedicated low-level digital signal processors and general purpose microcontrollers. The Radio element of the tuple designates communication-oriented SoCs, while the Memory element defines storage-dedicated components. The Energy member of the tuple is used to designate the energy capabilities of the device, and finally the Bus element defines the hardware buses used by different ICs to communicate.

Each component corresponds to a different hardware entity, and they all are considered as independent and self-contained entities with potential hardware/software interfaces. Several software components can be deployed on each of these hardware parts, resulting in hybrid hardware/software entities, which we refer as modules in the remaining of this chapter. As mentioned herein above, traditional design methods see microcontrollers as central parts of the devices, where they act as orchestrators between the hardware components.

From a software point of view, the architectural elements can be defined as a tuple:

$$\text{SW architecture} = < \text{firmware}, \text{bootstrap}, \text{interrupts} > \quad (3.2)$$

In order to ensure an internal modular hardware architecture, we provide a set of guidelines and best practices, decoupling hardware/software components into self-contained modules with well-defined hardware/software interfaces:

1. *Decouple hardware and software modules* in order to decrease the processing load of the microcontroller by implementing each of the software functions on dedicated components (i.e., signal processing functions implemented on a DSP, radio communication function on a radio-oriented co-processor, etc.)
2. *Use Direct Memory Access (DMA) and interrupts to communicate between components* as a mean for the microcontroller to stay in a sleep (and thus low-power) state during communication phases between digital components, consequently optimizing energy consumption.
3. *Serialize data exchanges.* With data serialization, the exchanged data becomes self-descriptive and causes better self-containment of each of the hardware components. Binary data serialization (such as MessagePack, CBOR or BSON) is preferred because of its relative compactness when compared to plain-text serialization.
4. *Specify hardware services in the radio component.* Hardware services expose smart device functionalities to the external world as independent and reusable services. They describe functional properties, internal resource management or on-the-fly configuration of hardware modules during execution. It is worth to mention that hardware services are exposed through radio communication to external smart devices:

$$\text{HW services} = < \text{actuation service}, \text{sensing service}, \text{resources mgmt.} > \quad (3.3)$$

5. *Enabling security in the radio component*: Security remains a major challenge in IoT, but many hardware cryptographic accelerators for data encryption and decryption are available as standalone or integrated hardware peripherals. Since we recommend the radio component to provide hardware functionalities as services (hardware-as-service), it should ideally handle data encryption/decryption with a dedicated accelerator. In this context, the main microcontroller does not have to implement software-based cryptographic functions, which thus reduce active time and energy consumption. Additionally, it is worth noting that some microcontrollers such as Cortex-M*x* also embed cryptographic accelerators, and that data can consequently be encrypted end-to-end.

3.2.2 Smart Devices Ontology

The design of smart devices requires the collaboration of designers with various skills and backgrounds, which leads to a complex hardware and software design process. Furthermore, the complex hardware and software design process introduces a rich vocabulary along with various concepts, which can be ambiguous and where misinterpretation can occur at any step of the methodological lifecycle. As a solution to this problem, we specify a comprehensive-yet-simple ontology for smart devices, which can be used as a shared knowledge source between all designers and developers involved in smart devices design. When implementing a specific smart device, the ontology can be instanced using the device's type (i.e., sensor or actuator) along with its resources. This instance is particularly useful to express functional and non-functional requirements (i.e., QoS requirements), and can be used to decide which properties should go through the validation and verification process. Consequently, we expect our ontology to reduce ambiguity between systems designers, and to improve global consistency in all steps of the design method.

Our ontology is given in Figure 3.1. It divides smart devices categories into three sections, which are not mutually exclusive (i.e., a smart device can belong to several categories):

- *Sensor*: A sensor is a device that converts a physical phenomenon to a digital representation. Commonly, physical phenomena are transduced to electrical properties (e.g., voltage, current, impedance), which are subsequently converted to digital representations though appropriate analog signal conditioning and ADCs. Our ontology details various properties of sensors as illustrated in Figure 3.1. The closest property from the physical world is the unit of the physical phenomenon being measured by the sensor, as depicted in the ontology. Getting closer to higher-level services, the sampling rate of a given sensor is another critical point with impacts on the smart devices modular architecture, as higher sampling rates require more storage and processing power. The output property represents the data format of sensor measurement (i.e., number of bits, signed vs unsigned, etc.). Sensor management and behavior can then be described using one or a combination of the following properties: real time, event-based or query-based. A real-time sensor simply sends measurements at a predefined regular rate, while an event-based sensor only sends relevant information under some predefined conditions derived from the sensor physical context. A query-based sensor only transmits measurements if requested by an external tier. Eventually,

the sensor is accessed through a single access point represented by the sensing service.

- *Actuator*: An actuator is the functional opposite of a sensor as it transforms information from the digital domain to the physical world through actuation. The highest-level elements of actuators take the form of actuation services, which represent actuation interfaces to the external world. Then, the input property in our ontology represents the data format leading to physical world actuation. Actuation can either be continuous (e.g., motor speed control) or binary (e.g., turning a light on or off). A binary actuation can also be monostable or bistable. Eventually, the transduction of digital information to the physical world is realized by a DAC.
- *Gateway*: This smart device class is used to categorize devices that perform digital-to-digital conversions such as signal processing or protocol translation. It is characterized, similar to sensors and actuators, by a set of both input and output that describe the different expected data formats.

The purpose of the ontology-based description of smart devices is to enrich subsequent steps in the design method, and thus directly depends on concepts assigned to the smart device. From the set of requirements provided by step 0 of the method (see the detailed design method depicted in Figure 3.3), an ontological concepts (also called classes) matching is performed and the resulting output classes are used as objectives for the subsequent modular architecture design.

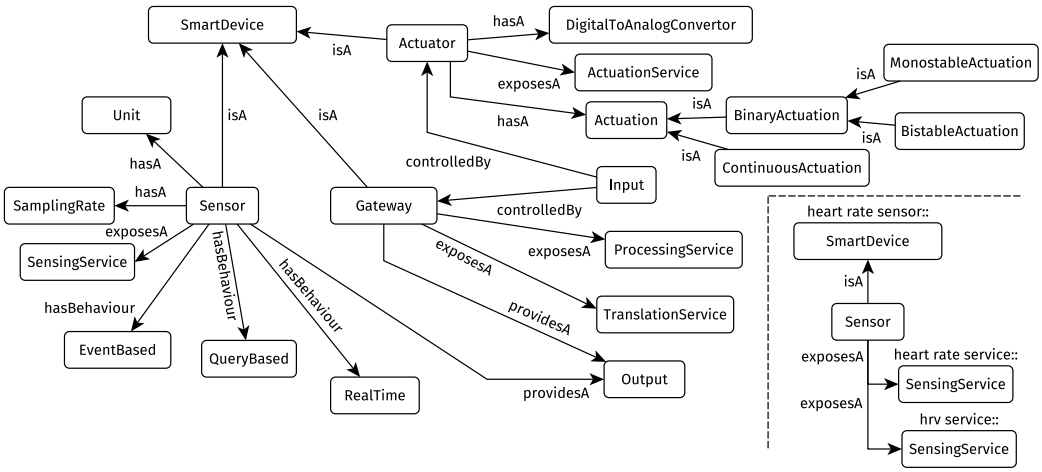


Figure 3.1 – Smart devices detailed ontology with a class instance toy example

Additionally, we detail a comprehensive smart devices' resources ontology in Figure 3.2. This ontology describes resources characteristics and their constraints, but also defines four distinct concepts: connectivity, energy, memory and computation.

First, smart devices connectivity is divided into three sub-classes (or sub-concepts). The *port connectivity* sub-class designates the physical connectivity of a given smart device (e.g., USB, Ethernet, etc.), while the transmission connectivity sub-class characterizes its wireless capabilities. These properties are key elements for smart devices interaction with external tiers. The interconnected bused sub-class describes the capability of devices to connect to higher level publish/subscribe buses, which are particularly suited for smart devices inter-connectivity.

Then, the *energy* sub-class describes battery power management along with some key battery characteristics, such as the maximum electrical charge (usually given in mAh) and the current electrical charge. The battery management services represents self-awareness regarding battery level and can be used as a basis for smart self-adaptive behavior.

The *memory* and *computation* sub-classes represent smart devices' capabilities to store, buffer or process data. Computation can be performed using common 8 to 32-bits micro-controllers (from small 8051 cores to powerful ARM microcontrollers), with a wide range of clock frequency (from a few megahertz to a few hundred of megahertz). Dedicated signal processors can also be used to perform real-time low-energy computation. The two sub-classes related to memory and storage are divided into volatile RAM and persistent ROM. Typical amounts of RAM range from a few tens of kilobytes to a few hundreds, while common amounts of ROM range from a few hundreds of kilobytes to a few thousands. More memory typically requires discrete ICs (usually either flash memory or a micro-SD card port), and the management of such component (e.g., RAM buffering followed by batch memory writing) is described by the memory management service.

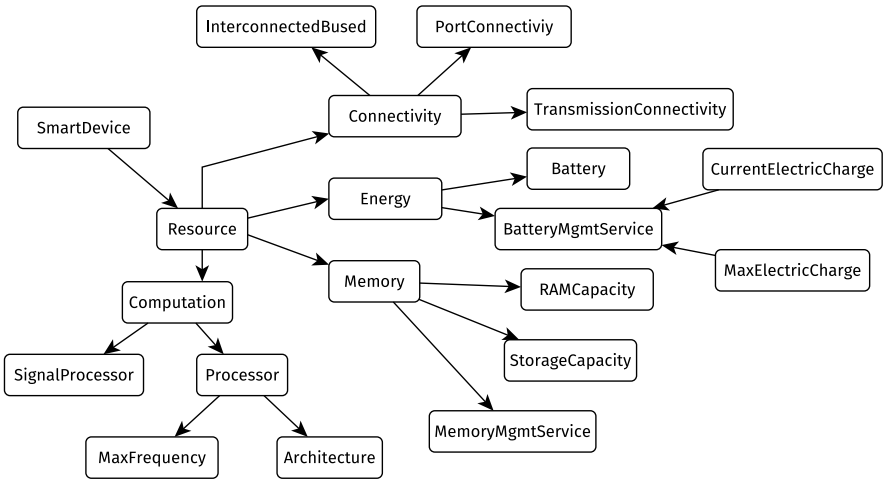


Figure 3.2 – Smart device detailed resources ontology

Ontologies for smart devices, sensor networks, or generic IoT-based systems have been developed in standard bodies and the academic literature (i.e. FIPA-Device ontology [1] or the SSN ontology [2]). However, they lack accurate representation of smart devices' hardware. Since our design method relies on a comprehensive knowledge of smart devices' hardware capabilities, our ontology attempts at the simultaneous modeling both smart devices and their limited resources.

3.3 Comprehensive Method Description

The design method lifecycle is given in Figure 3.3. It can be divided into three major phases: preliminary smart devices analysis (represented in blue in Figure 3.3), smart

1. FIPA-Device ontology website: `http://www.fipa.org/specs/fipa00091/PC00091A.html` (visited on 06/29/2018)
2. SSN ontology website: `http://w3c.github.io/sdw/ssn/` (visited on 06/29/2018)

devices specification and verification (in red) and eventually smart devices implementation and testing (in green). The first phase focuses on the study of the targeted smart device from different perspectives such as functional and non-function characteristics. An ontological analysis the device being designed is performed and used to assist system designers in the subsequent methodological steps. Finally, the phase ends with the definition of a reference modular architecture by which hybrid hardware-software modules are defined and articulated in order to fulfill functional goals while considering smart devices resources constraints.

The second phase is related to formal specification and validation of smart devices, represented as hardware/software hybrid smart components. The formal verification is mandated by the critical nature of smart devices: indeed, they are commonly used in various scenarios such as remote health monitoring or smart houses, where errors or failures can have disastrous consequences (e.g., loss of diagnostically relevant medical data, house security breach, etc.). Formal specification and subsequent verification helps to detect and correct design errors before devices implementation.

Eventually, the third and last phase consists of the implementation of the specified modules, along with their integration followinf to the predefined modular architecture, and prototype testing. Test procedures depend on the smart devices' category. By passing all defined tests, the device becomes ready for final production. Because of their self-containment and interoperability, modules can be considered as services, and the modular architecture thus relies an event-driven choreography of concurrent services, coordinated in order to achieve a predefined goal.

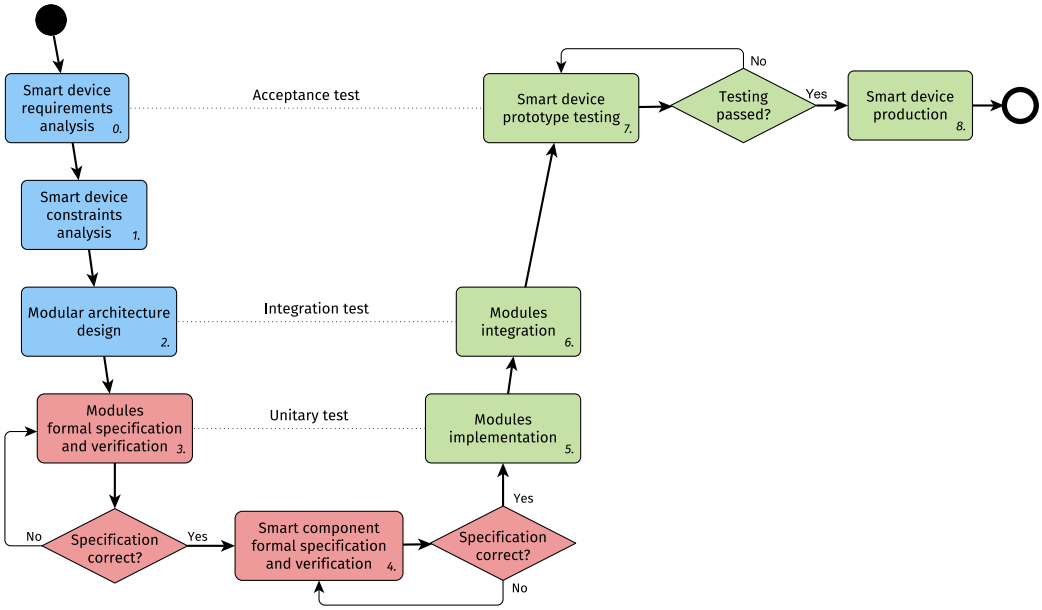


Figure 3.3 – Method lifecycle

As shown in Figure 3.3, the three phased (by color) steps are divided into individual method steps, and we will describe them extensively bellow.

3.3.1 Smart Device Requirements Analysis

The requirement analysis is closely tied with the functional analysis, and it establishes a list of various system properties. This requirement analysis can be performed through brainstorming, and it mainly answers the following questions: *what* is the purpose of the smart device, *how* is the device achieving this goal, *what for* is the smart device manufactured, and eventually *who* is it targeted to. By answering these questions, a list of functional and non-functional requirements can be created, and that list is used as an input for the subsequent step of the method lifecycle.

3.3.2 Smart Device Constraints Analysis

The smart devices constraints analysis is based on the resources ontology given in Figure 3.2, and a constraint list given bellow. Available resources for smart devices design are defined from the requirements analysis performed in step *0* of the method, and a summary of mutual relationships between constraints is given in Figure 3.4. The main constraints for smart devices are:

- *Size*, which can be represented by the final volume of the smart devices, usually expressed in cubed centimeters. This constrain directly impacts the battery life, because smaller sizes implies smaller batteries. The volume also impacts storage and computing capabilities because reducing global volume compels a smaller electronic footprint, and smaller components are commonly less powerful than their larger counterparts.
- *Computing capabilities*, which is usually evaluated using the microcontroller clock speed. The clock speed is usually configurable, and high speeds negatively impact battery life as they cause higher energy consumption [Wei+96].
- *Battery lifetime*, which is measured in hours, and is usually derived from the global hardware consumption of the circuit and the maximal battery electric charge.
- *Connectivity*: Connectivity impacts both battery life and computing capabilities. More advanced connectivity commonly results in higher microcontroller overhead, and conversely more powerful microcontrollers can be used to provide higher level connectivity.
- *Storage*: Larger storage implies larger electronic footprint, and it decreases battery life because of the energy cost of the writing and reading operation.

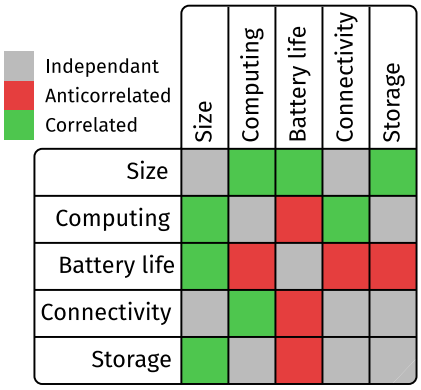


Figure 3.4 – Mutual constraints relationship

Both the constraint analysis introduced in this part and the requirement analysis introduced in step *0* support decisions of smart devices designers in order to produce a list of ICs, and to subsequently map software functions and services to the selected hardware components. This mapping results in the definition of hybrid software-hardware modules, that are organized together in the subsequent methodological step in order to produce the expected functional results. Finally, the outcome of this step is a list of hardware components carefully selected through the requirement and constraints analysis. This list can be considered as an optimum with respect to the different constraints presented above, and it is used as the basis for the next methodological step: the definition of a modular architecture.

3.3.3 Modular Architecture Design

In this step of our method, the smart devices ontology given in Figure 3.1 is used as a support to strengthen the device analysis. Using the ontology, manufacturers can assign one or several categories to the smart device, and the workflow is adapted consequently. This methodological step aims at instantiating classes from the ontology with respect to the smart device under development. A simple example of a class instance is given in the lower right corner of Figure 3.1. Comprehensive classes instances are used to define the modular architecture of smart devices.

The modular architecture design comprises the definition and organization of hybrid hardware/software modules that are, most of the time, concurrently executed using an event-based paradigm. This means modules execution is triggered by external events. Our event-driven modular architecture is typically implemented using interrupts, which must be classified by priority in order to avoid interrupt clashes. The scope of interrupts is however wider than a specific module, as they typically occur for an entire hardware component. For instance, if a microcontroller implements several modules, interrupts still occur at the microcontroller level, and designers must take that element into account. In addition to the modular architecture, smart devices designers should consequently produce a list of interrupts associated with a uniquely-defined priority.

Furthermore, we define two kinds of modularity: internal and external. Internal modularity is the result of the smart devices' internal organization into self-contained hybrid modules that communicate trough buses, as illustrated in Figure 3.5. External modularity is achieved through the exposition of hardware services as standard interfaces. In our case, we chose to use a JavaScript Object Notation (JSON) description of hardware services, as depicted in Figure 3.5. The use of a standard serialization language provides descriptions of available services and enables interoperability. Hybrid modules communicate through hardware buses, and serialized communication occurs through DMA to minimize microcontroller usage. Modules expose their functional and non-functional characteristics through input or output ports. They are capable of processing data to achieve functional goals (i.e., convert a voltage into numerical information, and then convert digital information to semantically-annotated data).

This modular architecture presents a fractal property because a module can be specified with several sub-modules. We thus empower smart devices designers with the capacity to chose the level of granularity they deem appropriate for their specific design, keeping in mind that highly optimized architectures often require a very small granularity to maximize hardware proximity.

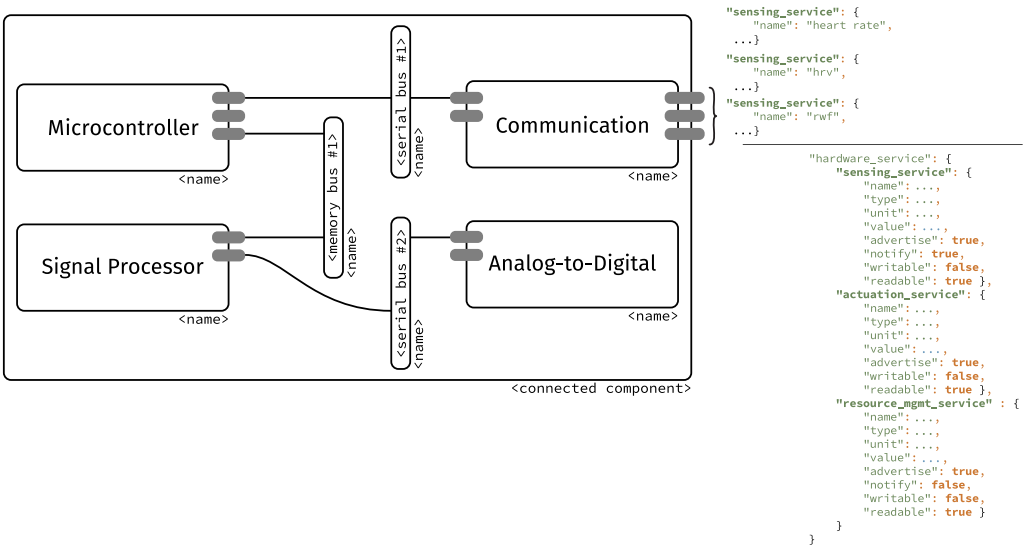


Figure 3.5 – Modular architectural simple generic example

3.3.4 Modules Formal Specification

In this methodological step, the modules outlined in the modular architecture are formally specified using a formal modeling language in order to establish a formal model that smart device designers can use in their implementation. Practically, every module need to be formally specified in order to achieve a comprehensive description of their behavior. This results in guarantees over the modules' behavior when implemented, hence resulting in a solid basis for more complex full smart component specification. Consequently, the activity of this design step consists of writing formal specifications for each module. As a result, it produces self-contained module specifications, which are later used for modules integration specification and verification.

3.3.5 Smart Component Formal Specification and Verification

Considered as a whole, smart components can be seen as distributed concurrent systems where all modules are running simultaneously. Communication between modules is, in the vast majority of cases, asynchronous, thus resulting in a strongly event-driven global architecture. Self-containing modules integration is then specified based on the models written in the previous methodological step, and model checking can be performed to verify if the system satisfies a set of predefined properties. The set of properties to be verified through model checking can be any formally specified properties such as system safety and liveness. Safety is informally defined as ensuring that a "bad" event never happens during the execution of a system, while the liveness definition is that a "good" event eventually happens [AS87]. If errors or deadlocks occur, they are identified during the model checking step, and the error trace provides a way for smart devices designers to identify and correct errors in their specification. The module and system specifications are inputs for the next methodological step, effectively defining references to be followed for the software and hardware implementations.

3.3.6 Modules Implementation and Integration

In this step, smart devices designers develop all modules as defined in the modular architecture, following the specifications produced in the step *3*. Because of system modularity, each hardware component typically comes with its own development toolkit (i.e., IDE, compilation chain, programming protocols), and system developers use the appropriate tools for each hardware sub-system. Languages used to develop the modules can also vary, as ICs manufacturers offer several options for their products: traditional C or less commonly C++ for microcontrollers, assembly for highly specialized signal processors, or even some proprietary and optimized scripting language for dedicated communication-oriented SoCs. The development of all modules produces programs than can be use in the subsequent step of the method, which deals with modules integration. Because of the hybrid nature of smart devices, the integration can happen both at the hardware or at the software level. For modules that are hosted by the same IC, the integration occurs at the software level, while for modules implemented on separate hardware components, the integration must be applied at the hardware level. The final smart component integration is the result of this methodological step.

3.3.7 Smart Device Testing

In our method, we propose the to use integration testing as traditionally used in software development. Even though formal specification and verification should prevent unexpected bugs to be discovered during the testing phase, complexity added by combining hardware discrete component still makes this methodological step mandatory. Since we want to test if the integration of discrete modules achieves predetermined functional and non-functional goals, we should test if the integration of modules is correct, and we thus perform integration testing. The testing procedure is summarized in Table 3.1, and depends on the ontological analysis of the smart device being designed. It is divided into four distinct cases: sensors, actuators, gateway, and integrated sensors and actuators. This categorization results from the similar domain transformations introduced by these cases, as displayed in Table 3.1. For instance, if sensors and gateways' classes are assigned, this will result in a physical → digital → digital transformation, that can be reduced to a physical → digital transformation from an integration point of view. Similar domain compression can be made for the other two classes assignment.

Table 3.1 – Detailed integration test description

Ontological analysis	Transformation	Testing to be performed
Sensor	Physical → Digital	Input: physical simulated signal Test: comparison of system digital output to expected output
Actuator	Digital → Physical	Input: digital signal Test: comparison of measured physical output to expected output
Gateway	Digital → Digital	Input: digital signal Test: comparison of digital output to expected digital output
Sensor & Actuator	Physical → Physical	Input: physical simulated signal Test: comparison of measured physical output to expected output

Testing helps to discover potential integration errors overlooked in formal specification and verification. Indeed, previous methodological steps are only related to the specification of the device, and misinterpretation of this specification by smart devices designers can still occur during the implementation, which can result in errors in the

smart device implementation. If further testing is required, a set of scenarios can be defined and the response of smart devices to the test scenario can be evaluated. In the case where testing is successful, smart device prototypes can be considered as final and the last methodological step can be performed. If pure software testing must be performed, typical testing procedures from the software engineering community can be applied, such as unit testing or traditional integration testing.

3.3.8 Smart Device Production

This last step transforms the prototype into the final product. To this end, an electrical schematic of the device is drawn, a precise bill-of-materials is established and PCB routing is performed. Final embedded software development is also realized. These documents can then be used to subcontract manufacturing to specialized electronics production companies.

We introduced our detailed design method for service-oriented smart devices, seen as a smart component made of a set of modular hybrid software and hardware component, with low-level services exposed using the communicating capabilities of the smart device. To demonstrate the efficiency of our proposed method in the context of smart devices development, we applied it to the design of a medical-grade electrocardiogram (ECG)-based cardiorespiratory sensor.

3.3.9 Adding Self-Adaptive Behavior as a System-Wide Requirement

As mentioned in the introduction chapter, self-adaptive behavior is an enabler for the construction of more intelligent devices, as it provides a mechanism for IoT-based systems to adapt to their external environments. As we promote the integration of self-adaptation mechanism in the smart devices layers of the IoT, they must integrate the capability to be either internally or externally reconfigured.

Practically, the integration of self-adaptive behavior can be seen as a constraint over Non-Functional Properties (NFP). Indeed, similar to sampling rate or battery life, self-adaptive behavior can be seen as a non-functional requirement pushing towards specific technical choices. Self-adaptation typically occurs through system reconfiguration, meaning that system designers must chose easily-reconfigurable SoCs. This can be done by preferably selecting digital industrial and integrated solutions rather than in-company analog development (i.e., choosing an already-existing IC over a new design derived from discrete components).

In addition to its impact on hardware choices, integrating a self-adaptive behavior into devices also requires specific software organization. In order to better match contributions dealing with the adaptation of reactive systems, we promote the integration of software state machines, corresponding to Labeled Transition System (LTS)-based models typically used by the reactive systems community. By such, transitions can then be associated with events, which typically take the form of interrupts associated with conditions in microcontroller-based reactive systems.

Eventually, self-adaptation can be seen as the reconfiguration, or reorganization, of the modular architecture defined in the methodology. Indeed, by defining self-contained hybrid modules, we can adapt smart devices simply by appropriately reconfiguring all the modules involved in the achievement of a specific adaptation goal.

3.4 Medical Smart Device Design

In this section, we illustrate the application of our method to the design of a medical-grade cardiorespiratory smart device. This sensor was developed in this thesis, and it is explained in details in the next chapter. In these design and development activities, all the methodological steps are illustrated, but we chose to treat them as phases rather than detailed steps for conciseness purposes.

3.4.1 Smart Device Analysis

The first step of our design method is smart device analysis, and we answer the four questions given above:

- *What*: measures Heart Rate (HR) and Heart Rate Variability (HRV) parameters, along with the Respiration Waveform (RWF), with at least 48 hours of autonomy. All measurements must be securely transmitted.
- *How*: based on ECG measurement and impedance pneumography, the smart device must be small and lightweight in order to achieve a reusable chest-patch form-factor.
- *What for*: perform real-time mobile monitoring of vital data for clinical studies.
- *Who*: biomedical and clinical researchers.

As a consequence from these answers, the functional requirements are the measurement of the HR and HRV parameters along with the RWF. The non-functional requirements are: the RWF is sent once every second by batch of three 24-bits values, the HRV parameters are transmitted every 5 minutes, the heart rate must be sent as soon as it is detected, and the autonomy must be of at least 48 hours. Additionally, the sensor should be self-adaptable in order to facilitate its integration to smart services, and this self-adaptive behavior should not impact quality of measurements. The expected QoS is that no data is lost during transmission, but also that the connection must be robust enough in order to maintain itself during 48 hours (if the transmitter is within the receiver range), and finally that data must be encrypted during the transmission.

Our smart device's ontological analysis is straightforward, and it is assigned to the sensor class. The sensor is both event-based and real-time. The reason for the first property is the fact that HR values are sent as soon as a new heartbeat is detected, and our smart device fits the real-time property because the RWF and HRV parameters are transmitted as soon are they are available. Battery levels can be requested by external tiers, thus making the sensor query based. Such external tiers can also be notified if the battery is low by subscribing to this notification channel. In terms of classes closer to the physical world, the analog-to-digital conversion occurs on two channels: the first channel is the direct measurement of the voltage produced by cardiac activity, while the second channel injects a high-frequency electrical current and measures the resulting voltage, thus effectively measuring the impedance variations between the two electrodes, which is correlated to respiratory activity. Since the sensor measures several phenomena, the unit class is a set: the HR unit is beats per minutes (BPM), the HRV parameters are given in milliseconds and percentages, and the RWF does not have any unit (also abbreviated as n.u. in this dissertation). The output of the analog-to-digital conversion is provided as two signed 24-bits frames (one for each channel), which corresponds to medical-grade resolution.

The next method step is smart devices resources analysis. With the assistance from our ontology, we assign the connectivity class to the transmission connectivity sub-class. Indeed, the goal of our smart device is to perform real-time medical-grade mobile measurement, so a port connectivity is not appropriate, and the constraints in terms of size and battery life prevents the deployment of a power-hungry Transmission Control Protocol (TCP)-based bused connectivity. Based on a state of the art of traditional low-power wireless connectivity protocols [Dem+13], we chose to utilize BLE to provide wireless connectivity. Our motivation was principally its widespread adoption in everyday devices such as smartphones, but also its low energy consumption. In terms of practical implementation, we used a Silicon Labs BLE113 SoC, which includes a fully programmable BLE stack with a relatively low surface footprint and low power requirements. All the communication-oriented functions of the sensor can be directly implemented on that circuit in order to reduce the load of other hardware, and we give more details about this implementation in the next chapter. When it comes to the energy class, the maximal electrical charge sub-class was assigned to the value of 300 mAh. This battery capacity represents an acceptable compromise between a small and lightweight targeted form factor and the desired autonomy of 48 hours. The battery management service simply consists of a battery level information available to external tiers. Since data is transmitted as soon as it is available, memory requirements are low as it does not need to be stored for extended periods of time. The strongest memory constraint is the need to store a 5 minutes long buffer of HR in order to compute the HRV parameters. There is no memory management services since there is no need to implement any storage-dedicated hardware.

The requirements in terms of computation are higher than most smart devices because of the relative computational complexity of real-time ECG processing. Indeed, ECG signal is acquired and processed at a rate of a 1000 samples per seconds (SPS). To reduce microcontroller load, it is better to process this data on a dedicated signal processor. We chose the PSoC5LP SoC as the computational hardware in our solution, because it features a powerful ARM Cortex-M3 microcontroller, a dedicated programmable signal co-processor and a fully programmable logic gates matrix, while remaining relatively low-power. Since most of the heavy real-time signal processing is performed by a dedicated power-optimized hardware, namely a programmable Digital Filter Block (DFB), the microcontroller frequency can be kept to a minimal 3 MHz, while being in sleep mode most of the time, thus saving energy.

The low-power yet high-quality analog signal acquisition requirements resulted in the choice of the ADS1292R for our ADC. This IC is an energy-efficient Analog Front-End (AFE) specifically dedicated at biomedical signal acquisition. Signals are acquired at a sampling rate of 1 kSPS, with 24-bits resolution. This component can also handle modulation and demodulation necessary for impedance pneumography. It is also entirely configurable, and communicates with the microcontroller using a Serial Peripheral Interface (SPI) serial communication.

The next methodological step is the definition of a modular architecture, which is illustrated in Figure 3.6. This diagram is simplified to include only the signal-processing path between the different modules. The analog-to-digital conversion is realized by a dedicated configurable front-end, which communicates with both the ARM core and signal co-processor using a serial bus and DMAs. The signal co-processor takes the raw ECG signal as an input and outputs a smoothed squared derivative of this signal, which

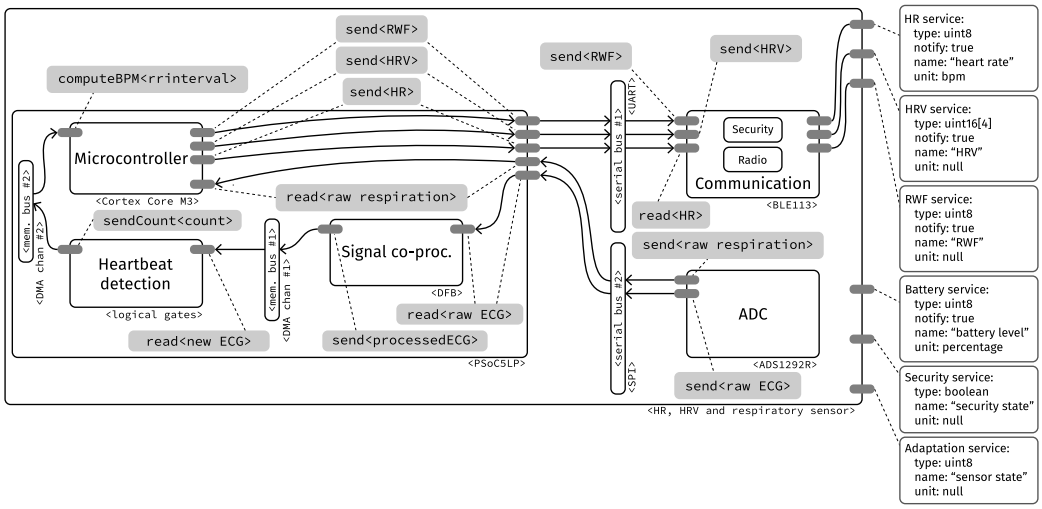


Figure 3.6 – Cardiorespiratory sensor modular architecture

will be used for heartbeat detection implemented using a set of programmable logic functions. This heartbeat detector determines the time interval between two heartbeats and communicates it to the microcontroller using a DMA bus, which are represented as memory buses on the figure. The HR, HRV parameters and RWF are computed by the microcontroller and communicated to the communication-oriented SoC using a Universal Asynchronous Receiver Transmitter (UART) serial bus. The BLE113 integrated circuit handles all the communication-oriented tasks of the sensor, such as external tier connection/disconnection, hardware services advertisement, HR value and HRV parameters communication, etc.

Further details about our smart devices implementation are given in the next chapter, with a strong focus on signal processing and peak detection. In addition, we also cover also the integration of self-adaptive capabilities in order to improve our sensor's intelligence.

The before-mentioned modules are concurrent with asynchronous communications. Please note that even though some buses used (such as the SPI bus) in our implementation are defined as synchronous because devices share a clock signal, we still consider interactions to be asynchronous because we implement an event-based mechanism rather than a polling approach. In the next section, we complete our medical smart device design with a formal specification developed for each module.

3.4.2 Smart Device Specification

Once the modules are defined and organized, our design method imposes a formal specification of each module. This results in the definition of a precise set of procedural steps a module must follow in order to achieve its goal-oriented objective. Several formal specification frameworks exist (e.g., UPPAAL, PRISM, TLA+, etc.). In our method we chose the temporal logic of action (TLA+) [Lam93]. In fact, this specification framework was chosen principally because it does not require advanced knowledge of complex mathematical concepts, consequently resulting in faster specification writing and higher return on investment, as this framework can be used to express a wide range of complex concurrent systems [New14]. Additionally, this framework was proven to be efficient for

both high level software specifications [New14] and low level hardware modeling [BL03] in industrial use-cases.

In this section, we introduce a sample specification for two modules of our medical smart device, along with an example of integration of two modules. For an illustration purpose, we chose to specify the heartbeat detection along with the HR computation from the microcontroller. The full specification is given in Figure 3.7. Heartbeat detection is specified in the module called *HRD*. Its specification is pretty straightforward, and it basically verifies if the condition to detect a heartbeat is satisfied or not. Every-time this condition is not satisfied, a counter is incremented, the current value of this counter is buffered, and a new value is loaded. Additional details about this algorithm are given in the next chapter. The condition simply states that the *old* sample of processed ECG must be greater than the *new* version of the sample, while the *new* sample is higher than a constant threshold and the counter is higher than 250. If this condition is verified, a Boolean variable is set to true. The heart rate processing specification is given in the *Proc* module. This module straightforwardly converts the interval between two heart beats in milliseconds to a HR value in BPM by dividing 60,000 by the variable *rrinterval*. A wrapper for the *HRD* module was written in order to keep the *old* and *new* variables of the module internal, as they should not be exposed to the external world. With this wrapper, the HRD module exposes only the Boolean flag representing a heartbeat detection and the counter value at the time of the heartbeat. The *Proc* module exposes both *rrinterval* (interval between two heartbeats) and *bpm* (HR value) variables.

```
MODULE HRD

EXTENDS Integers, TLC

CONSTANT Threshold
VARIABLES old, new, hr_detected, count

vars ≜ ⟨old, new, hr_detected, count⟩

TypeInvariant ≜ ∧ old ∈ 0 .. 255
                ∧ new ∈ 0 .. 255
                ∧ hr_detected ∈ BOOLEAN
                ∧ count ∈ 0 .. 2500

Init ≜ ∧ old = 0
       ∧ new = 0
       ∧ hr_detected = FALSE
       ∧ count = 0

Next ≜ IF (old > new) ∧ (new > Threshold) ∧ (count > 250)
       THEN ∧ old' = new
            ∧ new' = RandomElement(0 .. 255)
            ∧ hr_detected' = TRUE
            ∧ count' = 0
       ELSE ∧ count' = count + 1
            ∧ old' = new
            ∧ new' = RandomElement(0 .. 255)
            ∧ UNCHANGED ⟨hr_detected⟩

Spec ≜ Init ∧ □[Next]_vars

THEOREM Spec ⇒ □TypeInvariant
```

```
MODULE Proc

EXTENDS Integers, Reals

VARIABLES rrinterval, bpm
CONSTANT Message

Init ≜ ∧ rrinterval = 0
       ∧ bpm = 0

TypeInvariant ≜ rrinterval ∈ 0 .. 2500

BPMCompute ≜ ∧ bpm' = 60000 ÷ rrinterval
             ∧ UNCHANGED ⟨rrinterval⟩

Next ≜ BPMCompute

Spec ≜ Init ∧ □[Next]_⟨rrinterval, bpm⟩

THEOREM Spec ⇒ □TypeInvariant
```

```
MODULE Integration

EXTENDS Integers, Reals

CONSTANTS Message, Threshold
VARIABLES dma, hr_detected, count, bpm, rrinterval

HRDWrap ≜ INSTANCE HRDWrapper
ProcBPM ≜ INSTANCE Proc
DMAChan ≜ INSTANCE Channel WITH Data ← Message, chan ← dma

Initialization of all the modules
Init ≜ ∧ HRDWrap!Init
       ∧ ProcBPM!Init
       ∧ DMAChan!Init

To check the boundary-correctness
TypeInvariant ≜ ∧ HRDWrap!TypeInvariant
                ∧ ProcBPM!TypeInvariant
                ∧ DMAChan!TypeInvariant

SendRRInt ≜ IF hr_detected
            THEN ∧ DMAChan!Send(count)
                 ∧ ⟨bpm, rrinterval⟩
            ELSE UNCHANGED ⟨dma, bpm, rrinterval, hr_detected, count⟩

ReceiveRRInt ≜ IF hr_detected
               THEN ∧ DMAChan!Rcv
                    ∧ rrinterval' = dma.val
                    ∧ UNCHANGED ⟨bpm, hr_detected, count⟩
               ELSE UNCHANGED ⟨dma, bpm, rrinterval, hr_detected, count⟩

ComputeBPM ≜ IF hr_detected
             THEN ∧ ProcBPM!BPMCompute
                  ∧ UNCHANGED ⟨hr_detected, count, rrinterval, dma⟩
             ELSE UNCHANGED ⟨dma, bpm, rrinterval, hr_detected, count⟩

Next ≜ ∨ SendRRInt
       ∨ ReceiveRRInt
       ∨ ComputeBPM

Spec ≜ Init ∧ □[Next]_⟨dma, hr_detected, count, bpm, rrinterval⟩
```

Figure 3.7 – Specifications of heart rate detection, processing, and integration

Modules integration result in a distributed concurrent architecture, and TLA+ is the ideal framework to specify our modules' integration because it was created to represent and analyze distributed and concurrent systems. Specifications of modules can be integrated as parts of the smart device model. Global specification is only the description of how the modules communicate in order to execute the functional goals of the smart device being designed. The model checker provided in the TLA+ toolbox verifies the safety and the liveness of the smart device. If more advanced devices' properties need to be verified, the TLA+ proof system can be used to derive hierarchical proofs, consequently adding a degree of trust that the smart device will behave as expected.

Going back to Figure 3.7, the *Integration* module describes how the two individual modules described above are articulated. Modules integration occurs through a DMA channel, using the channel specification detailed in [Lam02]. When a heartbeat occurs, a DMA transaction between the *HRD* module and the *Proc* module is initiated, and the counter value is transferred from the *HRD* module to the *Proc* module. Upon transfer finalization, the computation of the HR in BPM is triggered. The *Integration* module is then model checked for two system characteristics: deadlock and type correctness. The first property describes the lack of situation where the system encounters states in which no progress can be made, while the second characteristic gives the assurance than the values produced by the system are within an expected range. Model checking resulted in a state space of 5002 distinct states, and no deadlocks neither boundaries violations were found. The sample specification and model checking thus gives the system developer guarantees that the smart device cannot be in an irremediable blocking state and that no value overflows will occur. This formal specification serves as the basis for module development and system integration, and developers must pay a particular attention to follow the specification.

Services exposed to the external world can also be specified. Here, we use the HR service as an example to illustrate how such specification can be used. We used the PlusCal specification language to demonstrate the versatility and ease-of-use of the TLA+ toolbox. PlusCal is a specification language that can be used to specify algorithms, and that transpiles to TLA+ specifications for the model checking process. The HR service is quite straightforward: when a new message is received from the microcontroller, the service checks the message header to verify if it corresponds to the appropriate HR message identifier. If it does, the service updates its HR value, which is then notified to external tiers or remotely accessed upon request. This service was specified using PlusCal and transpiled to TLA+, as shown in Figure 3.8. Model checking was performed and no errors were found.

3.4.3 Smart Device Implementation: an Overview

In this section, we briefly explain our hardware architecture leading to a first version of our real-world prototype. We however explain the final steps of our design method (namely testing and industrial-level production) in the next chapter, where details about the implementation and testing of our cardiorespiratory sensor are extensively described. We implemented our sensor using our method, and we thus adopted a module-by-module approach during the design and development lifecycle. These modules were implemented using the technology (in terms of languages, Integrated

```
--algorithm HRService{
  variable header_correct ∈ BOOLEAN ;
           service_hr ∈ 24 .. 240 ;
  procedure CheckHeader(message)
  {
    ckh: header_correct := (message.head = HRheader) ;
    rckh: return ;
  }
  {
    main: while (TRUE){
            call CheckHeader(message) ;
            up: if (header_correct)
                {
                    service_hr := message.hr
                }
        }
  }
}
```

```
MODULE Service
EXTENDS Integers, TLC, Sequences
CONSTANT HRheader
VARIABLES header_correct, service_hr, pc, stack, message
vars ≜ ⟨header_correct, service_hr, pc, stack, message⟩
Init ≜ Global variables
       ∧ header_correct ∈ BOOLEAN
       ∧ service_hr ∈ 24 .. 240
       Procedure CheckHeader
       ∧ message ∈ [head : 0 .. 10, hr : 24 .. 240]
       ∧ stack = ⟨⟩
       ∧ pc = "main"
ckh ≜ ∧ pc = "ckh"
      ∧ header_correct′ = (message.head = HRheader)
      ∧ pc′ = "rckh"
      ∧ UNCHANGED ⟨service_hr, stack, message⟩
rckh ≜ ∧ pc = "rckh"
       ∧ pc′ = Head(stack).pc
       ∧ message′ = Head(stack).message
       ∧ stack′ = Tail(stack)
       ∧ UNCHANGED ⟨header_correct, service_hr⟩
CheckHeader ≜ ckh ∨ rckh
main ≜ ∧ pc = "main"
       ∧ ∧ message′ = message
         ∧ stack′ = ⟨[procedure ↦ "CheckHeader", pc ↦ "up", message ↦ message]⟩
  ◦ stack
     ∧ pc′ = "ckh"
     ∧ UNCHANGED ⟨header_correct, service_hr⟩
up ≜ ∧ pc = "up"
     ∧ IF header_correct
         THEN ∧ service_hr′ = message.hr
         ELSE ∧ TRUE
              ∧ UNCHANGED service_hr
     ∧ pc′ = "main"
     ∧ UNCHANGED ⟨header_correct, stack, message⟩
Next ≜ CheckHeader ∨ main ∨ up
Spec ≜ Init ∧ □[Next]vars
```

Figure 3.8 – Heart Rate service specification: PlusCal and automatically generated TLA+

Development Environments (IDE), etc.) provided by each hardware manufacturer, in accordance with the specification developed in the previous section.

The communication modules to be deployed on the BLE113 were developed using the proprietary scripting language and compilation tool-chain provided by Silicon Labs. The implemented modules were: services advertising on system startup, connection management, deconnection management and transmission of the measured parameters using the appropriate hardware services. During idle periods, the BLE113 goes to a low-power sleep mode and it is woken-up by the microcontroller if new measurements are available and ready to be transmitted. The modules defined as being part of the PSoC5LP were developed using the IDE provided by the manufacturer (i.e., PSoC Creator). This tool chain can be used to simultaneously develop C code deployed on the ARM Cortex-M3 core and domain-specific assembly code for the signal co-processor. Logical functions arrangements to be deployed on the programmable logic gates matrix can either be programmed using a graphical interface or Verilog (i.e., a common Hardware Description Language (HDL)). Each of the modules were consequently implemented using their dedicated language, and deployed on the PSoC5LP IC before being interfaced with other modules. The ADC is programmed by the microcontroller through serial communication on bootstrap startup, and it is configured to run using its internal clock and to provide two samples from both channels every one millisecond. New sample availability is advertised by changing a logical level on a specific pin of the circuit, which triggers clock generation for serial communication on the PSoC5LP.

From a hardware standpoint, the first prototyping step consists of the integration of each IC's development board, as depicted in Figure 3.9. The preliminary prototype was used to perform a series of tests, which will be described in the next chapter. In addition, we describe sensor miniaturization, and how to convert a development board -based

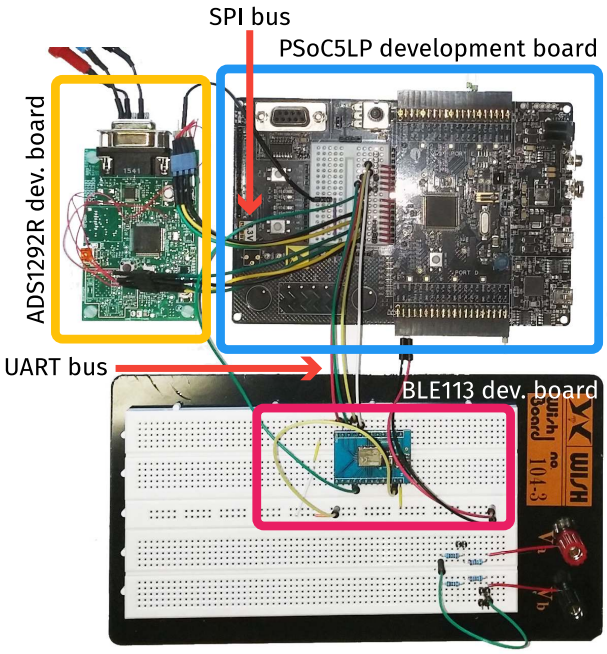


Figure 3.9 – First version of the cardiorespiratory sensor prototype

prototype (rendering it relatively useless for real-world applications) into a full-fledged mobile smart device.

3.5 Summary and Conclusion

In this chapter, we present a service-oriented design method for smart devices. Our principal motivation for this contribution stems from the fact that smart devices have unique features, principally due to their hybrid nature, but also because they are commonly used in critical applications in the context of IoT-based systems.

Even though numerous design methods have been described in the scientific literature, we found them to be lacking key characteristics when it comes to the design of smart devices. Namely, we emphasize on the need for such devices to be interoperable, modular and reusable in order to promote their integration in a wide variety of applications. To this end, we focus our research on SOAs and related concepts of self-contained, modular and interoperable services. The numerous design methods aimed at the development of service-oriented systems typically lack accurate hardware representation. Indeed, SOAs focus on software systems, and use tools which are often resources-hungry preventing their integration to resources-constrained devices. We however reuse the service concept in order to improve the modularity and flexibility of smart devices. In addition, other design methods targeted at the development of embedded software were created, but they are typically targeted at wide scale systems (e.g., flight control systems, industrial production lines automation, etc.), and are thus a poor fit for smart devices considering they are typically small-scale physical objects.

Other design methods focused on the simultaneous design of hardware and software under the name of hardware/software codesign have advanced modeling for both domains. They can also be used to validate and verify the systems being designed.

However, such design methods usually assume that both hardware and software are automatically generated, which is the case for SoC or IC design. Nevertheless, this assumption does not hold anymore when it comes to smart devices design, which can be seen as the integration of several ICs rather than their actual design.

Consequently, there is a real gap between high-level service-oriented design methods and low-level hardware/software codesign methods. Our proposed method attempts at bridging these two domains to provide a full-fledged design method for service-oriented smart devices, with a strong focus on functional and non-functional properties. To this end, we rely on a modular architecture based on hybrid hardware/-software modules exposed to the external world through a set of hardware services. We also integrate formal specification and verification as dedicated steps of our design method in order to provide guarantees on final smart device behavior. Our method adopts a V-shaped lifecycle, which provides advantages over a waterfall lifecycle. Indeed, V-shaped approaches introduce dependencies between the development phase and the design phase. In opposition, a method step from a waterfall lifecycle only depends on the previous step, with the risk of not designing appropriate testing processes or to discover bugs late in the design lifecycle. By specifying (and sometimes performing) tests at the beginning of the design process, V-shaped methods minimize such risks. Additionally, we introduce resources and smart devices ontologies to assist system designers in their design and implementation of devices.

In summary, our method introduces a novel service-oriented approach regarding the smart devices design process, with a particular emphasis on formal specification and testing. This work focuses on providing a practical design method, while keeping in mind that the whole design process consists of the integration of several application-specific ICs in order to achieve functional goals with acceptable QoS. Even though we apply our method using the TLA+ specification language, it is not a tool-oriented. In our opinion, lack of tools represents a limitation considering system designers still have to chose tools on their own. This choice is sometimes not trivial, and it may be hard to differentiate between the numerous formal verification tools available on the market. Another limitation comes from the use of a V-shaped lifecycle. Even though it provides numerous advantages by integrating testing procedures early in the design process, V-shaped lifecycles were progressively replaced by agile methods, which emphasize aspects such as the wide-scale collaboration of cross-disciplinary teams in software engineering. However, the static nature of hardware make the adoption of design methods with an agile lifecycle challenging, and we believe the V-shaped lifecycle is an acceptable compromise in terms of flexibility and ease-of-use. Additionally, concepts such as continuous improvement in agile design methods can not easily be transferred at the hardware level because it is less dynamic than software, and the disastrous implications in terms of development time that a hardware change can have. For instance, the development of new bare metal code for a different microcontroller implies deep modification in the code as peripherals addresses and various libraries vary from a manufacturer to the other.

Finally, our design method is limited by its high-level nature. Indeed, we only introduced a sequence of design steps and identified potential tools that can assist system designers in the conception of a smart object, but our method lacks comprehensive and global tooling. Moreover, the lack of tools makes the method's verification harder, as smart device designers are not provided with means to verify the compliance of

their design process with our service-oriented method. A potential solution would be the inclusion of tools and techniques from the Model Driven Engineering (MDE) community, as they force smart device designers to adopt a specific modeling framework, which consequently makes the development of helper tools easier. However, the strong hardware heterogeneity and the lack of global abstraction for low-level resource-constrained systems is still an open research challenge preventing generic models from being developed, and this needs to be addressed before being included in our design method.

In the next chapter, we further present the application of our service-oriented design method with a particular emphasis on practical design and testing procedures, resulting in the final implementation of our cardiorespiratory sensor.

CHAPTER 4

Smart Device Hardware Design and Implementation

A multiparametric cardiorespiratory sensor

Contents

4.1 Introduction . . . 73
4.2 Sensor Scope Statement . . . 75
4.3 Sensor Design . . . 76
4.3.1 Hardware Architecture . . . 76
4.3.2 Embedded Hardware and Software Signal Processing . . . 79
4.3.3 Integrating Self-Adaptive Behavior . . . 84
4.3.4 Development of an Android Companion Application . . . 86
4.4 Sensor Evaluation . . . 87
4.4.1 Evaluation on Synthetic Signals . . . 87
4.4.2 Energy Consumption . . . 88
4.4.3 Sensor Precision Evaluation . . . 90
4.4.4 Mobility Evaluation . . . 93
4.5 Conclusion . . . 95

4.1 Introduction

In this chapter, we detail the implementation of a cardiorespiratory sensor following our design method. Theoretical parts of sensor development such as modules specifications and verifications were already treated in the previous chapter, and we consequently focus on the practical aspects, namely sensor testing, and the integration of self-adaptive properties into our sensor with a detailed study of their impact on overall performance. The development of a cardiorespiratory sensor was motivated by the clinical relevance of cardiac activity monitoring, which can be used as a predictor of several diseases through the computation of Heart Rate (HR) and Heart Rate Variability (HRV) parameters [Tas96]. The sensor is however not designed to be used as a standalone device, but rather to be integrated in a personalized healthcare framework where a hospital manages the

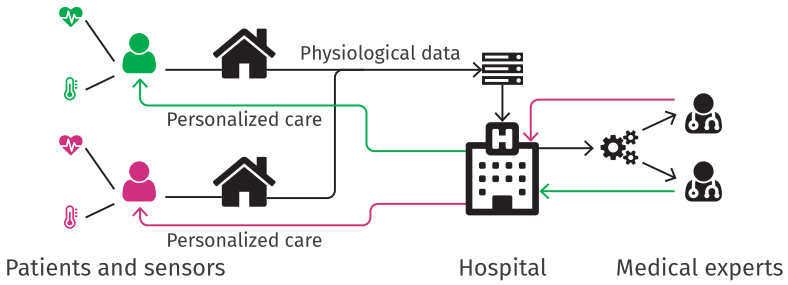


Figure 4.1 – Illustration of personalized healthcare for our case study

remote care of many patients by equipping them and their home with wearable and environmental sensors, as succinctly illustrated in Figure 4.1.

Even though such sensors are common and have already been described in the literature, all the identified wearable electrocardiogram (ECG)-based cardiorespiratory sensors do no feature advanced-enough connectivity and functionality to be integrated into wide-scale personalized healthcare framework. Indeed, this integration calls for continuous wireless communication with the external world through interoperable protocols, but also for the inclusion of self-adaptive behavior (typically through remote configuration) in order to build smarter systems and medical-grade accuracy in precision in order to provide guarantees about the clinical relevancy of the collected data. In a brief literature review, we focus on recently developed wearable ECG or ECG-based cardiac activity sensors because they can be used to accurately derive short term cardiac activity parameters [SV13]. For instance, authors of [Mag+14] implemented a wearable HR and respiration rate sensor with simultaneous Bluetooth Low Energy (BLE) and proprietary 802.15.4-based connectivity by integrating several development boards. This sensor however lacks remote self-adaptation capabilities and a self-contained prototype, which both are necessary for real-world usage. Another example can be found in [Tuo+17], where authors built a remotely configurable ECG sensor with local storage (i.e., data is stored on a SD card). This makes this sensor difficultly usable in personalized healthcare frameworks as data is stored locally and not streamed to external tiers. In [Izu+15] the construction of a remotely configurable ECG sensor is described. Both data and remote configuration are transmitted using Near Field Communication (NFC). Nevertheless, the short range of NFC is a strong limitation to sensor integration to healthcare systems since it implies a NFC transceiver must be periodically brought in proximity to the sensor to collect data or perform reconfiguration. Eventually, a HR and HRV parameters sensor is described by authors of [Mas+15], and it offers embedded signal processing in order to achieve on-board computation of both HR and HRV parameters. The measured variables are then transmitted using a wireless BLE communication. Nevertheless, this sensor does not embed self-adaptation capabilities, which limits its potential use as the basis of smart healthcare services. In addition, the recent study consider

In addition, the IoT was recently promoted as a technical solution to the design and implementation of large-scale personalized health systems as it features the capability to interconnect and manage numerous smart devices [Gub+13]. In this chapter and this manuscript, we define smart devices as physical objects embedded with computational capabilities, wireless communication and self-adaptive behavior. Particularly, we

emphasize the importance of performing signal processing at the lower level of the IoT stack (see Figure 1.1) in order to only transmit relevant information to the upper layer of the system, which is similar to what computer scientists describe under the term of *edge computing* [Shi+16]. Eventually, the combination of on-board signal processing along with both internal and external self-adaptive behavior represents a key element for the design of more intelligent devices, which can then be used as atomic elements for the creation of smart services.

In summary, this chapter will describe the implementation of a novel smart cardiorespiratory sensor following the design method detailed in the previous chapter, with a particular emphasis on the three following characteristics:

- *Medical grade precision*: Since our sensor is designed with medical applications in mind, it needs to be precise-enough. This can be achieved by using pre-certified medical-grade System-on-Chip (SoC), and is confirmed by the careful analysis of our sensor quantitative performances.
- *Self-adaptive behavior*: In order to be integrated to wide scale IoT framework, our sensor needs to implement both internal and external self-adaptive behavior, and it must be taken into account as a smart device-wide non-functional property.
- *Embedded signal processing*: By performing signal processing operations on board, our sensor only streams relevant data to external tiers. This approach reduces considerably the amount of transmitted information. By doing so, we are able to provide global bandwidth savings and can ensure systems built using embedded signal processing can scale up more easily than systems where raw data is continuously streamed at a much higher rate.

The remaining of this chapter is organized as follows: Section 4.2 extends the brief sensor functional analysis given in the previous chapter and discusses all the computed parameters along with their clinical relevancy. Section 4.3 discusses sensor hybrid architecture and details signal processing steps and algorithms embedded on our smart device. We then introduce in Section 4.4 the exhaustive testing procedures our sensor went through, both in terms of quantitative measurement analysis, but also in terms of longevity, mobility and energy consumption. Indeed, since the goal of our method is to provide smart devices usable in a variety of situation, extended testing of our sensor under real-life conditions should be performed. Eventually, our work dealing with the cardiorespiratory sensor is concluded in Section 4.5, where we discuss limitations and possible research directions.

4.2 Sensor Scope Statement

The first methodological step introduced in Chapter 3 is the definition of a set of requirements for the smart device being designed. Our high-level functional requirements for this sensor is to simultaneously measure HR and HRV along with the Respiration Waveform (RWF). Non-functional requirements are an extended battery life (i.e., at least 48 hours), internal and external self-adaptive behavior, medical-grade precision and being compact enough to be considered as a wearable product. The sensor should also be easily interoperable and should connect to usual appliances, practically meaning that the sensor should adopt a widespread wireless communication protocol. All of these requirements are necessary for our sensor to be easily included in telemedicine

systems or wider-scale Internet of Medical Things systems, where patients need to be continuously remotely monitored.

Real-time and on-board computed HRV parameters are described in Table 4.1. In summary, we compute the most common parameters both in the time and frequency domain, which results in comprehensive characterization of the RR intervals tachogram. Practically, HRV information can then be tied to various higher-level functions of the subject, such as stress [Lee+07] or physical activity, or more generally to autonomic nervous system balance [Hen+09]. However, HRV parameters estimation is a computationally intensive task because it requires time-to-frequency domain transformation. Additionally, the RR intervals tachogram is irregularly sampled (because HR are, by nature, irregular), and Power Spectral Density (PSD) estimation requires advances algorithms such as the Lomb-Scargle periodogram.

Table 4.1 – HRV parameters description

Variable	Unit	Domain	Description
SSDN	ms	Time	Standard deviation of RR intervals
RMSSD	ms	Time	Quadratic mean of differences between consecutive RR intervals
LF/HF	n.u.	Freq.	Ratio of the low-frequency (0.04 to 0.15 Hz) to high-frequency (0.15 to 0.4 Hz) components of the PSD of the RR intervals
Norm. LF	%	Freq.	Normalized low-frequency components to sum of low- and high-frequency components of the PSD ratio, i.e., LF/(LF+HF)

In conclusion, the combination of functional and non-functional properties drove us to the adoption of highly energy-efficient Integrated Circuits (IC) and SoCs, in order to simultaneously meet requirements over battery life and on-board of signal processing capabilities, and technical choices and indications about how they help to achieve desired functional and non-functional requirements are given in the next section.

4.3 Sensor Design

In this section, we detail how the modular architecture we detailed in the previous chapter is implemented. Additionally, we discuss the impact of technical choices over these architectures, but also over non-functional properties.

4.3.1 Hardware Architecture

The overall architecture of the sensor is displayed in Figure 4.2, which is only a simplified version of the modular architecture described earlier. This figure puts a particular emphasis on the digital communication buses used to interface the components, and illustrates modules interoperability. Because of the overhead introduced by the addition of self-adaptive characteristic to our device and the on-board signal processing, we selected components in order to maximize their computational and power efficiency.

Because microcontroller load and microcontroller active period reduction were important requirements of our sensor, real-time signal processing was performed using dedicated hardware. Indeed, by minimizing microcontroller active periods, we can put it in a sleep state most of the time, leading to important energy savings. To this end, we chose the PSoC5LP (Cypress Semiconductor, San Jose, CA) microcontroller. Indeed, this IC offers both an ARM Cortex-M3 core and a programmable Digital

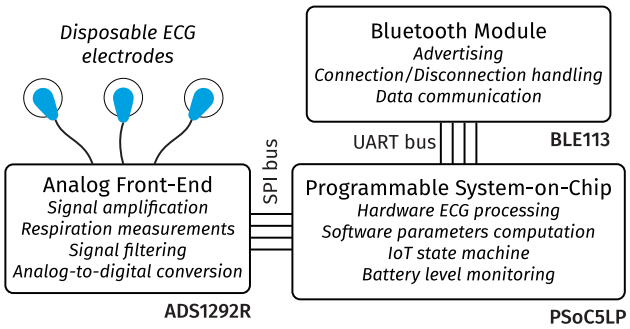


Figure 4.2 – Simplified block diagram of the sensor

Filter Block (DFB) in a single package. The DFB can be used as a dedicated signal co-processor, effectively reducing CPU load and thus decreasing overall power consumption. Because of the widespread adoption of BLE in smart devices [Har+16], it was selected to provide wireless connectivity to our sensor, thus providing optimal interoperability and ease-of-connection to external tiers. In addition, BLE is specifically designed for resources constrained applications, thus providing further energy savings at the smart device scale. We chose to use the BLE113 RF-dedicated SoC (Silicon Labs, Austin, TX) as the communication-dedicated microprocessor because it integrates both a full hardware and software BLE stack with integrated antenna design in a small 15.75 mm × 9.15 mm × 1.9 mm package. This IC is programmable, and it can handle all communication-dedicated code in a self-contained fashion, thus reducing the microprocessor computing load.

Practically, we used BGScript, a scripting language from Silicon Labs, to code the BLE113's firmware. We implemented a GATT server exposing hardware services described in the previous chapter (i.e., HR service, HRV parameters service, RWF service, battery service and adaptation service). We did not implement the security features of our smart device as a service, but we require an encrypted connection when a BLE device attempts to connect to our smart device. The additional mandatory (from the BLE standard) generic application profile and device information services were also implemented, and the HR service follows the official BLE specification.

In order to simultaneously measure both ECG and respiration signals, we selected the ADS1292R (Texas Instruments, Dallas, TX). This low-power Analog Front-End (AFE) integrates two differential amplifiers and two 24 bits Analog-to-Digital Converters (ADC). It also embeds a right-leg drive (RLD) amplifier implementing common mode rejection [WW83]. Circuitry handling lead-off detection and respiration signal modulation and demodulation is also present. Measurements are sent over a Serial Peripheral Interface (SPI) bus at a predetermined and configurable sampling rate (from 125 to 8000 samples per seconds). Transmitted data are divided into three 24-bit words: the first is a status word, which principally contains information about lead-off states, while the second and third words contain digital measurements of ECG or respiration signal, depending on the configuration of the IC.

In the default state, the AFE is configured to measure both the respiration signal on channel 1 and the ECG signal on channel 2 at a sampling rate of a 1000 SPS using the internal clock of the IC. This sampling rate provides a resolution of 1 ms for RR interval

measurements. The RLD amplifier and lead-off detection circuitry are both enabled, along with the respiration modulation and demodulation modules.

The first iteration of our prototype consisted in an assembly of development boards as displayed in the previous chapter. However, this preliminary step is not enough because it cannot be used in real world experiments, which are necessary to test our smart device mobility and precision in ambulatory applications. To this end, we developed a comprehensive electrical schematic of the sensor. We subsequently performed manual Printed Circuit Board (PCB) routing, and we subcontracted PCB manufacturing to an electronic manufacturer. This resulted in the mounted PCB displayed in Figure 4.3.(a), measuring only 40 mm × 20 mm × 6 mm. Additionally, we 3D-printed a compact casing, and the final smart device is displayed in Figure 4.3.(c). The final sensor dimension are 45 mm × 30 mm × 16 mm for a weight of 26.7 g. Consequently, our final smart device is small enough to be worn during daily activities, and we promote a sensor placement as displayed in Figure 4.3.(b).

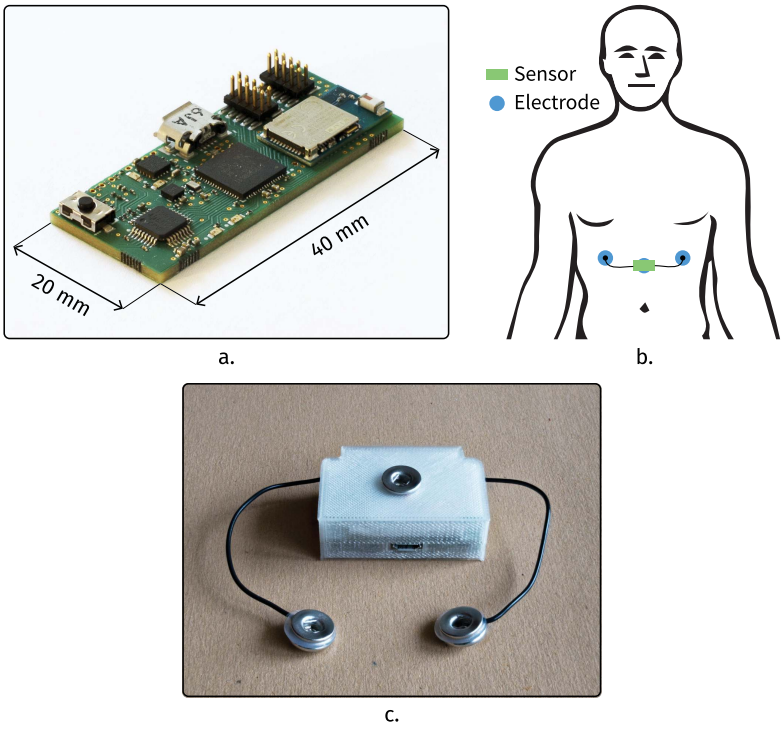


Figure 4.3 – Mounted PCB (a.), body placement (b.) and encased sensor (c.)

In the next section, we describe the division between hardware and software signal processing. Indeed, even though we tried to maximize the proportion of processing occurring on application-specific components of the SoC, not all functions are implementable using this approach. Indeed, the signal co-processor of the PSoC5LP is relatively small, and can only perform real-time filtering operations, which lead to the delegation of some signal processing computations to the ARM Cortex-M3 microcontroller.

4.3.2 Embedded Hardware and Software Signal Processing

The comprehensive signal processing chain is displayed in Figure 4.4. This graphic should be read from top to bottom: the upper part of the figure shows low-level (i.e., hardware or low-level software such as assembly) signal processing, while the lower part shoes the algorithm handling high-level signal processing operations.

Low-level signal processing is performed by a combination of pure hardware programmable logic and the PSoC5LP digital signal co-processor. The functional goals of this signal processing it to perform reliable and robust heart beat detection from the raw ECG signal, which is formally defined as the time interval between two R peaks (ECG signal is traditionally partitioned using letter indexes from Q to T, in the alphabetical order). The R peak corresponds to the sharper peak of the ECG, and is consequently easier to detect.

HR detection from the ECG signal is a widely explored field of study, and numerous solutions can be found in the literature, such as wavelet transform [CCC95] or Hilbert transform [Ben+00] based approaches, or pre-trained algorithms [AL12]. In our particular case, we chose to use the Pan and Tompkins [PT85] peak detection algorithm. Our choice was motivated by its real-time nature and relative simplicity, making an embedded implementation a practical reality. This algorithm consist of signal processing operations followed by simple peak detection based on Boolean conditions. The original algorithm specifies the chaining of a band-pass filter, a linear differentiator, a squaring function, and eventually a moving average filter [PT85]. In our specific use case, band-pass filtering is included in the ADS1292R, but we implement an additional low-pass filter, namely a moving average, in order to remove residual high-frequency artifacts from the ECG signal. This average is computed using 96 samples (i.e., the window width is 96 ms, and the optimal size was determined from experimental results). After this step, the Pan and Tompkins algorithm is thus reduced to differentiation, squaring and moving average, and these three steps are implemented on the PSoC5LP's DFB (which is also called Digital Signal Processor (DSP) or signal co-processor at it is dedicated at digital signal processing).

In terms of signal processing, the filters implemented are straightforward. The differentiation filter exhibits the following difference equation, which corresponds to an unweighted numerical differentiation:

$$d[n] = \sum_{j=1}^{N_d} x\left[n+j\right] - \sum_{j=1}^{N_d} x\left[n-j\right] \tag{4.1}$$

where $d[n]$ is the output of the differentiator at sample n, $x[\cdot]$ is the input and N_d is the differentiator's order.

The squaring function is trivially implemented as:

$$s[n] = (x[n])^2 \tag{4.2}$$

where $s[n]$ and $x[n]$ are respectively the output and input of the squaring function.

Eventually, the difference equation of the unweighted moving average is:

$$a[n] = \sum_{j=1}^{N_a-1} x[n-j] \tag{4.3}$$

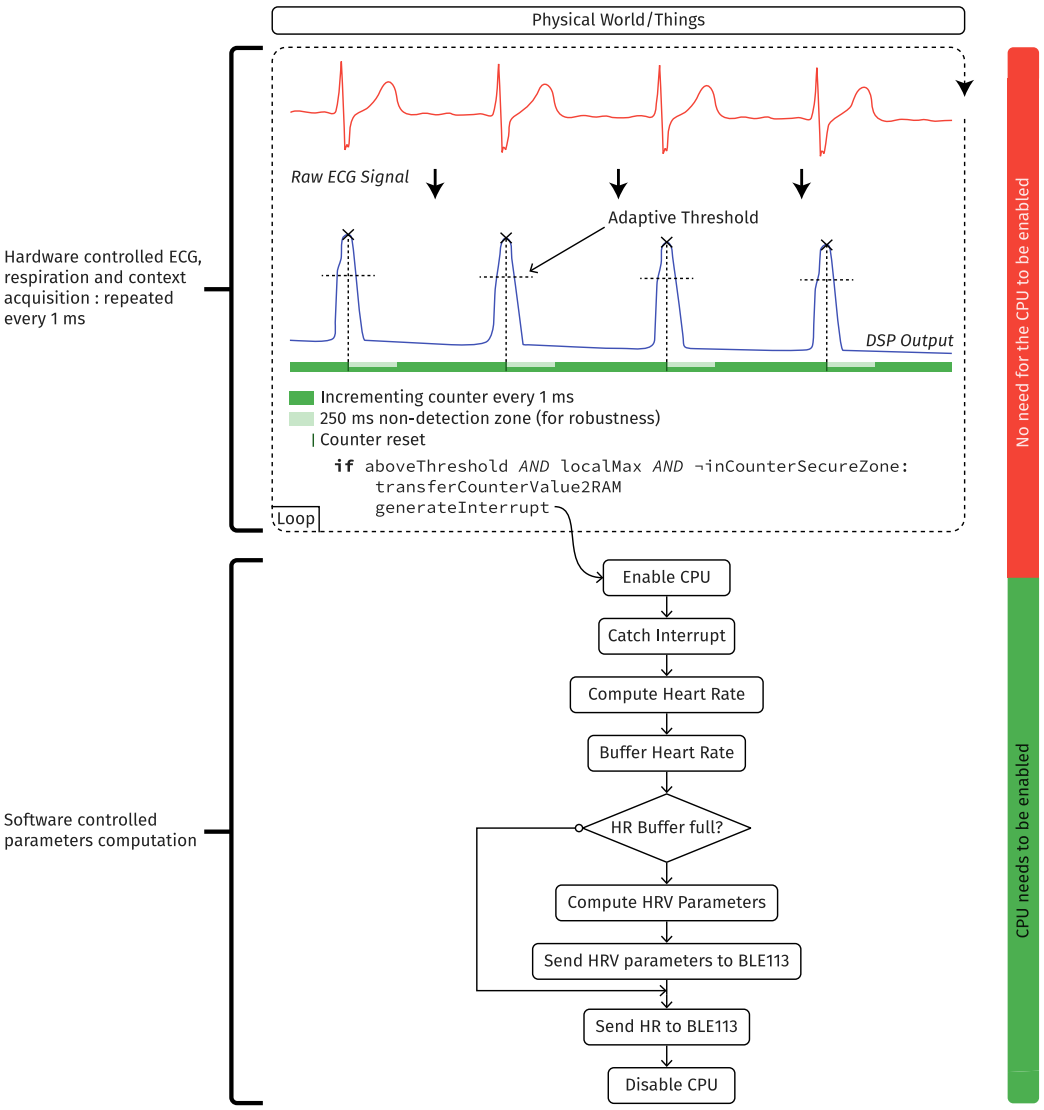


Figure 4.4 – Hardware vs software signal processing

where $a[n]$ is the output of the moving average filter at sample n, $x[\cdot]$ is the input and N_a is the filter's order.

Please note that the original implementation use weighted versions of the differentiator and the moving average, but we chose to utilize unweighted versions because the dynamic range of the DFB is wide enough (i.e., 24-bit) with respect to the fastest variation of the ECG signal. Additionally, the AFE features a remotely-configurable gain, which can be adjusted to correct overflow in extreme cases. Finally, the final equation implemented on the DFB can be summarized as:

$$y[n] = a\,[s\,[d\,[x\,[n]]]] \tag{4.4}$$

where $y[\cdot]$ is the output of the DFB.

After empirical experimentation using the first version of the prototype (depicted in the previous chapter), we concluded that the best order for our specific use-case were:

$$\begin{cases} N_d = 9 \Leftrightarrow 9 \text{ ms} \\ N_a = 64 \Leftrightarrow 64 \text{ ms} \end{cases} \tag{4.5}$$

Such coefficients were carefully selected in order to avoid traditional pitfalls of poorly configured Pan and Tompkins algorithms, such as the presence of multiple slope inversions during the peak produced by the algorithm, or the merging of the T wave with the *QRS* complex [PT85].

Practically, we implemented the equations introduced above using the application-specific assembly-like language provided by Cypress Semiconductor. An informal representation of the DFB's output in response to a typical ECG signal is given in blue in Figure 4.4. Please note that even though the raw ECG signal is provided as a signed 24-bit integer and the DFB operates on 24-bit data, we only use the 8 most significant bits of the output to perform peak detection. This technical choice was motivated by the fact that the processing of 8-bit data by the programmable logical gate matrix consumes less energy than the processing of the full 24-bit signal because fewer gates need to be powered to perform the hardware signal processing operations.

The next step of the Pan and Tompkins algorithm is peak detection, which can be used to determine RR intervals (defined as the time interval between two peaks). To this end, we implemented a hardware-only solution using the PSoC5LP's programmable logic gate matrix because this approach minimizes microcontroller wake-up time and logical gate feature minimal power consumption. The diagram of our solution is given in Figure 4.5, but is also verbosely detailed in Figure 4.4. Peak detection is straightforward and is based on a set of three simple conditions, which must be simultaneously verified in order for a heart beat to be detected: the current output of the DFB must be directly less than the anterior output (indicating a change in the output's slope sign), the same current output must be above an adaptive threshold, and a counter must be above a certain value (in our case 250). The first condition is relatively trivial, and denotes a local maximum in the DFB signal. The threshold condition makes peak detection more robust to motion artifacts, which can introduce low-amplitude local peaks on the filtered ECG signal. Eventually, the counter is a key element of this hardware peak detection as it determines the actual value of the RR interval. It is incremented every one millisecond, and its output upon peak detection corresponds to the time interval (in millisecond) between the two R peaks because it is reset after each peak detection. The counter output is transferred to the microcontroller using a Direct Memory Access (DMA) channel (illustrated as the DMA1 block on Figure 4.5) at every peak detection (before counter reset). The comparison with a value of 250 prevents the consecutive detection of unrealistically close peaks (250 ms corresponds to 240 BPM), consequently improving overall robustness. Additionally, the sensor is configured to automatically reset after 2.5 s. The comparators, *and* logical gate and counter are all implemented as hardware blocks using the graphical Integrated Development Environment (IDE) provided by the PSoC5LP's manufacturer.

As illustrated in Figure 4.5, the threshold value is adaptive. Our motivation for choosing adaptive threshold over fixed threshold came from the observation of important variations on the DFB's output amplitude. Such variation are typically caused by vari-

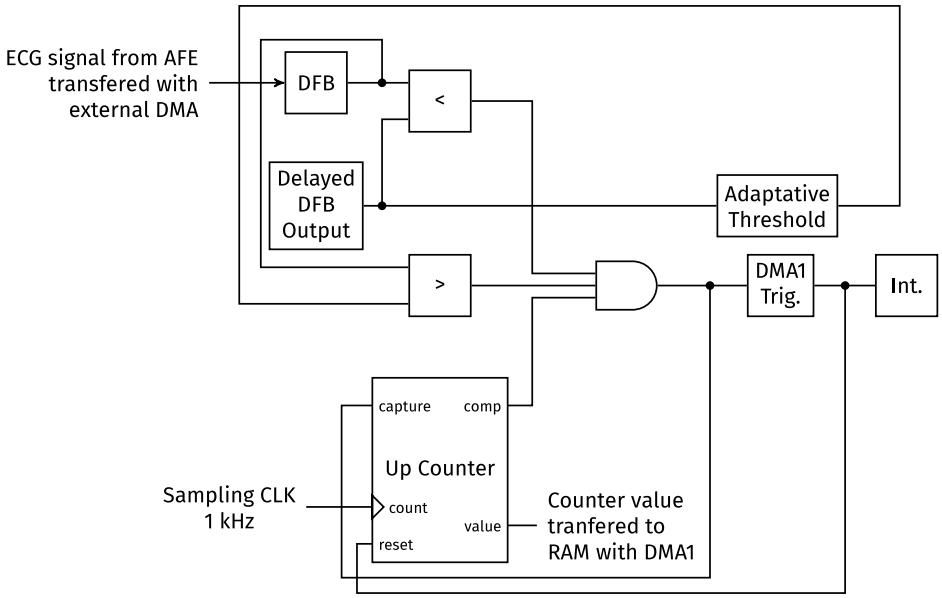


Figure 4.5 – Diagram of hardware-based RR interval detection

ous physical phenomena: skin-electrode contact impedance variations, evolving skin conductivity, etc. These phenomena are further increased by long term monitoring, mandating the use of an adaptive threshold approach, which is implemented as an exponential filter [Mas+15]:

$$t[n+1] = \alpha \cdot \beta \cdot p[n] + (1-\alpha) \cdot t[n] \tag{4.6}$$

where $t[\cdot]$ is the threshold value, α such that $\alpha < 1$ is the filtering coefficient, and β is the scaling factor for current peak $p[n]$. Practically, we chose $\alpha = 0.5$ and $\beta = 0.5$. We implemented the hardware adaptive threshold using the Verilog Hardware Description Language (HDL), which is subsequently synthesized as a hardware arrangement of logical gates when compiled.

As a consequence, of real-time signal processing is performed either using the dedicated signal co-processor (i.e., the DFB) or the programmable logical gate matrix, and the PSoC5LP's microcontroller is put in an energy-saving sleep mode during most signal processing operations. The ARM Cortex-M3 core is only activated through the interruption depicted in Figure 4.5 when a heart beat is detected. The interrupt service routine associated with peak detection only sets a flag, which is then handled appropriately by the main program (depending on the non-functional state of the sensor, which will be described later in this chapter).

Software signal processing is in charge of the computation of the four HRV parameters, but also the conversion of RR intervals into a HR value in BPM. The calculation of the HR is straightforward:

$$HR = \frac{60,000}{RR_{\text{int}}} \tag{4.7}$$

where HR denotes the Heart Rate in beats per minute, while RR_{int} is the RR interval in milliseconds which is transferred from the hardware counter to the microcontroller's RAM using a DMA channel. This operation is performed by the microcontroller and is implemented using C code.

The computation of HRVs parameters is more complex, and requires advanced signal processing operations. In opposition to heart beat detection and HR calculation, HRV parameters must be computed offline, meaning we need to have a history of HR values in order to process them. Practically, we use a buffer of the last 5 minutes of HR recordings, following the clinical recommendations of [Tas96]. The buffer was statically allocated with a size of 2048 values, which improves our smart device's robustness to segmentation faults. Indeed, considering the minimal RR interval is set to be 250 ms by the signal processing hardware, we need at least 1,200 values to represent a 5 minutes buffer and to prevent potential buffer overflow. Eventually, HRV parameters processing occurs when the buffer contains 5 minutes of HR values, and the relevant number of samples within the buffer vary since the HR is a continuously-evolving physical parameter.

The computation of the time-domain parameters is trivial, and we used traditional formulas to compute the standard deviation of RR intervals and the quadratic mean of differences between consecutive RR intervals. The processing of frequency-domain parameters is more complex, and it requires computationally intensive time-to-frequency domain conversion. First, HR data is not evenly sampled because heart beats occur at different intervals. Practically, this means we can not use traditional PSD estimations such as the Fast Fourier Transform (FFT). Methods using interpolation as means of obtaining an evenly sampled signal followed by a FFT have been proposed, but they are known to introduce high-frequency distortions on the PSD, which introduces errors in the computation of HRV parameters [CT05].

The calculation of the PSD of unevenly sampled data was studied by Lomb [Lom76], and further improved by Scargle [Sca82]. This method is a modified version of the classical periodogram, and it introduces specific weights along with a phase shift τ in order to provide a PSD estimate for this category of data. The expression of the Lomb-Scargle periodogram is defined as [Sca82]:

$$P_X(\omega) = \frac{1}{2}\frac{\left[\sum_j X_j \cos\left(\omega\left(t_j-\tau\right)\right)\right]^2}{\sum_j \left[\cos\left(\omega\left(t_j-\tau\right)\right)\right]^2} + \frac{1}{2}\frac{\left[\sum_j X_j \sin\left(\omega\left(t_j-\tau\right)\right)\right]^2}{\sum_j \left[\sin\left(\omega\left(t_j-\tau\right)\right)\right]^2} \tag{4.8}$$

with τ defined as:

$$\tan(2\omega\tau) = \frac{\sum_j \sin\left(2\omega t_j\right)}{\sum_j \cos\left(2\omega t_j\right)} \tag{4.9}$$

The weights mentioned above correspond to the denominators of each term of the formula. The Lomb-Scargle periodogram is equivalent to the classical periodogram if data is evenly sampled. Even though this method can precisely calculate the PSD of the HR data, its naive implementation computationally intensive and requires complex trigonometric computations. Nevertheless, literature showed that the Lomb-Scargle periodogram can be computed as fast as the processing of two FFT [PR89b], and we adopted this approach for our implementation of the time-to-frequency domain conversion. Once the PSD is computed, frequency-domain HRV parameters can be trivially calculated using sums and ratios.

Considering the computations HRV parameters are only performed periodically using buffered RR intervals, we do not consider them as real-time signal processing. These operations are however computationally intensive, as they require the use of floating-point operations. Additionally, the ARM Cortex-M3 embedded in the PSoC5LP does not feature a floating-point unit, which considerable slows down the computation of the parameters, resulting in a processing time of about 2 seconds [Mas+16].

The last signal software signal processing operation is the filtering of the RWF. Practically, our smart device buffers the respiration signal in the PSoC5LP's RAM and transmits it every second. Because real-time streaming of 24-bit data at a rate of 1 kSPS (which is the sampling rate of the RWF) would consume too much energy considering the targeted battery life, we chose to downsample the RWF to 6 SPS. This new sampling frequency of 6 Hz results in a Nyquist rate of 3 Hz, which is within the bandwidth of typical respiratory signals (the upper bound of this bandwidth is 1.5 Hz) [ZRF94]. Upon downsampling, the RWF is low-passed filtered using an exponential filter:

$$y[n] = (1-\alpha)\,y[n-1] - \alpha\,x[n] \tag{4.10}$$

where $x[n]$ represents current sample, and $y[\cdot]$ designates the output of the filter. The parameter α is used to control the smoothing strength, and we chose $\alpha = 0.5$ for our smart device.

In summary, our sensor features advanced on-board signal processing performed either using pure hardware mixed with the DFB, or using C software and the ARM Cortex-M3 microcontroller. The signal processing is modularized according to the design method specified in the previous chapter. Indeed, we separate real-time and *offline* signal processing, which constitute two separate modules in the architecture of the sensor. Such signal processing is used to only transmit relevant information about HR and HRV parameters instead of streaming the raw ECG data. All these modules can be formally specified and verified through model checking, as mentioned in the service-oriented design method. Additionally, HR and HRV parameters are exposed to external tiers using BLE services. These services can be read remotely, but they also are associated with notification channels which transmit new data as soon as it is available. Similar observation can be drawn from the RWF.

4.3.3 Integrating Self-Adaptive Behavior

As mentioned here-in above, self-adaptive behavior is a non-functional requirement of our smart device. The integration of self-adaptation capabilities, along with on-board signal processing, is a technological enabler of smarter IoT devices. Our smart device must thus be able to measure internal and external characteristics in order to take appropriate action, but it must also feature the ability to be integrated in wider-scale self adaptive frameworks.

To this end, we listed two principal non-functional characteristics that would benefit self-adaptive behavior: the detection of sensor attachment (i.e., our smart device should be able to detect if it is connected to electrodes on patient), but also the monitoring of battery charge level. Sensor unattachment detection was trivially implemented using internal capabilities of the AFE which was selected as the ADC of our device. Practically, a small electrical current (in the range of a few μA) is injected from one electrode, and

returns to the measurement circuitry if all electrodes are properly connected, thus having little impact on ECG measurement [Cal15]. However, if an electrode is not properly connected, the injected current directly feeds the ECG amplifier, which results in a lead-off detection. In the ADS1292R, this unattachement detection is transmitted through the first 24-bit frame of the SPI bus, which is processed using hardware in order to generate an interrupt notifying the microprocessor of electrode unattachement.

In terms of battery level monitoring, we chose to use a voltage measurement associated with a lookup table for voltage-to-percentage conversion in order to decrease PCB surface and energy consumption. Battery level is measured just before HRV parameters processing, every 5 minutes, which prevent unnecessary wake-ups of the microcontroller.

From both these parameters, we implemented a self-adaptive Labeled Transition System (LTS) detailed in Figure 4.6. Transitions between the different states of the LTS are either controllable or non-controllable. The first category designates transitions which can be remotely triggered (i.e., they can be used in global self-adaptive scenarios by external controllers), while second category defines transitions which are automatically triggered by the sensor depending on its internal state and context. As a convention, a transition written as "`a \ b`" describes the internal triggering of the transition on internal event `a`, or the external triggering of the transition caused by the remote call of service `b`. When a transition is only labeled with a single event, we assume that this event is internal. More theoretical details about the LTS model of computation will be detailed in the next chapter.

Practically, our smart devices comprises 5 states:

- *Initialization*: This state constitutes operations performed during bootstrap start-up of our smart device. Namely, we send appropriate configuration from the ARM Cortex-M3 microcontroller to the ADS1292R AFE. A software reset operation is also performed on the BLE113 communication-oriented SoC. The sensor is automatically placed in the *normal* state upon initialization completion.
- *Normal*: This represents the default state of the sensor. In this state, both ECG and RWF are sampled and processed. HR values are transmitted to external tiers as soon as they are computed, and the HRV parameters are calculated every 5 minutes. Additionally, the RWF is processed and sent every second.
- *Failsoft*: This is the energy-saving state of our sensor. This state is accessed either on external request, or if the battery level is less than 20 %. In this state, HRV parameters processing, lead-off detection, and RWF acquisition are disabled in order to decrease power consumption. In addition, HR is not transmitted at every heart beat anymore, but it is internally buffered during 5 minutes. When the buffer is full, it is averaged and this value is transmitted to external tiers. The decrease in transmission frequency causes energy savings, as wireless data transfer implies important peak consumption during transmission periods.
- *Unattached*: The smart device is placed in this state if a lead-off event occurs. Please note that it can only be accessed from the *normal* state as lead-off detection is disabled in all other states. Upon state access, a timer is launched, an is used to place the sensor into the *stop* state after 1 minute of smart device unattachement.
- *Stop*: Eventually, the last sensor state designates sensor power-off. In normal use, this state should be avoided as long as possible as it can be only exited though hardware reset (i.e., a press on the hardware reset button). It is accessed if the

battery reaches a critically low level (namely, 5 %), or if the sensor is not connected to any BLE device for more than 1 minute. The sensor can also be externally placed in the stop mode for convenience reasons. Please note that in this state, the sensor is only placed in a deep-sleep mode rather than a complete off mode. The stopped state can also be used to save energy when the sensor is not in use.

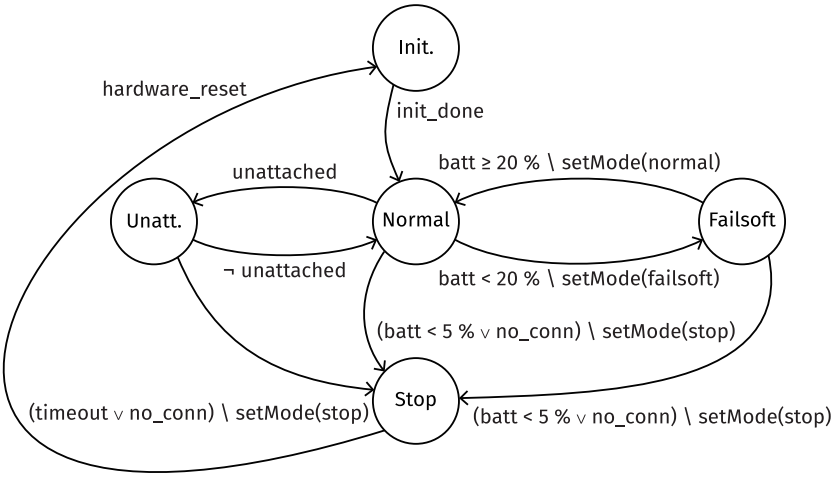


Figure 4.6 – Cardiorespiratory sensor Labeled Transition System

We implemented this LTS on the PSoC5LP's ARM Cortex-M3 microcontroller, and events are either generated internally (from interrupts), or forwarded by the BLE113 in the case of external remote state change request.

Please note that the AFE gain can be changed in the *normal* and *failsoft* states, improving the flexibility of our smart device. Indeed, cross-patient variability impact amplitude of the DFB signal, and optimization can be performed at run time though the remote configuration of the AFE gain. This value is stored in the non-volatile PSoC5LP's EEPROM, and it is accessed and read at every sensor hardware reset. The overall firmware size (i.e., ROM usage) is 29,312 bytes with a RAM memory occupation of 30,625 bytes.

This concludes our discussion of the integration of the self-adaption non-functional properties to our smart device. Additionally, the LTS approach introduces negligible processing overhead, making it particularly efficient when the increase in energy consumption to added value ratio is considered. Since this self-adaptive behavior only reconfigures various modules of the hybrid modular architecture, it does not impact the medical-grade precision of the device.

4.3.4 Development of an Android Companion Application

In order to facilitate experimentations based on our sensor, we developed a companion Android application. Indeed, Android phones represent ideal gateways to perform mobile performance assessment of our smart device. Additionally, smartphones are continuously connected to the Internet, which is ideal for real-time remote data collection. This Android application can connect to the sensor, but it also plots collected data and stores it on the phone memory. Our application is based on the nRF Toolbox

open-source application [1] developed by Nordic Semiconductor (Oslo, Norway), and a screenshot of the application is given in Figure 4.7.

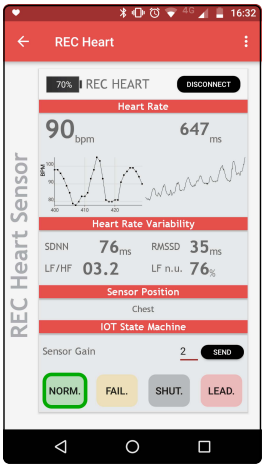


Figure 4.7 – Screenshot of the developed Android companion application

Even though the signal processing modules developed in previous sections are specified and verified in the methodological framework, it is necessary to perform various testing in order to provide additional guarantees on the functional and non-functional properties of the final product, and these various testing processed are detailed in the next section. Our application acts as a facilitator for testing procedures, as it can be used to easily connect to the sensor and collect data.

4.4 Sensor Evaluation

Although the design methodology emphasizes the formal specification and verification of service-oriented smart devices, testing is still required for various reasons. First, formalism is only introduced at the specification level, which does not provide guarantees over actual implementation. Indeed, system designers can still make errors during devices' implementation. Then, numerous non-functional properties are difficultly verifiable using traditional model checking tools, which mandates further testing in experimentation in order to practically verify these properties. We consequently performed various testing of both the preliminary and final prototypes of our smart device, and the process is detailed bellow.

4.4.1 Evaluation on Synthetic Signals

The first experiment, performed using the preliminary prototype depicted at the end of the previous chapter, was used to perform acquisition on synthetic ECG signal. Indeed, the testing procedures specified in the design method mandate the study of devices assigned to the sensor ontological class against reference physical signals. In our case, this is simplified by the fact ECG signal is already an electrical signal, and we must only verify that provided a reference synthetic ECG, our sensor provides the expected results.

1. nRF Toolbox open-source application sources: `https://github.com/NordicSemiconductor/Android-nRF-Toolbox` (visited on 02/23/2018)

To this end, we generated a realistic ECG signal using an Agilent 33220A (Santa Clara, CA, USA) arbitrary waveform generator. The advantage of this approach is we can comprehensively parameter the generated signal, and we can consequently quickly verify various smart device parameters. This first experiment consisted in the simulation of heart rate variability using triangular frequency modulation to the synthetic ECG signal. The parameters were:

$$\begin{aligned} \text{base frequency} &= 1.25 \text{ Hz} \\ \text{frequency deviation} &= 200 \text{ mHz} \\ \text{frequency sweep} &= 50 \text{ mHz} \end{aligned} \tag{4.11}$$

which results in RR intervals ranging between:

$$\begin{aligned} RR_{\text{int}} &< \frac{1000}{1.25 - 0.2} \approx 952.4 \text{ ms} \\ \text{and, } RR_{\text{int}} &> \frac{1000}{1.25 + 0.2} \approx 689.7 \text{ ms} \end{aligned} \tag{4.12}$$

Results from this experiment are displayed in Figure 4.8. As expected, measured RR intervals are between 688 and 953 milliseconds, and the time between two consecutive peaks is 20 seconds, which corresponds to the sweep frequency of 50 mHz.

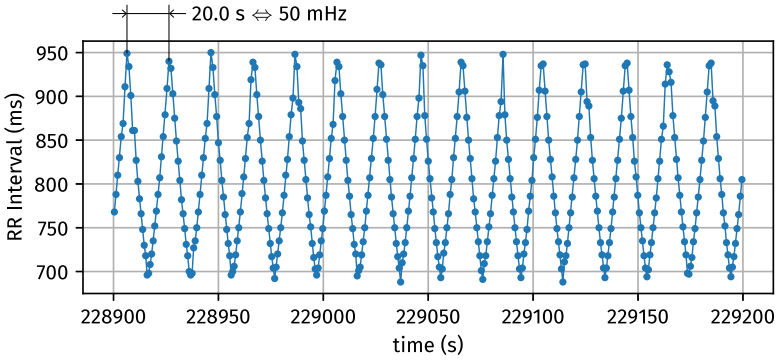


Figure 4.8 – Sensor evaluation using synthetic ECG

Synthetic ECG was also used to perform connectivity robustness testing. To this end, we continuously collected data during 1 day, 17 hours and 9 minutes to test for BLE disconnections. To log the event, we used a Raspberry Pi and a simple Python script. A total of 185,153 HR values were recorded. We performed simple data analysis to verify for data points outside of the theoretical range mentioned above (with a tolerance of 1 % over the limits), and we found that no record did violate these bounds.

4.4.2 Energy Consumption

In order to verify the battery life non-functional property, we measured our sensor's power consumption. Measurements were carried out using a Keithley 2400 sourcemeter (Beaverton, OR, USA). We first performed a static power consumption evaluation of the *normal*, *failsoft* and *stop* states, ignoring the *initial* state because it is only accessed after hardware reset for a brief amount of time, but also discarding the *unattached* state because the sensor is in the same configuration as in the *normal* state. This static characterization was achieved using default Android 7.0 BLE connection parameters

(i.e., connection interval of 48.75 ms, timeout of 20 s and latency of 0), and results are displayed in Table 4.2. The measured power consumptions imply a 75 hours battery life in the normal state and a 85 hours battery life in the *failsoft* state (assuming a 300 mAh battery). In the stop mode, the sensor can last more than two months on a fully charged battery.

Table 4.2 – Static power characterization of IoT states

IoT state	Power consumption (mW)
Normal	10.53
Failsoft	9.18
Stop	0.505

We also studied the influence of the BLE connection's parameters in order to deepen our understanding of the overall power consumption of our smart device. The BLE standard describes three connection parameters: interval, latency and timeout. First, the connection interval defines the period between two requests from the BLE master to the BLE slave (in our particular case, the BLE master is the phone or gateway our smart device connects to, which is in turn considered as a slave). Then, the latency parameter characterizes the number (given as an integer) of connection intervals that can be safely ignored by the slave. Finally, the connection timeout parameter describes the time period after which the BLE master will consider the connection to the slave as lost, and it is only after that period has elapsed that the master can attempt a reconnection to the lost slave. For our smart device, we chose to fix connection timeout and connection interval. Data was collected using a Raspberry Pi, because its BLE implementation always accepts changes of the connection parameters by the BLE slave (in opposition to Android phones, which tend to force default values for the connection parameters). After fixing the connection interval and timeout, we progressively changed the latency parameter in order to measure its influence on the global power consumption of our device.

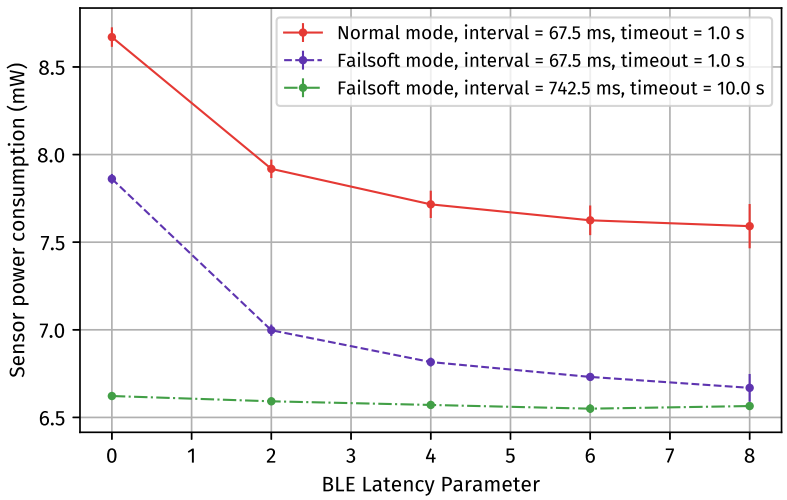


Figure 4.9 – Impact of the BLE latency parameter on global power consumption

The result of this experiment are given in Figure 4.9, where the consumption of the *normal*, *failsoft* and *stop* states are plotted against latency parameter value. For all

states, an increase in the latency parameter value resulted in a decrease in the global power consumption, with a sharper diminishing for smaller latency values. Please note that for a latency value of 8, we observed connection instability (meaning the sensor would at time disconnect from the BLE master). In order to reduce power consumption while preserving acceptable connectivity stability, we configured the BLE113 SoC so that it requests a latency value of 2 when a new connection is established. Nevertheless, BLE masters such as Android phones can refuse connection parameters changes requests, and set their own set of parameters, which usually cause higher power consumption because of smaller connection intervals, latency parameters and connection timeouts. This problem was tackled by a slight oversizing the battery capacity in order to guarantee at least 48 hours of battery life.

4.4.3 Sensor Precision Evaluation

One of the most important test when it comes to the design of medical-grade smart devices is sensor precision evaluation. The term precision is however relatively vague, and we must consequently define quantitative characteristics that we will use to evaluate our smart device. In this section, we consider precision as defined by the information retrieval (i.e., in terms of true positives and negatives, but also false positives and negatives), and a formal definition is given bellow.

Sensor Sensitivity and Precision Evaluation

Our first experiment consisted in the evaluation of optimal electrodes' placement for our smart device. Indeed, the mobile nature of our sensor mandates the study of electrodes' placements in order to propose the placement resulting in minimal movement artifacts, which is particularly important for real-life use. All the evaluated placements are depicted in Figure 4.10.(a.). We chose to evaluate the Left Arm (LA) - Right Arm (RA) derivation because of the emergence of patch-like form-factors in mobile ECG sensors, and such sensors are traditionally worn by patients on their upper chest area. The Left Leg (LL) - RA placement coincides with the standard lead-II derivation of traditional 3-electrodes ECG devices. Finally, the last derivation to be tested is referred as the "abdominal configuration," where the medial electrode is placed in proximity to the lower limit of the xiphoid process. The lateral electrodes are then placed 5 cm from each side of the central electrode.

We also performed a second experiment aimed at the evaluation of the influence of the electrodes' nature of measurements quality. We evaluated three categories of electrodes: ConMed Softrace Large electrodes (also used in the first experiment), Ambu (Ballerup, Denmark) Blue Sensor L electrodes and a Polar (Kempele, Finland) dry electrodes belt. This experiment is displayed in Figure 4.10.(b.).

As medical-grade precision is required, we need gold standard instrumentation to collect reference ECG signal. This signal can then be used as a baseline our sensor should be tested against. For both experiments, the reference ECG was recorded using an an ADInstruments (Sydney, Australia) PowerLab 26T data acquisition apparatus using the standard LL-RA derivation. Reference ECG was recorded at a 4 kSPS sampling rate, and we processed this signal using a bandpass filter (bandwidth: 5 to 20 Hz) in order to remove high- and low-frequency artifacts.

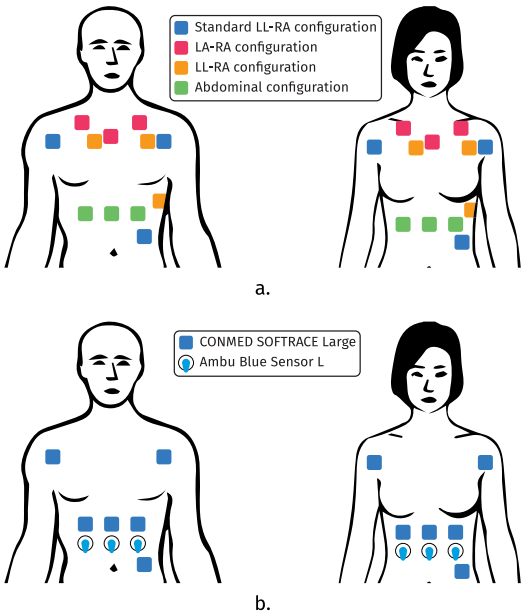


Figure 4.10 – Electrodes' placement (a.) and nature (b.) evaluation

For these two experiments, reference ECG and smart device data were collected on 8 consenting voluntary subjects. Three of the subjects self-reported as females, while 5 of the subjects self-reported as males. Ages ranged from 22 to 47 years (average: 31 years). Subjects were requested to complete the same three-minutes-long exercise. During the first minute of the experiment, they performed a baseline rest period where subjects had to sit still. For the second minute, they stood still, and for the third and last minute of the experiment, subjects sat back down and carried out a series of arms rotations in the coronal (or frontal) plane in order to test our sensor's robustness to movement artifacts.

Our sensor was quantitatively evaluated in terms of sensitivity and precision. The sensitivity is defined as the ratio between true positives and the sum of true positives and false negatives, while the precision is the ratio between true positives and the sum of true positives and false positives. In our specific application, true positives are characterized as heart beats correctly identified as such. False positives are defined as heart beats detected by our sensor while no heart beat is detected in the reference ECG signal. Finally, false negatives are heat beats present in the referenceECG missed by our smart device. Heart beat detection on the reference ECG signal was achieved using the following conditions:

$$\text{Peak detected} \iff \begin{cases} x_{n-1} - x_n < 0 \\ x_n > x_{\text{thres}} \\ t_m - t_{m-1} > 250 \text{ ms} \end{cases} \tag{4.13}$$

where x_n denotes the reference ECG signal, x_{thres} designates a manually-adjusted and predefined threshold (manual adjustment was performed for each subject, as cross-subject variability introduces variations in the ECG signal amplitude). Finally, t_m represents time occurrences of the detected peaks. The first condition coincides with a local maximum in the ECG signal, and the second condition is utilized to detect R peaks

in this signal. Eventually, the third condition enhances peak detection's robustness by rejecting consecutive peaks separated by less than 250 ms. After ECG signal analysis, RR intervals can be computed and compared to RR intervals measured by our smart device.

In summary, we extracted and manually validated a total of 9,428 heart beats from reference ECG apparatus, which we subsequently compared to RR intervals from our sensor. We calculated true positive, false positive and false negative rates in order to estimate sensor sensitivity and precision, and results from both experiments are given in Table 4.3. It is worth noting that, except for the LA-RA configuration, all tests produced sensitivities and precisions higher than 99 %. Optimal electrode placement is the abdominal position using ConMed Softrace Large electrodes, which features a precision of 99.89 % and a sensitivity of 99.95 %. Based on these measurements, we promote the use of the abdominal positioning of our sensor.

The LA-RA placement yielded in poor results both in terms of precision (89.39 %) and sensitivity (94.57 %), and this points towards the fact that patch-based form-factors for ECG sensors may result in unaccurate measurements. The robustness of our sensor with respect to motion artifacts is probably caused by electromyographic interference introduced by movements. Indeed, the negative effect of electromyographic signals on the ECG is well-known, and attempts at artifact reduction can be found in the literature [CD99; JVT13]. Nevertheless, these methods can commonly be applied only on full (i.e., offline) ECG signal, rendering their embedded implementation difficult and resources-consuming. Poor results or the LA-RA placement is likely caused by the proximity of pectoral muscles when our sensor is placed in this position. These muscles introduce strong electromyographic signal, which in turns introduces errors in heart beat detection, consequently increasing false positives and negatives. This can be corrected by placing the sensor lower on the rib cage, such as in the abdominal position, because muscle thickness is typically lower than in the LA-RA case.

Table 4.3 – Precision and sensitivity of the sensor

Configuration	Sensitivity	Precision
LL-RA	99.47%	99.89%
LA-RA	94.57%	89.39%
Abd. (I)[1]	99.95 %	99.89 %
Abd. (II)[2]	99.89 %	99.84 %
Abd. (III)[3]	99.47 %	99.31 %

[1]Abdominal position using ConMed electrodes
[2]Abdominal position using Ambu electrodes
[3]Abdominal position using Polar dry electrodes belt

We found that the abdominal placement yielded in the best results, with very little impact of the electrodes' nature (maximal differences between best and worst performance in terms of sensitivity and precision are respectively of 0.48 % and 0.58 %). This concludes our evaluation of the influence of electrodes' nature and placement on our smart device performance in terms of sensitivity and precision.

Quantitative Measurements Agreement Evaluation

The principal drawback of the sensitivity- and precision-based approach is it does not quantify the agreement between gold-standard instrumentation and our smart device.

Agreement is here used to define the quantitative difference between measurements from two different apparatus measuring the same physical phenomena. To measure agreement between our sensor and reference instrumentation, we performed an additional experiment where a single subject was requested to sit still during 10 minutes, and we collected both reference ECG signal and RR intervals from our sensor. Our smart device was placed in the abdominal position using Ambu Blue Sensor L electrodes. Data was processed using the same algorithm as in the previous experiments, a Bland-Altman plot was created from the collected measurements, and it is displayed in Figure 4.11.(a.).

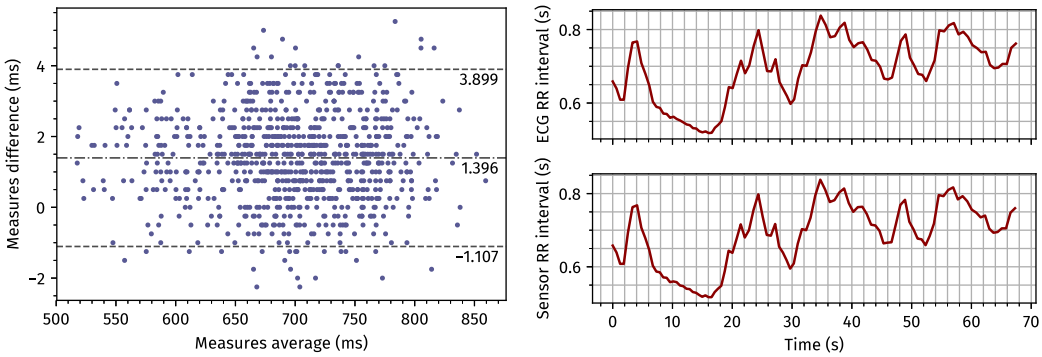


Figure 4.11 – Bland-Altman diagram (a.) and comparative tachograms (b.)

For illustrative purposes, we also graphed tachogram using data from the reference ECG and from our sensor in Figure 4.11.(b.), and one can notice there is the two plots look identical. Getting back to the Bland-Altman plot in Figure 4.11, one can observe that the greatest positive difference is of 5.25 ms for an average measurement of 783.6 ms, and that the greatest negative difference is of −2.25 ms for an average measurement of 668.9 ms. Both these differences results in a relative error smaller than 1 %. It is worth noting the presence of an average positive shift of 1.396 ms in measurements difference, which can be explained by the precision of the internal clock of the AFE used as sampling frequency, which is of only 1.5 % on the IC's standard temperature range. We used this internal clock rather than a more precise external clock for energy saving and PCB surface minimization purposes.

In conclusion, we quantitatively evaluated our smart device's agreement against a reference gold-standard instrumentation. Results were satisfactory, as we observed a maximal relative error of less than 1 % between the two measurement apparatus. We are thus confident in our sensor's capabilities to deliver medical-grade HR estimations.

4.4.4 Mobility Evaluation

Our final experiment dealt with the evaluation of the performance of our sensor in ambulatory conditions. This was driven by the mobility non-functional property listed during the preliminary steps of our service-oriented design method. To verify such property, a healthy patient was asked to wear our smart device during one hour while performing normal daily activities. Sensor data was recorded using the Android companion application described above.

The collected RR intervals and normalized RWF are given in Figure 4.12, and no measurement artifact occurred during the whole experimentation as no abnormal

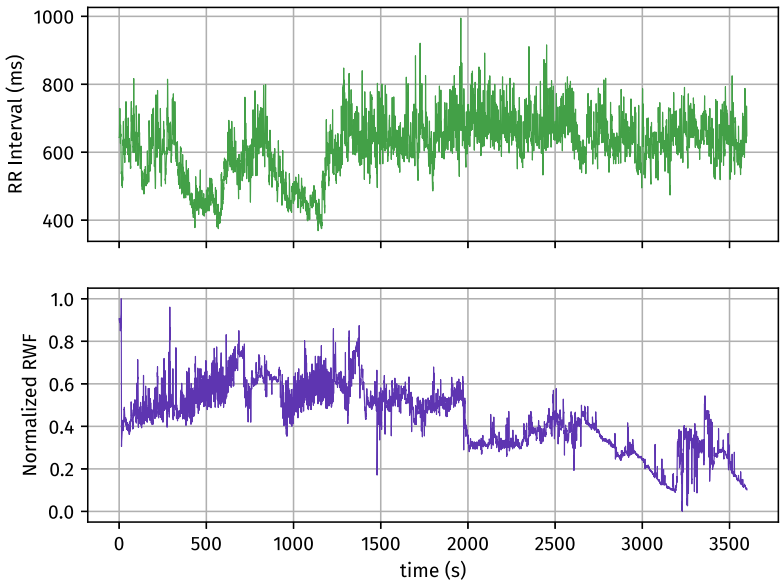


Figure 4.12 – Ambulatory HR and RWF

values were observed (i.e., biologically aberrant values). Please note that during this test, the sensor was placed in the abdominal position in order to minimize the chance of movement artifacts. Additionally, one may notice the respiration signal accounts for a slowly-evolving (i.e., low frequency) component. This is caused by the fact RWF measurements are using impedance, which slowly evolves over time because of physical changes occurring at the electrode-skin interface. Such alterations are induced by various phenomena such as the presence of sweat, ambient humidity, etc. The low-frequency component can be corrected with offline high-pass filtering.

Furthermore, Figure 4.13 depicts the four HRV parameters plotted as a function of time. These parameters coincide with HR measurements displayed in Figure 4.12, and fall within the typical ranges defined by the literature [NSB10].

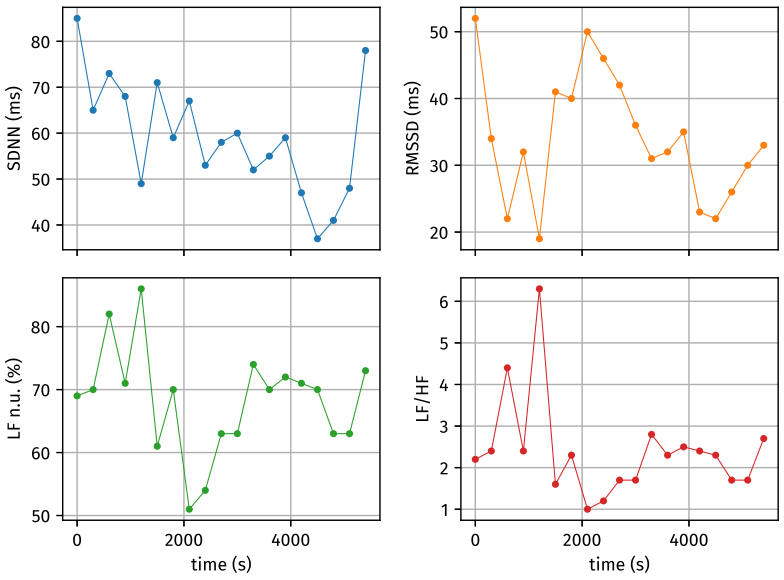


Figure 4.13 – Ambulatory HRV parameters

This concludes the comprehensive testing of our sensor with respect to various non-functional properties. We measured a variety of parameters such as energy consumption, response of the sensor to synthetic ECG signal, and we evaluated our smart device's precision in order to guarantee medical-grade measurements. This testing process provides additional assurances about overall sensor behavior, and can help to detect implementations errors.

4.5 Conclusion

This chapter detailed the application of our service-oriented design method to a cardiorespiratory sensor. From the modular hybrid architecture described in the previous chapter, we built hardware and software implementing the formally specified hybrid modules. A careful selection of appropriate ICs and SoCs was performed in order to simultaneously verify functional and non-functional properties. We also put a particular emphasis on the development of embedded signal processing algorithm, and used low-level application specific hardware to perform real-time signal processing, along with software algorithm to carry out higher-level offline calculations of the HRV parameters.

Eventually, we extensively described the testing of our smart device. Power consumption measurements resulted in a maximal consumed power of 10.52 mW when connected to an Android phone using default BLE connection parameters, which results in a battery life of more than 3 days. Additionally, we used synthetic ECG signal in order to verify both measurement conformance with expected results, but also BLE connection robustness. Testing was further performed in order to determinate optimal electrodes' placement and nature, but also to quantitatively measure our sensor performance against gold-standard medical-grade ECG measurement instrumentation. With a maximal sensitivity of 99.95 %, precision of 99.89 %, and relative error of less than 1 %, we are confident that our sensor can provide medical-grade estimations of HR and HRV parameters. We also evaluated our smart device performance during real-life ambulatory conditions, and no artifact were observed during this experiment.

However, our smart device presents several limitations. For instance, flexibility could be improved by the addition of Over-the-Air (OTA) update capabilities. OTA updates designate the capability for a smart device to be remotely updated, which in terms of implementation designates the capability to wirelessly transfer a new version of the smart device's firmware. To achieve this, a bootloader needs to be implemented so that the microcontroller can either be booted using its classical firmware or using an update mode, which typically receives the new firmware using a classical serial communication bus. Adding a bootloader to a microcontroller implies reserving a portion of program memory (i.e., ROM) to the bootloader, which reduces the amount of available memory for the principal firmware. Particular attention must be paid when developing such feature as OTA update can have disastrous consequences if an error occurs, and it can ultimately result in unusable smart devices.

Furthermore, we discuss the integration of internal and external self-adaptation capabilities as a non-functional property of our smart device. The integration of such feature was implemented through a simple embedded state machine, which is used to describe various sensor configurations associated with different Quality of Service (QoS). Indeed, self-adaptive behavior at the lower levels can be seen as a preliminary form of

device intelligence, and it facilitates integration of smart devices to higher-level smart services. Even though we integrated internal self-adaptive behavior which makes our sensor self-aware about its environment, self-adaptation is only fully leveraged when it is also considered from a higher level perspective. This higher-level self-adaptation mandates the study of adaptive framework applied to the IoT, and we extensively detail the integration of smart devices to such framework in the next chapter.

CHAPTER 5

Self-Adaptation Framework for Smart Devices

Contents

5.1 Introduction 97
5.2 Self-Adaptation Case-Study 100
5.3 Dynamic and Synchronous QoS-Driven Self-Adaptation for the IoT . . 101
5.3.1 Managing Dynamic Objectives and Monitoring Infrastructure . 102
5.3.2 Synchronous Programming Languages 106
5.4 Adaptation Objectives Specification 111
5.4.1 Declarative Self-Adaptation Specification 111
5.4.2 Specifying Adaptation Objectives with a Rule-Based Language . 113
5.5 From Synchronous to Hybrid Self-Adaptation 116
5.6 Conclusion 118

5.1 Introduction

IoT-based systems are, by nature, in continuous interaction with the physical world. As physical phenomena are in constant evolution, IoT-based systems must be able to appropriately react to both internal and external changes in order to maintain expected behavior under a wide range of situations. To do so, self-adaptive behavior must be considered and integrated in smart devices at design time. Indeed, self-adaptation describes the capability for self-contained systems to adapt to internal or external contextual changes in order to verify a set of adaptation goals (also called control objectives). Additionally, IoT-based systems are dynamic networks, and smart devices can be added or leave the network at any moment, which further strengthen the need for self-adaptability.

In our work, we emphasize the importance of self-adaptive behavior as a preliminary solution to the improvement of IoT devices intelligence. Indeed, by being able to appropriately react to a variety of unpredictable situations, smart devices minimize the need for human intervention, consequently making IoT-based systems more autonomous.

Even though we have discussed the importance of the integration of self-adaptive behaviors at the smart device level, it should also be integrated to higher-level layers of the IoT in order to unlock the capability of building full-scale smart services using smart devices as their basis. To do so, we must study self-adaptation at the device integration level at design time.

The study of self-adaptive systems is traditionally performed under a software perspective. In such systems, researchers deal with the self-adaptation of complex and distributed software applications in response to changing execution environment [Zha+13; Zha+14; Vil+13; WMA10]. The principal motivation for self-adaptation is to reduce the need for human intervention when systems are confronted with changes in both internal and external context. Typical solutions of self-adaptive software frameworks rely on variations of feedback loops. Practically, such systems are enabled with self-monitoring through various software probes, and self-adapt based on a variety of thresholds and adaptation goals. Self-adaptation can typically be defined with respect to an expected target behavior (i.e., functional properties), but also related to Quality of Service (QoS) preservation (i.e., the monitoring of Non-Functional Properties (NFP) to ensure the system stays within the desired non-functional range) [Vil+11]. Typical examples of QoS preservation are the minimization of response time or the study of the scaling capabilities of the system just to mention a few. Such self-adaptation framework commonly use imperative specifications of adaptation goals and mechanisms [IW14; Tam+13].

Aforementioned specifications mandate the comprehensive modeling of the entire self-adaptive execution flow, which limits their use in large systems since the complexity of this flow increases along with system size. This increasing complexity can be the source of a variety of problems, such as unmaintainable code bases or difficultly debuggable adaptation mechanisms. The scalability concern can be solved using declarative specifications, where systems designers specify *what* are the system's objectives rather than *how* to achieve them, and it is a particularly good fit for self-adaptation frameworks as designers are only required to specify adaptation goals.

In an IoT-related context, self-adaptation is an intrinsic and important property of smart devices. Indeed, they should be remotely-configurable in order to account for a variety of potentially extreme situations to ensure system's objectives verification, where objectives are typically specified in terms of system-wide Service Level Agreements (SLA).

Additionally, the dynamic nature of the IoT network increases self-adaptation difficulty, as systems objectives might change at run time because of alterations in smart devices' context or non-functional requirements. Such changes influence associated thresholds, monitoring variables or even the entire monitoring logic if contextual variables are added or removed from the system. Consequently, when adaptation goals evolve at run time, both monitoring logic and self-adaptation flow may become invalid because self-adaptive behavior may rely on outdated goals, or because monitoring logic utilizes probes irrelevant to the new adaptation goals.

In addition, we highlight the difference between functional and non-functional self-adaptation. The former deals with self-adaptive behavior of IoT-based systems under a purely functional perspective (e.g., if a presence is detected in a room, the lights should be turned on, if a window is opened, the heater should be turned off, etc.). The latter are used to guarantee consistent QoS over a variety of operating conditions (e.g., if a sensor fails, it should be replaced by a functional one, if network latency increases,

irrelevant sensors should stop streaming data, etc.). Rule-based functional adaptation has already been discussed in the literature [Zha+13; Zha+14; SLR17; Syl+17], but non-functional requirements are often elapsed or very succinctly treated. Nevertheless, QoS preservation is a salient property of critical IoT-based systems such as the ones used for healthcare or traffic-control applications. Therefore, we put a particular emphasis on non-functional self-adaptation as a mean to increase systems' robustness and reliability.

Additionally, IoT-based self-adaptation should consider smart devices limited resources. To do so, our proposed framework is made resources-aware by the modeling of limited and consumable smart devices' resources. Such modeling approach can then be used as the basis for self-adaptive behavior, in which conditions of devices' internal context (i.e., internal resources) can be monitored and compared to predefined thresholds in order for smart devices to self-adapt and preserve appropriate QoS.

In summary, we develop a novel self-adaptation framework for IoT-based systems, particularly emphasizing on the following characteristic:

- *Declarative adaptation goals specification*: By using declarative specifications for representing goals, we not only tackle the scalability issue brought by imperative approaches, but also reduce the difficulty of self-adaptive systems implementation in distributed environments. Indeed, it is easier for end-users to only specify *what* a system should be doing rather than explicitly describe its entire execution flow using procedural instructions. Declarative programming -like approaches however need to rely on detailed systems model, which is discussed bellow.
- *Formal self-adaptation models*: Declarative approaches typically rely on models to derive actual execution flows from high-level objectives specification. For our specific use-case, which uses wearable devices, the utilization of models also increases the degree of certitude that systems will behave as expected. Indeed, formal objectives specification provides additional guarantees over global system correctness, which ensures that adaptation goals will be achieved for any situations specified in the models of our systems. We consequently use formally-defined models of computation as representations of smart device behaviors, (i.e., run time behaviors) which are then used to build self-adaptation strategies.
- *Automated controller generation*: We also integrate advanced automation and controller synthesis to our self-adaptation framework. Indeed, as IoT-based systems grow bigger, the development time of self-adaptive behavior increases. In order to minimize this time, we emphasize on the importance of automating the self-adaption process by automatically generating controllers and remotely deploy them in response to any changes in the objective goals. By such, we are able to handle potentially large systems accounting for hundreds or even thousands of devices.
- *Ability to handle dynamic self-adaptation scenarios*: As before-mentioned, the dynamic nature of IoT-based systems requires the integration of the capability to handle changing adaptation objectives and monitoring logic. To this end, we apply separation of concerns between adaptation goals, the actual adaptation loop and the monitoring infrastructure as an intrinsic solution in our self-adaptation framework.

The remaining of this chapter is organized as follows: Section 5.2 details an illustrative case-study from the healthcare domain based on wearable devices, that will be used throughout the upcoming chapters to demonstrate our self-adaptation framework.

Section 5.3 subsequently focuses on the integration of synchronous self-adaptive behavior with dynamic adaptation goals and monitoring infrastructure capabilities, while Section 5.4 extensively presents rule-based adaptation goals specification. Eventually, the extension of our self-adaptation framework with hybrid capabilities is considered in Section 5.5. We conclude our chapter in Section 5.6 and summarize our contributions while shedding light on their limitations.

5.2 Self-Adaptation Case-Study

In this section, we extensively present a healthcare-based self-adaptation case study, which will be used as a background for the non-functional adaptation of critical IoT-based systems. We consequently consider the remote monitoring of patients at risk for myocardial infarction recurrence, for which patients require continuous monitoring of various physiological parameters as means to detect cardiac malfunctions and subsequently trigger rapid medical response. Continuous supervision of physiological parameters is accomplished using battery-powered wearable sensors. Additionally, we also consider the remote monitoring of the patients' living environment using either continuously-powered or battery-powered environmental sensors. In order to realistically represent typical IoT-based systems, we assume that both wearable and environmental smart devices are resources-constrained, with limited computing, storage and RAM capabilities. Such limitations also have implications on communication protocols, and we consider that devices are using energy-saving and simple protocols in order to minimize computing overhead and energy consumption.

This case-study particularly focuses on the robust detection of cardiac malfunctions and the consecutive generation of alarm signals to trigger urgent medical response. Robustness requirements are two-fold: false positives cardiac malfunction detection (i.e., detecting a cardiac malfunction while none occurred) should be avoided, and more importantly the detection of heart failure should occur even if the system does not operate at full capacity.

Our case-study focuses on self-adaptive properties. Indeed, the global adaptation goal is to guarantee the patients' safety property by maintaining appropriate QoS of both physiological and environmental data continuous monitoring. To satisfy this safety objective, we define it in terms of several factors: the resource-awareness factor, the resilience factor (i.e., the substitution of defected objects with normally-operating alternatives) and eventually the healthcare-awareness factor (i.e., the request for medical assistance in case of myocardial infarction or technical intervention if a sensor reports abnormal values). As a consequence, our case study details safety-enabled smart homes which support self-adaptation objectives based on resources consumption, resilience and external medical and technical assistance. In such system, self-adaptation is implemented through the remote configuration of smart devices based on internal resources monitoring and the substitution of failed devices with other devices in working order, but also the triggering of medical response if abnormal physiological parameters are measured. Namely, the monitored variables in this specific case-study are: battery levels, sensors states and sensors values. From these data, various thresholds can be established and can be used to trigger self-adaptive actions.

Practically, we assume that patients and smart homes each account for the following wearable and environmental sensors:

- *Battery-operated cardiorespiratory sensors*. As mentioned in previous chapters, this smart device monitors Heart Rate (HR), Heart Rate Variability (HRV) parameters and Respiration Waveform (RWF). They are also able to self-monitor their battery level, and to determine if they are unattached. Such sensors embed a self-adaptive state machine, which includes an energy-saving failsoft state. In this state, respiration measurements and HRV parameters computation are stopped, and the average HR is transmitted every five minutes.
- *Wearable Photoplethysmography (PPG) sensors*, which continuously stream PPG parameters, such as average HR and long-term HRV. This sensor consequently provides redundancy, as it is able to measure similar parameter than the cardiorespiratory sensor. However, measured parameters are less precise, and they should only be used as a backup in case the first sensor is failing. In these sensors, we assume three self-adaptation states: the stopped state, the low sampling frequency state and the high sampling frequency state.
- *Gross position sensors*, which are continuously powered and stream a gross position estimate of the patients' (i.e., in what room or room area are the patients located).
- *Fine position sensors*. They are also continuously powered, but such sensors stream precise patients' location (typically with an accuracy in the order of a centimeter).

Both categories of position sensors feature only two simple self-adaptation states: *on* and *off*, and the transitions between those states can be remotely triggered. This case-study can be considered as a critical IoT-based system because malfunctions in remote monitoring implies risk-patients are not accurately overseen because of the lack of medical response if a critical health crisis occur. Additionally, we assume that patients' health can be estimated using either the cardiorespiratory sensor (optimal case), or using a combination of data from the other sensors (suboptimal case). Preservation of QoS consequently means ensuring that the patients are always appropriately monitored through self-adaptation. In our case-study, self-adaptive behavior is achieved by triggering various state changes of the smart devices' embedded state machines. In the following sections, we give more detail about adaptation objectives specification and controller generation.

5.3 Dynamic and Synchronous QoS-Driven Self-Adaptation for the IoT

As mentioned earlier, IoT-based systems are dynamic: they are confronted with an always-evolving physical world, but they are also faced with smart devices addition or removal at run time. In order to handle such characteristics, IoT-targeted self-adaptive systems should be able to manage changing adaptation goals, but also evolving monitoring infrastructure. In this section, we present our separation of concerns based approach as a solution for full-fledged dynamic QoS-driven self-adaptation for the IoT.

5.3.1 Managing Dynamic Objectives and Monitoring Infrastructure

In the context of IoT-based systems, functional and static self-adaptation is a well-explored research area [Zha+13; Zha+14; Can+14]. Nevertheless, to the best of our knowledge and based on our survey, the study of IoT-based systems' behavior under dynamic adaptation goals dealing with NFPs and resources constraints is lacking.

In the context of self-adaptive software systems, the DYNAMICO reference model details a conspicuous solution to the management of changing adaptation objectives and monitoring infrastructures [Vil+13]. To do so, this model embeds self-adaptive capabilities to adaptation objectives and monitoring through three distinct MAPE-K feedback loops:

- *Adaptation objectives feedback loop*: This MAPE-K loop governs self-adaptation of the software systems with respect to changing adaptation goals. It features all reference elements of traditional MAPE-K loops (i.e., monitor, analyzer, planner and executor), and adaptation objectives are considered under the form of SLAs.
- *Adaptation feedback loop*, which implements actual self-adaptive behavior in reaction to changes in internal or external context.
- *Monitoring feedback loop*: This loop infers relevant context variables that should be monitored in order to self-adapt the system. Because of the presence of feedback, the monitoring logic can also adapt itself if changes in the monitored variables are required.

In fact, DYNAMICO implements separation of concerns between adaptation objectives, self-adaptive behavior and monitoring logic. By doing so, flexibility and modularity is improved as each of the MAPE-K loops can be implemented as independent elements. However, this reference architecture is only defined for software-oriented systems, and typical implementations rely on resources-hungry infrastructures such as the SMARTER-CONTEXT monitoring framework or the FRASCATI middleware [Tam+13], which are not relevant for the self-adaptation of IoT-based systems using distributed, heterogeneous, and resources-constrained smart devices.

As depicted in Figure 5.1, we propose an IoT-targeted dynamic self-adaptation framework based on separation of concerns. Our framework comprises a distributed self-adaptation infrastructure and a controlled IoT-based system. It includes smart devices, which are accessed trough gateways that handle the translation between low-energy communication protocols and higher-level Internet-oriented protocols. Gateways typically access smart devices through the hardware services they expose, and they can perform remote reconfiguration through the self-adaptation state machine in order to guarantee contracted SLAs, usually expressed in terms of non-functional requirements.

Adapted from DYNAMICO, our framework considers four types of interactions between the three MAPE-K loops:

- *Adaptation objectives*: These interactions feed the reference adaptation goals determined by the objectives controller to the adaptation loop and the monitoring loop. For instance, the first resource-aware adaptation to be considered in our case-study is related to the battery levels of the sensors, which implies a reference input under the form BatteryLevel > 20 %. This reference will be used by the adaptation MAPE-K loop as an element to be analyzed for potential device self-

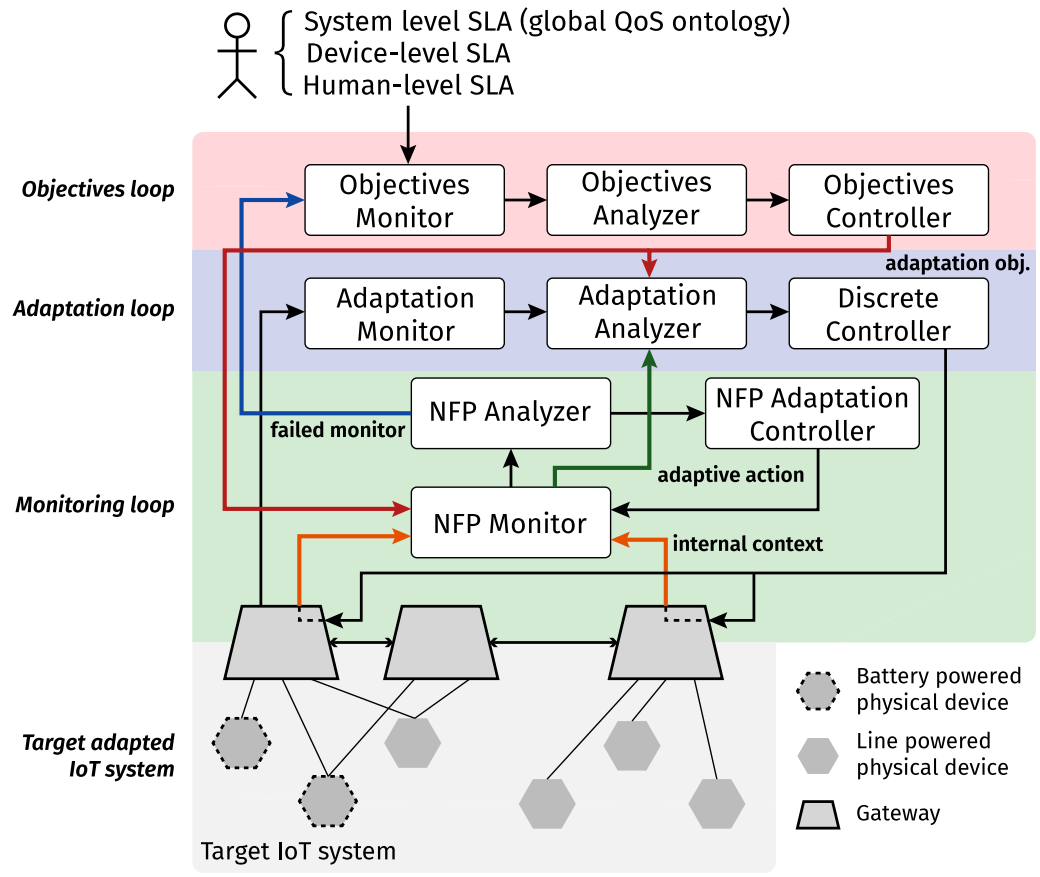


Figure 5.1 – Detailed QoS-driven self-adaptation framework

adaptation, and by the monitoring feedback loop as a reference input for context monitoring.

- *Failed monitor*: Such interactions are used when the need for a change in the adaptation goals is detected by the monitoring MAPE-K loop. In our framework, they are practically used to communicate failed monitors (e.g., in case of total battery drain of one of the sensors) in order to adapt the adaptation goals appropriately.
- *Adaptive action*: These interactions feed adaptation-triggering events from the monitoring feedback loop to the adaptation feedback loop. For example, a consistent and faster-than-normal drain can be used as an event to trigger preemptive self-adaptation of the system and a quicker adoption of a battery-saving failsoft state in order to extend the duration of quasi-optimal system behavior. Such adaptation triggering events can also be produced by the monitoring system using a combination of several NFPs from the controlled smart devices. This use of measured NFPs to derive adaptation triggering events can be implemented in software only using software probes.
- *Internal context*: Such interactions feed the internal context from the adaptation loop to the monitoring loop. They can be used to insure system consistency after self-adaptation. For example, in our case study, if the cardiorespiratory sensor must be replaced by the fine position smart device for health monitoring, the internal context interaction is utilized to ensure that fine position sensors are

all in the functional state after the self-adaptation, thus ensuring global system safety.

The three feedback loops share the same architectural organization, derived from classical MAPE-K autonomic elements. They all feature monitors, analyzers and controllers, the latter resulting from the combination of planners and executors of traditional MAPE-K loops. The purpose of these architectural elements is similar in each of the feedback loops. Monitors provide the system with information on adaptation-triggering events. For instance, the objectives monitor of the adaptation objectives' feedback loop check for changes in the high-level adaptation goals (i.e., the addition or removal of adaptation rules), while the NFPs monitor subscribes to streams indicative of the controlled systems' QoS properties (such sensor state for instance). Such monitors can also implement software probes that make use of the subscribed streams to derive higher-level monitoring variables. Eventually, the adaptation feedback loop monitor's implements sensing probes required by self-adaptive synchronous discrete controllers (i.e., the controller raw inputs).

The objective analyzer decides which adaptation rules must be applied using discrete controllers (i.e., QoS preserving adaptation rules) and which adaptation rules should be transferred to a rule engine, which handles all functional adaptation goals. Please note that the rule engine is not explicitly represented on Figure 5.1 as we focus on self-adaptation with respect to NFPs. The adaptation analyzer takes raw QoS properties and higher-level QoS properties, and feeds them to self-adaptive synchronous discrete controllers. Finally, the NFP analyzer detects adaptation-triggering events in the monitoring infrastructure (e.g., a smart device fails and occurrences of this device should be removed from the adaptation objectives), and delivers such events to both objective monitors and NFPs monitors in order for the system to suitably self-adapt to such changes. When it comes to controllers, the objectives controller transmits rules to the self-adaptation feedback loop. Additionally, the adaptation feedback loop's discrete controller is an automatically-synthesized synchronous discrete controller performing system's self-adaptation with respect to pre-contracted QoS. Moreover, the NFP adaptation controller provokes the subscription or unsubscription to QoS parameters' streams.

In addition, our dynamic self-adaptation framework implements self-adaptive behavior using synchronous discrete controllers. By doing so, we rely on advanced formalism and tools developed by the reactive systems community to provide guarantees over targeted system behavior. Furthermore, the use of a state-based discrete approach in discrete controller synthesis ensures interoperability, since no preliminary hypothesis are made neither on cross-devices communications, nor on devices reconfiguration interfaces. Using discrete controllers is also realistic in an IoT-related context, as most smart devices feature remote configuration capabilities using low-energy wireless protocols.

Practically, this entire synchronous self-adaptation framework can be deployed into gateways, which have sufficient computational resources and can handle the event flows generated by the IoT-systems. This is usually the case considering typical hardware used in IoT gateways (e.g., Raspberry Pis or Intel Edisons). For example, the NFPs monitors and adaptation monitors are realized by subscribing to the NFP hardware services of smart devices. Such monitors can also be implemented by inferring the non-functional state of the system by listening the environment (e.g., the overall load of the system can

be inferred from measuring the time between two events, and appropriate action can be taken if system load becomes too high).

In addition, Figure 5.1 defines several levels of SLAs:

- *System-level SLAs* designate a contract between users and service providers at the system level. Indeed, end-users of IoT-based systems do not need extensive details to specify their requirements. By providing system-level SLAs, we can simplify the specification of global functional and non-functional requirements by subsequently dividing SLAs into high-level requirements that can be easily mapped to device-level SLA for further analysis.
- *Device-level SLAs* define the guarantees provided by devices manufacturers about their functional and non-functional properties. These SLAs present finer granularity, and are more static, than system SLAs. Differentiation between smart devices and simpler devices is necessary when considering device SLAs: indeed, while SLAs of smart devices are subject to changes over time and are reconfigured by users, SLAs of simple devices remains static. In our opinion, this is a consequence of the black-box nature of simple devices, which are designed by manufacturers to have functional and non-functional properties that cannot be reconfigured during their execution lifecycle.
- *Human-level SLAs* characterize cross-user variability in human-centric IoT-based systems. Integrating people in the system feedback loops substantiates the accurate modeling of system properties (i.e., biological properties) to be controlled or monitored. These system properties may differ from one individual to another considering various diseases impact different physiological parameters.

Expressions in each of these SLAs can be mapped to QoS factors, as described by the ontology in Figure 5.2. For example, the resource awareness QoS factor is a device-level SLA because of resources variability between smart devices (i.e., continuously-powered sensors do not have the need for a low battery Service Level Objective (SLO) obligations, while battery-operated sensors do). In our framework, SLOs are typically given as a set of rules. In opposition, the resilience QoS factor is commonly a system level SLA, where resiliency is specified at the system-level (e.g., if a sensor is failing, it is then substituted with other sensors *of the system* in order to compensate for the loss of information). Eventually, the health awareness QoS factor is a human-level SLA because cardiac malfunctions are associated with different diseases producing different effects on heart activity. As a consequence, the monitored QoS should be adapted for each of the patients in the global healthcare IoT-based system.

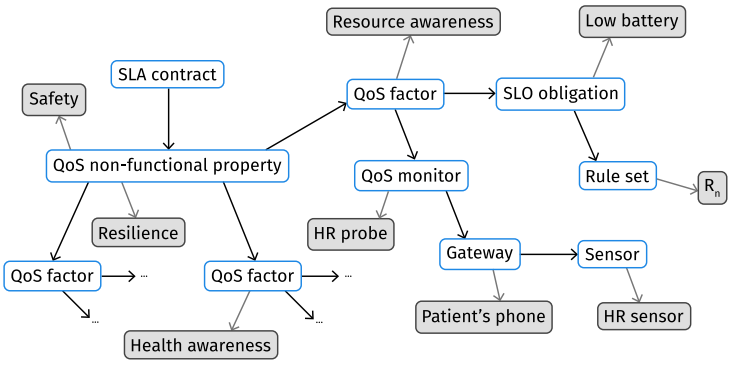


Figure 5.2 – SLA ontology

This section concludes the description of our dynamic and IoT-oriented self-adaptation framework. We consider gateways and their associated smart devices as synchronous reactive systems, and we consequently rely on the numerous tools and methods developed by the reactive systems community to build discrete synchronous controllers for smart devices with limited resources.

5.3.2 Synchronous Programming Languages

In this section, we present how we used Synchronous Programming Languages (SPL) to achieve smart devices' self-adaptation. First, we emphasize on the importance of the synchrony hypothesis, introduced formerly in the state of the art chapter. The hypothesis stipulates that systems can be considered synchronous if and only if the computation time of a reaction to a set of external events is negligible when compared with the rate of such events. In this section, we assume that IoT-base sub-systems controlled by our framework are small-enough for this hypothesis to be valid. Additionally, the reactive system philosophy is a good fit for the representation of IoT-based systems, as most application cases call for the reaction of extremely distributed computer-based systems to external or internal events occurring at any moment in time.

We thus used methods from the reactive systems community in order to provide safe and reliable controllers for self-adaptation in the IoT. Practically, this section focuses on the discrete controller block of Figure 5.1. Common paradigm for programming reactive systems are SPLs. Such languages typically rely on Labeled Transition Systems (LTS) to model system's interaction with the external world.

Formally, a LTS is defined as a quadruple $(S, L, \rightarrow, s_{in})$, where S is a set of states, L is a set of transition labels, $\rightarrow \subseteq S \times L \times S$ and s_{in} the initial state. We define the set of transitions labels as $L = (events, actions, \backslash)$, where $\backslash \subseteq events \times actions$. LTSs are comprehensive models of discrete systems, where non-controllable variables are members of the set *events* and remote services invocations are members of the set *actions*. In our framework, *events* are generated by monitors and *actions* are various remote configuration options presented to the external world as a set of hardware services.

In the context of discrete controller synthesis applied to reactive systems, transition labels are separated in two sets: controllable variables and non-controllable variables. Practically, controllable variables define transitions that can be triggered remotely. They thus can be used by the discrete controller to achieve a predefined adaptation goal. In opposition, non-controllable variables describe transitions which can only be triggered internally, based on self-measurements of the system the LTS is modeling. In our case, we introduce a syntactic separation of non-functional and functional variables in order to improve the expressivity of discrete systems models. To do so, we define the statement "$e \backslash a$" as the control of event e by service a during the firing of the transition. When no adaptation service is provided for a given transition, we assume that this transition is non-controllable. For example, in battery-operated smart-devices, the transition going from an empty battery state to a normal operation state is non-controllable as battery must be charged externally (i.e., the system can not charge the battery by itself, consequently mandating external human intervention).

Figure 5.3 illustrates the LTS of the cardiorespiratory sensor as it is described in the previous chapter. Please note that this LTS differs from the one given in Figure 4.6 because some simplifications were needed in order to integrate it in typical SPLs. First,

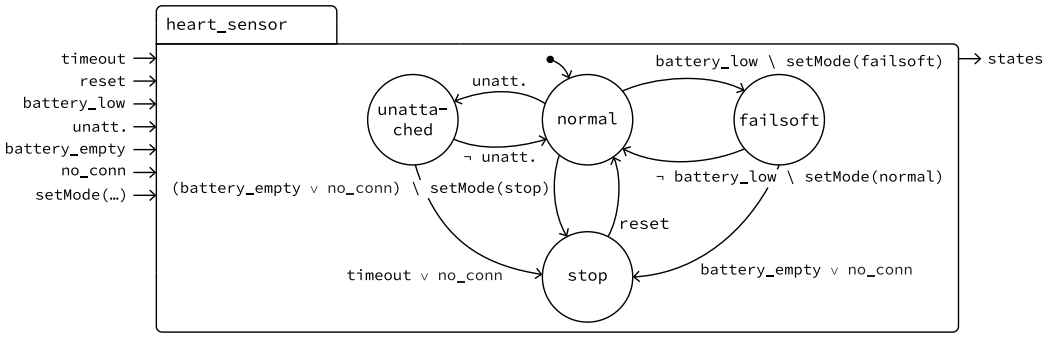


Figure 5.3 – Cardiorespiratory sensor LTS

most synchronous languages only deal with Boolean-typed transition variables. Some attempts at the integration of other variable types were described, but they make state-space exploration complex and computationally intensive. We thus convert all transitions to Boolean-typed labels. Additionally, the initialization state is not represented, as it is only a transitive state which always goes into the normal state.

Practically, our model takes input (see on the left-hand side of Figure 5.3) and produces outputs (on the right-hand side of Figure 5.3). Inputs of the model are used to trigger transitions of this LTS, while outputs are produced by the states of our LTS. The initial state of the heart activity sensor is the `normal` state. In this state, our smart object streams real-time HR data and HRV parameters values. If a low battery level is detected, the sensor is placed into an energy-saving `failsoft` state. In this state, only the average HR is transmitted every 5 minutes. It is worth observing that transitions between the `failsoft` state and the `normal` state are controllable. This was motivated by our need to control transitions between these states if a precise and instantaneous estimation of cardiac health is mandated by the adaptation context (e.g., during a critical health crisis). The two remaining states are the `stop` state and the `unattached` state. The stop state in accessible from any of the other states. It is accessed from the normal state if the battery is empty, if it is requested externally, or if a no connection timeout occurs, rendering the transition from the normal to stop state controllable. This timeout is triggered if our smart device is not connected to any external tier for more than one minute. The `stop` state is reached from the `failsoft` state if the battery is empty, or if a no connection timeout occurs. The `unattached` state is reached from the `normal` state if our smart device self-detects its unattachement from the patient's skin. Once the device is in this state, the sensor is either placed back in the `normal` state if our sensor self-detects its reattachment, or it is placed in the `stop` state if the `unattached` state is active during a predefined amount of time (i.e., if the sensor is unattached for too long, it is turned off for energy-saving purposes). Once our smart device is stopped, the only way to reach the `normal` state again is to reset it manually, and an uncontrollable transition is defined between the `stop` and the `normal` state.

Figure 5.4 describes LTSs for both gross and fine sensor positions, which comprise two controllable states: `on` and `off`. These LTSs only take the controllable label `c` as an input and its output corresponds to the states of the smart devices.

For our framework, we chose to use the Heptagon/BZR SPL to build the controller. Our choice is driven by the fact that Heptagon/BZR is open-source, actively maintained, and surrounded by many discrete controller synthesis research works (such as its ex-

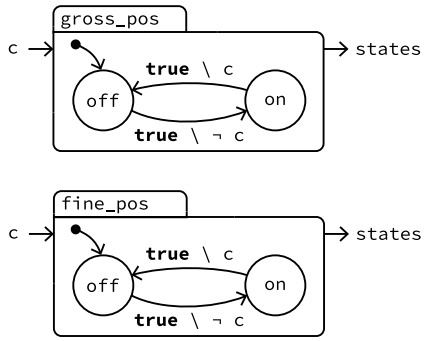


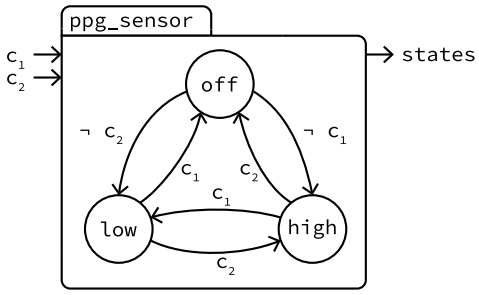
Figure 5.4 – Fine and gross position sensors LTSs

tension to handle infinite state spaces and rich data types). Figure 5.5 illustrates the LTS of the PPG sensor along with its Heptagon/BZR specification. In Heptagon/BZR, all the concurrent entities of the system are modeled as nodes taking inputs and returning outputs. Figure 5.5.(b.) illustrates the specification of the node modeling the PPG sensor.

Because the battery-life of the PPG sensor is considered to be much longer than the battery-life of the cardiorespiratory smart device (because the hardware is much less energy consuming, and because much bigger batteries can be embedded), we assume that all the transitions are controllable, and they are noted as c_1 and c_2. All transitions should be noted as **true** \ c_1 or **true** \ c_2, but the **true** were omitted for conciseness purposes. Some transitions are associated with the negation of the conditions c_1 and c_2 because during the controller synthesis process, values of **true** are preferred for controllable variables. This can consequently be used for finer control of transition between states and global non-functional properties. For example, in our case, negations of conditions are associated with transitions leaving the `off` state of the PPG sensor. Considering the controller synthesizer prefers **true** values of c_1 and c_2, we are able to save energy by only leaving the `off` state when this smart device must be used for self-adaption purposes. In Heptagon/BZR, the PPG sensor is defined as a node taking both controllable variables as inputs and producing three outputs, corresponding to the sensor states. This LTS is encoded using an automaton, which fully specifies LTSs using a very simple syntax. First, states of the automaton are defined using the keyword `state`. Then, actions to be performed during the state are specified after the keyword `do`. Eventually, transitions are written using the `until` and `then` keywords. If the condition provided after `until` is true, the automaton goes to the state provided after `then`.

When it comes to the LTS of the PPG sensor, we define three states: `off`, `low` and `high`. In the `off` state, the smart device is turned off and does not stream any PPG parameters. The `low` state can be seen as an energy- and bandwidth-saving state where parameters are streamed at a low frequency. In opposition, in the `high` state, sensor data are streamed at a high frequency in order to have better remote estimation of PPG data.

Once all smart devices in the IoT-based sub-system are described as Heptagon/BZR nodes, discrete controller synthesis can occur. This process relies on control contracts, which are specified using basic logic statements. Such control contracts introduce four keywords: `contract` defines the beginning of a control contract, `enforce` describes the adaptation rules, `assume` is used to define preconditions on the adaptive actions,



a.

```
1  node ppg_sensor(c1, c2: bool) returns (s_high_sr, s_low_sr, s_off: bool)
2  let
3    automaton
4      state Off
5        do s_off = true; s_low_sr = false; s_high_sr = false;
6        until not c1 then HighSR
7          | not c2 then LowSR
8      state HighSR
9        do s_off = false; s_low_sr = false; s_high_sr = true;
10       until c2 then Off
11         | c1 then LowSR
12     state LowSR
13       do s_off = false; s_low_sr = true; s_high_sr = false;
14       until c1 then Off
15         | not c2 then HighSR
16   end
17 tel
```

b.

Figure 5.5 – PPG sensor LTS (a.) and its Heptagon/BZR specification (b.)

and `with` designates controllable variables. An example of a BZR contract is given in Figure 5.6. This control contract describes 5 rules, each of which corresponding to different QoS level. Rule `r1` is aimed at preserving a reasonable workload in the local smart home network, and it stipulates that if the cardiorespiratory sensor is operating normally, the sub-system's controller must ensure that both the PPG and fine position sensor are turned off. This rule improves energy savings for the PPG sensor, and keeps the workload in the local network as low as possible by disabling fine position sensors. Indeed, such sensors stream positions at a high rate, which might cause network throttling and thus introduce latency if they are overused. Rule `r2` is related to the safety QoS because it states that if the cardiorespiratory smart devices is in the `failsoft` state, gross position sensors must be turned on, and the PPG sensor must be placed in the low sampling rate state. This rule provides enough data to the cardiac health estimation remote service in order to compute rough estimate of the patients' health. The state is however suboptimal as in the `failsoft` state, the cardiorespiratory sensor only streams average HR every 5 minute. Rule `r3` and `r4` are also used to guarantee the cardiac health estimation service is fed enough data to be able to compute a health estimation. Practically, rule `r3` states that if the cardiorespiratory smart device is unattached, the gross position sensors should be turned on and the PPG sensor is again placed in the high sampling rate state. Rule `r4` states that if the cardiorespiratory sensor is stopped, the fine position sensors are turned on and the PPG sensor is placed in the high sampling rate state Rule `r5` specifies that if gross sensor position are *on*, the fine position sensor should be turned *off*, consequently minimizing redundant information.

After the specification of all system nodes along with a control contract, Discrete Controller Synthesis (DCS) can occur. With Heptagon/BZR, two controller synthesizers

```
node smart_home(h_battery_low,
                h_battery_empty,
                h_timeout,
                h_no_conn,
                h_res,
                h_unatt: bool) returns (h_s_normal, h_s_failsoft, h_s_unatt, h_s_stop, gp_s_on,
                                        gp_s_off, fp_s_on, fp_s_off, ppg_s_high_sr,
                                        ppg_s_low_sr, ppg_s_off: bool)
contract
  var r1, r2, r3, r4, r5: bool;
  let
    r1 = not h_s_normal or (fp_s_off and ppg_s_off);
    r2 = not h_s_failsoft or (gp_s_on and ppg_s_low_sr);
    r3 = not h_s_unatt or (gp_s_on and ppg_s_high_sr);
    r4 = not h_s_stop or (fp_s_on and ppg_s_high_sr);
    r5 = not gp_s_on or fp_s_off;
  tel
  enforce r1 & r2 & r3 & r4 & r5
  with (h_c_set_failsoft_n, h_c_set_normal, h_c_set_shutdown_n, fp_c, gp_c, ppg_c1, ppg_c2: bool)
let
  (h_s_normal, h_s_failsoft, h_s_unatt, h_s_stop) = inlined heart_sensor(h_battery_low,
                                                                         h_battery_empty,
                                                                         h_timeout,
                                                                         h_no_conn,
                                                                         h_c_set_failsoft_n,
                                                                         h_c_set_normal,
                                                                         h_c_set_shutdown_n,
                                                                         h_res,
                                                                         h_unatt);
  (gp_s_on, gp_s_off) = inlined gross_position_sensor(gp_c);
  (fp_s_on, fp_s_off) = inlined fine_position_sensor(fp_c);
  (ppg_s_high_sr, ppg_s_low_sr, ppg_s_off) = inlined ppg_sensor(ppg_c1, ppg_c2);
tel
```

Figure 5.6 – Heptagon/BZR control contract example

can be used: Sigali [1] and ReaX [2]. The discrete controller synthesis process is given in Figure 5.7, where S_b and S_c respectively describe states of the system and the contract, outputs e_a and e_g respectively designate an environmental model of the contract and a contract requirements that should be enforced. Eventually, **x**, **y** and **c** are respectively the system inputs, outputs and controllable variables. In Heptagon/BZR, both the contract and system's nodes are converted to symbolic transition systems, that are subsequently used to perform controller synthesis through the state space exploration.

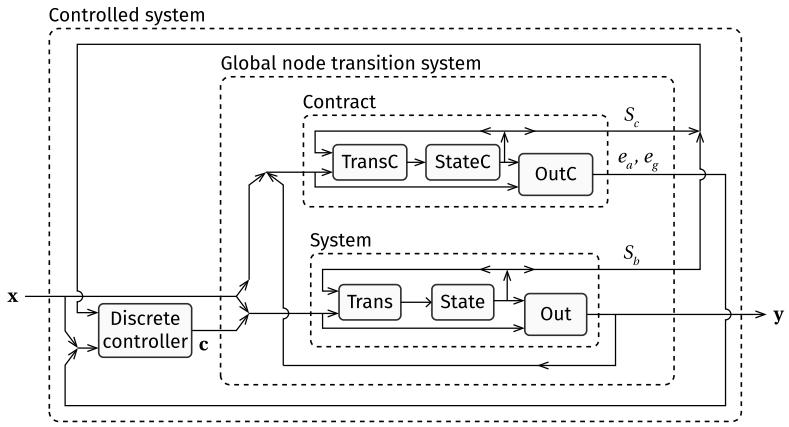


Figure 5.7 – Discrete controller synthesis workflow, adapted from [DMR10]

1. Sigali toolbox: `http://www.irisa.fr/Polychrony/Sigali.php` (visited on 07/18/2018)
2. ReaX controller synthesizer: `http://nberth.space/en/opam#reatk` (visited on 07/18/2018)

Practically, DCS results in a correct-by-construction C or Java program that can be integrated to our final system and that guarantees adaptation rules are verified at all time. Please note that, in the DCS context, correct-by-construction only means that the generated code ensures the LTSs are correctly translated into imperative code and that their coordination respects the control contract. Additionally, for the code execution to be correct, the synchrony hypothesis should be verified during runtime. Hetagon/BZR generates a `reset` and a `step` function, the former being called upon system statup and the latter being called after system's inputs acquisition in order to compute outputs of the system. Output values can subsequently be used to update smart devices to the expected states. Since the controller is synchronous, its inputs can be periodically acquired through a simple loop.

In summary, DCS is used to build controllers using LTS-based models of the system and a control contract. However, adaptation goals specification still require knowledge of logical connectors and low-level knowledge on system models, such as the available states and possible transitions.

In our opinion, this represents a limitation as IoT-based systems end-users are often non-experts, thus calling for declarative and simple adaptation objectives specification. To tackle this disadvantage, we introduce a new rule-based language to specify self-adaptive behavior.

5.4 Adaptation Objectives Specification

In this section, we discuss our contributions with respect to the specification of adaptation objectives. This research problem must take into account a variety of challenges. First, we emphasize the importance of adaptation objectives maintainability, which is motivated by the fact IoT-based systems can grow to account for thousands of devices. Adaptation objectives that specify desired devices' behavior must consequently be able to accurately handle large number of smart devices. They should also be easily added and updated. In addition, adaptation objectives specification should be expressive enough so that non-expert users can specify their own self-adaptation goals. Indeed, IoT-based systems are often used in applications requiring the collaboration of user with various profiles and backgrounds. For instance, healthcare-related IoT-based applications involve interactions with a wide variety of staff, such as medical doctors, nurses, emergency medical technicians, etc. In order to democratize IoT-based systems adoption, their use and adaptation should be easy-enough for non computer science experts. Eventually, such systems can grow up to large scale and require tools and techniques, encouraging the use of automation, as it can consequently be used to save development time.

5.4.1 Declarative Self-Adaptation Specification

The traditional approach when designing adaptable systems is that system designers specify the flow of instructions necessary to achieve a particular adaptation goal. By using this approach, system designers should have a global understanding of their system in order to specify step-by-step all the instructions needed to implement the desired systems' specification.

While this approach is appropriate for small- to medium- sized systems, it is the source of several issues as IoT-based systems grow larger. Indeed, while development paradigms such as agile methods or object-oriented programming improved code organization and code lifecycle management, changes in the objectives of the software or changes in the requirements can still imply important and time-consuming code changes, where a deep and total understanding of the existing code is necessary to implement the changes correctly. This causes difficulty for the maintenance of complex and large systems.

As our work considers the IoT context, scalability and reusability are criterions of paramount importance when attempting to develop solutions for IoT. Indeed, the IoT is:

- *Dynamic*: In IoT-based systems, changes occur constantly. Since IoT-based networks are based on heterogeneous smart devices using a variety of technologies and protocols deeply embedded in the physical world, IoT-based systems constantly change in reaction to variations of the physical world. Management tools and strategies must take into account this high dynamicity, and tools developed for the IoT must be able to manage such changes, that occur on different scale. Indeed, changes can occur on the lower level of the IoT stack (i.e., failing devices, network throttling, etc.), or in the higher level of the stack (changes in the goal of the IoT systems, changing in the NFPs requirements).
- *Large*: Solutions developed in an IoT context must be scalable. Indeed, IoT systems can be made of hundreds, if not thousands, of devices constantly interacting with the physical world, with each other, and with software frameworks or Cloud-based solutions. In order to realistically answer to IoT challenges, frameworks must thus be scalable, and be able to handle the management of potentially huge number of devices.
- *Constrained*: Large systems are nothing new, and the management of highly distributed clusters or Cloud infrastructure are well discussed topics in the scientific literature. However, a new challenge brought by the IoT is related to constraints on resources caused by the limited computational and storage capabilities of devices used in the IoT. These constraints must be taken into account when working in the IoT context.

Imperative approaches show drawbacks when handling dynamicity and scalability requirements. Indeed, the poor maintainability of imperative programming makes it relatively static and reaction to changes is difficult, especially when large systems, with a lot of interacting components, are considered. Imperative programming is however able to deal accurately with constrained systems, as precise instruction flow can be specified by experts and optimized in order to appropriately balance resources management and systems functionalities.

In opposition, declarative approaches were developed. In such methods, system designers do not specify detailed instruction flows, but they rather specify what a system should do. Declarative approaches can be found in technologies such as relational databases and the Structured Query Language (SQL), functional programming languages or business and production rules. These technologies rely on the use of a detailed and formal model of their application domain and a declarative specification syntax of the system objectives. Models and specification are then used in combination to automatically implement the system goals. Advantages of these methods are two folds:

- *Reliance of formal models of systems*: As mentioned above, declarative languages often rely on comprehensive formal models of the systems being programmed. These models are used during program executions, and can be used to provide guarantees on system behavior.
- *Improved maintainability and scalability*: Because declarative languages specify only systems' objectives, programs are typically shorter than their imperative counterpart. This size decrease improves maintainability and scalability because it leads to smaller code bases.

In conclusion, we emphasize the importance of a declarative approach to the adaptation of IoT-based systems. Such approach can simultaneously tackle both the scalability and maintainability issues by focusing on the specification of systems' objectives rather than actual execution lifecycle specification.

5.4.2 Specifying Adaptation Objectives with a Rule-Based Language

In order to make the specification of adaptation goals easier for non-experts users, our solution follows the classical Event-Condition-Action (ECA) pattern. We propose a rule-based language as illustrated in Figure 5.8. Practically, our language handles devices such as *sensors*, *actuators* and *gateways* (keywords **`SENSOR`**, **`ACTUATOR`** and **`GATEWAY`**), which are the three principal smart devices categories. These devices are associated with events (keyword **`EVENT`**), that can be filtered using time windows (keyword **`WINDOW`**). Such devices are typed (keyword **`TYPE`**), and this can be used to perform self-adaptation on groups of devices rather than specific smart devices instances. Similar to objects and classes in the object-oriented paradigm, the device type is a common data structure of similar devices. Each device is described by a set of attribute-value pairs. Attributes may hold information about devices such as characteristics, configuration parameters, and their sensing data from the physical environment. As outlined in the methodology chapter, smart devices implement various services (keyword **`SERVICE`**), which are used to access measurements or to perform remote configuration. Adaptation (keyword **`RULE`**) is specified in terms of predicates, quality name and adaptive actions (keywords **`ON`** and **`QUALITY`**, **`IF`** and **`DO`**). Alternate adaptive actions can optionally be defined in case of unavailability of the smart device involved in the original self-adaptive action through the keyword **`ALTERNATE`**. Such actions are using sensor services to access smart devices non-functional state and remotely adapt if deemed necessary by the discrete controller. For convenience purposes, we also integrate the capability to specify timers, which are typically used if rule needs to be triggered periodically (e.g., a sensor might need to be checked regularly in order to verify if it is still active).

Furthermore, we introduce the **`GLOBAL`** keyword to declare variables related to contextual data external to smart devices attributes. This keyword can be used to bind the external environment to smart devices, and they can refer to external services or cached data in memory for runtime configuration of the rule engine. In our specific case-study, global variables save smart devices' states in order to restore correct system configuration if controller synthesis occurs because of changes in the adaptation objectives or in the monitoring infrastructure.

The adaptation rules specified in Figure 5.6 are given in Figure 5.9 in our specification language. Please note that this is only a simplified example and that smart device type assignment is not explicitly displayed. A rule starts with the keyword **`RULE`** followed by

```
<ECA-system> ::= <variables> <objectives> <adaptation> <devices> <timer>
<variables> ::= <variable> | <variable> <variables>
<variable> ::= GLOBAL <variable_name> | GLOBAL <object_name>
<timer> ::= TIMER <value> <unit>

<objectives> ::= <objective> | <objective> <objectives>
<objective> ::= QUALITY <quality_name> ON <SLA_expression>

<rules> ::= <rule> | <rule> <rules>
<rule> ::= RULE <rule_name> ON <quality_name> IF <predicates>
                            DO <adaptations>
<adaptations> ::= <action_name> | <action_name> <adaptations>
                                | <action_name> [ALTERNATE <action_name>]

<predicates> ::= <predicate> | <predicate> [AND] <predicates>]
<predicate> ::= <device_Type>(<filter>)
                  | <sensor_name>(<filter>)
                  | <actuator_name>(<filter>)
                  | <event_name>(<filter>)

<filter> ::= <sensor_name>.<attribute_name> <operator> <value>
                | <actuator_name>.<attribute_name> <operator> <value>
                | <deviceType>.<attribute_name> <operator> <value>

<action_name> ::= <sensor_name>.<service_name>(<parameters>)
                    | <actuator_name>.<service_name>(<parameters>)
                    | <gateway_name>.<service_name>(<parameters>)

<devices> ::= <device> | <device> <devices>
<device> ::= <sensor> | <actuator> | <gateway>
<sensor> ::=  SENSOR <sensor_name> TYPE <object_name> <events>
<actuator> ::= ACTUATOR <actuator_name>] TYPE <object_name> <events>
<gateway> ::= GATEWAY <gateway _name>] TYPE <object_name> <events>

<events> ::= <event> | <event>, <events>
<event> ::= EVENT <event_name> <attributes> [WINDOW <value>]
<object> ::= OBJECT <object_name> ATTRIBUTES <attributes>
                                  [SERVICES <services> END]
<attributes> ::= <attribute> | <attribute>  <attributes>]
<attribute> ::= VAR <attribute_name> [= <attribute_value>]

<services> ::= <service> | <service> <services>
<service> ::= SERVICE <service_name>(<parameters>) [RETURN(<parameters>) END]

<operator> ::= > | < | >= |<= | == | != | in
<parameters> ::= <parameter> | <parameter>, <parameters>
<parameter> ::= <object_name>.<attribute_name>
<SLA_expression> ::= <object_name>.<attribute_name> <operator> <value>
```

Figure 5.8 – Self-adaptation language Backus-Naur form grammar

the rule's name. The line in the left-hand side of the rule is the conjunction of logical predicates, each of which is written on a separate line. The predicate can be applied on individual device instances or on all instances of a given device type. Predicates work as functions with conditions (called also *filter conditions*) as their input parameter. The filter condition is a logical expression on device attributes, and the logical operator *and* between predicates is explicitly omitted. For instance, in Figure 5.9, a filter is used on line 3, and it is used to select all cardiorespiratory sensor (called **`HRSensor`** for compactness purposes) in the normal state.

```
RULE "R1"
  ON HeartNormal
  IF HRSensorType(State == "normal")
     $ppg: PPGSensorType()
     $fpos: FPosSensorType()
  DO $ppg.setState("off")
     $fpos.setState("off")
END

RULE "R2"
  ON HeartFailsoft
  IF HRSensorType(State == "failsoft")
     $ppg: PPGSensorType()
     $gpos: GPosSensorType()
  DO $ppg.setState("low")
     $gpos.setState("on")
END

RULE "R3"
  ON HeartUnattached
  IF HRSensorType(State == "unattached")
     $ppg: PPGSensorType()
     $gpos: GPosSensorType()
  DO $ppg.setState("high")
     $gpos.setState("on")
END

RULE "R4"
  ON HeartStopped
  IF HRSensorType(State == "stop")
     $ppg: PPGSensorType()
     $fpos: FPosSensorType()
  DO $ppg.setState("high")
     $fpos.setState("on")
END

RULE "R5"
  ON GrossPositionActive
  IF GPosSensorType(State == "on")
     $fpos: FPosSensorType()
  DO $fpos.setState("off")
END
```

Figure 5.9 – Simplified sample adaptation goals using our rule-based language

The `$` prefix is a binding operator, and it attaches a variable to a specific device type or instance (i.e., `$eda: EDASensorType()` binds the `eda` variable to all sensors of `PPGSensorType`). In rule `R1` of our specific case-study, we use binding variables as means of selecting PPG and fine position sensors instances related to the context of the cardiorespiratory sensor in the `normal` state. There is an implicit conjunction between `HRsensorType()`, `PPGSensorType()` and `GPosSensorType()` predicates. Practically, all PPG and fine position sensors in the same sub-system as cardiorespiratory sensors in the `normal` state are respectively placed in the `off` states. In addition, predicates on device types are particularly useful to specify adaptation strategies at the system level while the adaptation must only be triggered in relevant situations. Predicates on device instances allow a fine-grained adaptation at the device level.

Rules introduced in Figure 5.9, even though they describe actions on the lower-levels of IoT-based systems (i.e., sensors and actuators), improves the declarativity of adaptation goals specification because the execution flow is not explicitly specified. Indeed, the execution flow is delegated to an external rule engine, which executes rules when deemed appropriate considering current system input. Such rule engine can then,

though appropriate bindings, propagate its output (i.e., states change in our case) to the devices. Maintainability is improved as end-users only have to specify what device configurations ensure targeted QoS for a particular application. Furthermore, the set of rules introduced in Figure 5.9 should be seen as the initial adaptation goal (i.e., the set of adaptation objectives that is deployed when the system runs for the first time). Since IoT-based systems are dynamics, such rule-based adaptation objectives are bound to evolve at runtime, and this is easily tackled by our dynamic self-adaptation framework.

A common example of an event mandating self-adaptation with respect to adaptation objectives is the PPG sensor failure. Indeed, the PPG smart device deficiency mandates changes in the adaptation goals in order to reflect device unavailability. In our system displayed in Figure 5.1, this is realized though interactions "*failed monitors*" and "*adaptation objectives*." Practically, when the NFP analyzer detects a PPG sensor failure, it forwards this information to the objectives feedback loop. Objectives are subsequently renegotiated in order to remove the PPG sensor from the rule-based adaptation goals so that self-adaptation can still safely occur. The objectives analyzer then translates the high-level specification into a BZR contract, as mentioned above, and controller resynthesis ensues. Eventually, newly generated controllers are deployed on the sub-systems' gateways.

In conclusion, we introduce a full-fledged rule-based self-adaptation language. Our language is used to specify self-adaptive behavior of IoT-based systems. Its declarative nature improves scalability and maintainability. However, our framework is strictly confined at systems verifying the synchrony hypothesis, which is a limitation in an IoT-based context. Indeed, as such systems grow bigger, more computational power is necessary to handle streams of events from devices included in IoT systems, and the synchrony hypothesis thus might not hold anymore. In order to tackle the scalability issue, we transform our synchronous and dynamic self-adaptation framework into hybrid systems with both synchronous and asynchronous elements, and more details related to this matter are discussed in the following section.

5.5 From Synchronous to Hybrid Self-Adaptation

As mentioned at the beginning of this chapter, the synchrony hypothesis claims that for a system to be considered synchronous, one must have enough computational power to compute system reactions to external events instantaneously. While this statement might be true for small-scale IoT-based systems (i.e., including tens of devices) because gateways are typically order of magnitudes more computationally powerful than smart devices, it does not hold for large-scale IoT-based systems (i.e., made of thousands or hundreds of thousands of devices). Indeed, if the events' transmission rate becomes too high, controllers might not be able to compute appropriate self-adaptive actions fast-enough. To tackle this issue, we extend our framework with fully-synchronous system capabilities to include asynchronous coordination of synchronous sub-systems. In this dissertation, sub-systems are defined in terms of gateways managing subsets of smart devices.

Such approach is not new and is traditionally considered under the scope of Globally Asynchronous Locally Synchronous (GALS) systems, which exhibit a global asynchronous behavior with local sub-systems adopting synchronous behaviors [MVF00].

Since IoT-based systems are still principally built around networks of gateways controlling sub-networks of smart devices, this model of computation is a good fit for such systems. Indeed, because the number of events in a gateway-controlled sub-system is limited because of the smaller number of devices in the network, the synchrony hypothesis is locally verified. Nevertheless, when a global view of the system is adopted, where numerous gateways are interconnected and communicate, the high number of events generated calls for an asynchronous approach. Additionally, GALS models are usually used for the description of very low-level systems [MVF00], and are typically used for the hardware synthesis of Application-Specific Integrated Circuits (ASIC) or System-on-Chips (SoC). Higher-level specification languages for GALS systems such as SystemJ [Mal+10] make possible the description of IoT-based systems [Eli+15]. However, SystemJ does not offer automated controller synthesis, which is a key aspect of the controller design at a large scale in the IoT context. Indeed, the dynamic nature of IoT-based systems, where sensors and actuators can be added or removed to the IoT at any moments, compels the presence of automation tools for the controller generation. The evolving nature of such systems also instructs great maintainability, which is penalized by using centralized languages such as SystemJ.

Our hybrid and dynamic self-adaptation framework in Figure 5.10 comprises adaptation objectives MAPE-K loops which are now considered to be asynchronous instead of synchronous. Additionally, adaptation and monitoring MAPE-K loops are now distributed, and independently deployed on a cluster of gateways. The adaptation objectives adaptation feedback loop is implemented in centralized servers to adapt distributed synchronous controllers.

The hybrid nature of our self-adaptation framework comes into play when considering the global system: the asynchronous adaptation objectives manager, because of the mass of system-generated events, queues events upon arrival and processes them using a first-in-first-out strategy. Once the buffered events are processed asynchronously by the manager, synchronous action can be sent to the controlled sub-systems. Sub-systems monitoring occurs synchronously, because of the small size of the considered sub-systems.

Interactions defined in Figure 5.1 still hold in Figure 5.10. Because of the synchronous nature of the adaptation feedback loop, we can still rely on SPLs as means to specify sub-systems and automatically generate discrete controllers. Please note that in this specific case, we do not require intercommunicating gateways. This is motivated by the fact that our considered IoT-based sub-systems are self-contained, and that self-adaptation is typically bounded to a single synchronous system.

Our self adaptation architecture can be distributed to a massive amount of gateways and smart devices, thus improving overall scalability. Nevertheless, the objective feedback loop is still centralized, as system managers need a unique access point to specify self-adaptation adaptation objectives. This does not represent a limitation to the system scalability property, as technologies are available to manage clusters of systems. Additionally, even though the MAPE-K loops must analyze numerous events, tools and methods from the Complex Event Processing (CEP) community can be used to identify and process relevant events. Indeed, CEP tools are able to handle large amount of events, and they commonly implement a publish-subscribe-based approach in order to acquire events streams [CM12]. Additionally, CEP systems generally associate application-specific semantic for the sake of application logic specification.

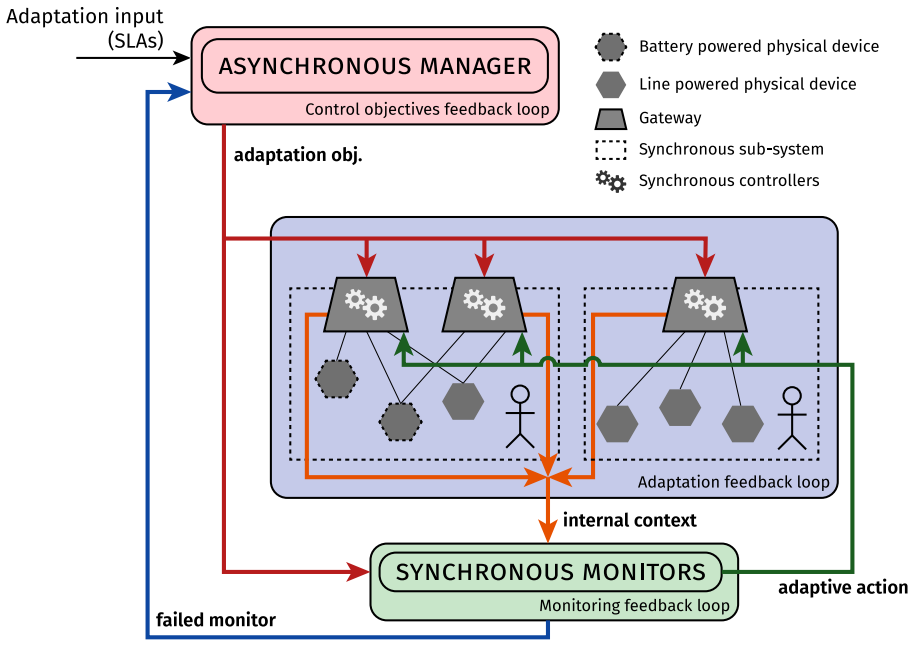


Figure 5.10 – Simplified hybrid self-adaptation framework

Asynchronous CEP engines can consequently be used to analyze streams of dynamic self-adaptation triggering events, such as failed monitors. Application logic can then be implemented in order to redistribute event streams loads (or processed version of the events) to relevant IoT-based sub-systems. Additionally, we also promote the use of synchronous and lightweight CEP engines for the implementation of our synchronous monitors as described later in the implementation chapter. Indeed, since CEP engines generally analyze events using various application-specific rules, they can be used to specify and implement advanced software monitors which derive relevant non-functional data about sub-systems through the combination of several streams of events. More details about the practical integration of CEP engines to implement monitoring logic are given in the next chapter.

In conclusion, we extended our original synchronous dynamic self-adaptation framework to include asynchronous adaptation objectives management capabilities. This improves the overall system scalability as we can now consider the asynchronous coordination of synchronous IoT-based sub-systems accessed trough gateways connected to smart devices of such systems. We can consequently still rely on advanced discrete controller synthesis capabilities provided by SPL while preserving the dynamic nature of the framework.

5.6 Conclusion

In this chapter, we introduce our dynamic self-adaptation framework. We start with a fully synchronous version implementing separation of concerns of adaptation objectives, self-adaptation process and monitoring infrastructure. Based on this framework, we are able to manage changes in adaptation objectives and monitoring infrastructure, which is of particular interest in IoT-based systems since they continuously interact

with the always-evolving physical world and comprise dynamic networks where smart devices can join or leave the systems at runtime.

Self-adaptive behavior is implemented with synchronous discrete controllers. Practically, we consider IoT-based systems as reactive systems reacting to external solicitations mandating self-adaptation of the smart devices included in the systems. Each smart device can be modeled as a specific LTS representing its self-adaptive capabilities. Such models can be specified using SPLs, which make possible to automatically generate discrete controllers. In fact, such controllers are C or Java imperative programs that can be easily integrated into systems as means to provide self-adaptation.

However, we encounter limitations of the purely synchronous systems, principally in terms of scalability. Indeed, for a system to be considered synchronous, it must be able to acquire and process streams of input events fast enough. This represents a disadvantage when it comes to IoT-based systems as they can account for hundreds or thousands of devices, and the computational resources to process streams of generated events becomes unreasonably high for an implementation on standalone gateways. Additionally, discrete controller synthesis relies on the state space exploration, and the time needed to perform this exploration increases with respect to the number of devices.

In order to tackle such limitation, we present our hybrid self-adaptation framework, comprising asynchronous and synchronous capabilities. Namely, we implement asynchronous adaptation objectives management along with synchronous self-adaptation and monitoring. This leads to the asynchronous coordination of synchronous subsystems, and thus a Globally Asynchronous Locally Synchronous system. Synchronous sub-systems handle asynchronous events by buffering them and processing them using the first-in-first-out logic. This improves system scalability as we can now rely on fully distributed self-adaptation and non-functional properties monitoring feedback loops, along with a centralized adaptation objectives management. Centralizing adaptation objectives adaptation cohere with the fact that most IoT-based systems are built to answer specific problems, and unique access points for system managers are preferred.

However, our framework is confined at the adaptation of the IoT higher layer. Indeed, we assume self-adaptation LTSs to be hard coded into the smart devices' firmware. This represents a limitation as advanced self-adaptive behavior may require broader change of such devices, thus mandating modifications of the self-adaptive LTSs. Additionally, we only use DCS as means to generate discrete controllers to be deployed in the gateways. This mechanism is of particular interest as it generates correct-by-construction code, and it would be interesting to include automatically generated discrete controllers to advanced smart devices requiring the concurrent execution and coordination of several LTSs modeled as synchronous nodes. Please note that even though the code is correct because controller synthesizers ensure that it accurately implements the models of the system, this should not be understood as correct execution, considering the synchrony hypothesis might be violated during run time for a variety of reasons (e.g., increase in network latency, smart device processor overload, etc.).

In addition, even though our rule-based adaptation objectives language improves the declarativity of systems' goals specification, it lacks a formal declarative semantics and it is still oriented at low-level smart devices manipulation. Declarativity is however improved as our rule-based approach shifts the focus of IoT-based systems developer from specifying smart devices coordination's execution flows to the specification systems being designed objectives. The lack of declarative semantics prevents

comprehensive language analysis, and questions such as rules conflict or rules data persistence cannot be formally answered. Authors such as in [CDR14] attempted at providing solutions to avoid rules conflicts or circular execution of rules using their own synchronous programming language, but such approach are not broad enough and lack the formality of a full-fledged declarative semantics.

Yet another limitation comes from the fact our self-adaptation case-study only includes sensors and gateways. Actuators are not considered, which limits its strength from a control theory perspective. Indeed, the only interactions illustrated are simple reconfiguration, which do not accurately capture the full dynamics of controlled systems, where the modeling and characterization of actuation on the physical world is the principal subject of study. However, the reconfiguration of sensors considered in our case-study is a good representation of adaptation scenarios, as they comprehensively represent the range of digital actions available to adapt the system to a wide variety of scenarios. As a consequence, the comprehensive (both from theoretical and practical standpoints) actuators to our case-study represents a promising research direction as they represent components of interest in the broadening of our adaptation framework.

Yet another limitation comes from the fact our self-adaptation case-study only includes sensors and gateways. Actuators are not considered, which limits its strength from a control theory perspective. Indeed, the only interactions illustrated are simple reconfigurations, which do not accurately capture the full dynamics of controlled systems, where the modeling and characterization of actuation on the physical world is the principal subject of study. However, the reconfiguration of sensors considered in our case-study is a good representation of adaptation scenarios, as they comprehensively represent the range of digital actions available to adapt the system to a wide variety of scenarios. As a consequence, the addition (both from theoretical and practical standpoints) actuators to our case-study constitutes a promising research direction as they represent components of interest in the broadening of our adaptation framework.

Finally, a future research direction for our self-adaptation framework would be the inclusion of techniques derived from control theory. Indeed, our approach lacks typical concerns of classical control, such as the comprehensive study of the system's dynamics, inertia and stability [Dut+10]. Even though our framework offers the automatic generation of discrete controllers, we did not consider the impact of various physical parameters such as network throttling, variations in network latency or the effect of sensors mobility on the global performance of the system. Moreover, we assumed actuations (i.e., reconfigurations) to be instantaneous. As a consequence, controllers generated by our system are only considered correct by construction in use-cases where all these parameters produce no measurable effect our framework. The study of our adaptation infrastructure could be extended with concepts derived from queuing theory [Hel+04] for its computationally-intensive parts such as the comprehensive characterization of the asynchronous-to-synchronous subsystems interface, while hybrid automata from the Cyber-Physical Systems (CPS) community could be used to study the interaction of smart devices with the physical world.

In the next chapter, we present the prototype of our dynamic self-adaptation framework, with a particular focus on tools and methods we used to implement the different feedback loops necessary to achieve full-fledged dynamic self-adaptation.

CHAPTER 6

Implementing the Self-Adaptation Framework

Software implementation and experiments

Contents

6.1 Introduction . . . 121
6.2 Selected Technical Solutions . . . 123
6.2.1 The Data Distribution Service Standard . . . 123
6.2.2 Asynchronous Reactive Systems Using Vert.x . . . 125
6.2.3 The Drools Rule Engine . . . 127
6.2.4 The Maven Software Management Tool . . . 129
6.2.5 MongoDB Database . . . 129
6.3 Implementation Architecture . . . 130
6.3.1 Global Architecture . . . 130
6.3.2 Implementation Model . . . 131
6.3.3 Implementing the GUI . . . 136
6.3.4 Deployment Life Cycle . . . 137
6.4 Implementation Evaluation . . . 137
6.4.1 Case Study . . . 138
6.4.2 Experimental Results . . . 139
6.5 Conclusion . . . 143

6.1 Introduction

In this chapter, we discuss the Proof-of-Concept (PoC) of our dynamic self adaptation framework and its implementation. As mentioned in the previous chapter, our framework comprises three interacting MAPE-K loops achieving separation of concerns between management of adaptation goals, actual self-adaptation mechanisms and non-functional monitoring infrastructure. These loops communicate through four

interactions. Additionally, we deploy synchronous controllers and monitors on simple gateways such as Raspberry Pis, while the adaptation objectives management is centralized onto a powerful server.

Several elements are of importance when it comes to the implementation of our framework. In fact, we set requirements that assist us with the selection of appropriate tools, protocols and software libraries:

- *Scalable and reliable communication*: As discussed above, the three feedback loops defined in our system are constantly communicating. Smart devices are generating event streams that our framework must analyze in real time in order to decide if the adaptation of our IoT system is required, or to decide if changes in the monitoring infrastructure or adaptation objectives should be considered. Because IoT-based systems are often used in critical application cases, communication must be reliable both between the feedback loops and between gateways and smart devices. Even though it is not realistic to compel with the adoption of any particular communication protocol between smart devices and gateways because of the variety of existing low-energy wireless communication protocols, we decide to promote the use of single communication protocol for the higher level of our systems as means of minimizing interoperability issues. Because the self adaptation framework should manage numerous devices, the communication protocol for the higher layer must be easily scalable. This practically means that Quality of Service (QoS) characteristics such as latency or throughput should be independent of the amount of managed smart devices.
- *Complex Event Processing (CEP) capabilities*: As mentioned in the previous chapter, real-time analysis of numerous concurrent streams of events can be tackled using CEP techniques. In such techniques, end-users typically define event-driven actions to achieve application-specific objectives. By using such technique, we specify advanced monitoring logic, deriving relevant information about the system's non-functional state. We also use them to perform smart functional adaptation as an enabler of smart services. Additionally, CEP software are traditionally designed to handle massive streams of events, which makes their use relevant in an IoT-based context where multiple heterogeneous data streams are produced by numerous smart devices.
- *Interoperable*: Eventually, we require our smart devices to be as interoperable as possible. In this context, interoperability is two-fold: first, it ensures communications between all elements of the system should occur seamlessly without the need for complex protocol conversion software. Then, we enable interoperability regarding the generated controllers. Indeed, the dependency on automated code generation to build discrete controllers should not be a limitation when it comes to the deployment of our system. For instance, we consider that generated controllers should run on a wide range of gateways such as Raspberry Pis, Intel Edisons or traditional computers. Additionally, self-contained software deployment is particularly appreciable in distributed environments, as it prevents concerns about dependencies installation or integration.

In addition, we require for our prototype to be lightweight, as it should execute on low-cost and relatively resource-constrained gateways. It is worth noting that here, resource-constrained means computational limitations when compared to state-of-the-art computers running last generation processors. Furthermore, we aim at integrating

a full-fledged Graphical User Interface (GUI) in order to empower casual users with capabilities to specify self-adaptation strategies with minimal efforts.

Relying on Synchronous Programming Language (SPL) has implications in terms of practical implementation of our framework. Indeed, Discrete Controller Synthesis (DCS) and the modeling of smart devices as synchronous nodes are the cornerstone of our self-adaptation system, and they consequently should be seamlessly integrated with our framework. We thus adopt a middle-out approach to prototype our framework, where we progressively implement the distributed components of our framework from Labeled Transition Systems (LTS) -based modeling of smart devices.

With such requirements in mind, we design and implement a PoC of our dynamic self-adaptation framework. In Section 6.2, we describe selected technologies, and the overall framework architecture is described in Section 6.3. We then evaluate our implementation in Section 6.4. Eventually, we conclude this chapter with an executive summary of our prototype and discuss its limitations in Section 6.5.

6.2 Selected Technical Solutions

In this section, we present the choice of technical solutions along with the motivation behind their selection. More precisely, we identify the advantages of each tool, but also how it contributes with respect to our framework requirements identified in introduction section. Please note that we tend to choose open-source solutions as they commonly feature very active development and communities, which can later benefit the wide adoption of our framework in the IoT community. Additionally, we adopt the Java programming languages for cross- operating systems compatibility (i.e., heterogeneous gateways). It is worth noting that the Heptagon/BZR SPL also generate Java codes to deploy discrete controllers, which constitutes further motivation for the use of this language.

6.2.1 The Data Distribution Service Standard

As mentioned in the previous section, we require that all system communications are scalable and reliable. To this end, we avoid communication protocols featuring single points of failure as they can potentially decrease system reliability. We adopt publish-subscribe -based solutions as they can easily be integrated to event-driven frameworks and typically exhibit good scalability. Another advantage of publish-subscribe solutions is that smart devices in IoT-based systems commonly exhibit event-based behaviors rather than query-based behaviors, where devices self-determine when measurements should be transmitted to external tiers.

Numerous publish-subscribe -based protocols can be found in the IoT landscape, such as Message Queuing Telemetry Transport (MQTT) [1] , Advanced Message Queuing Protocol (AMQP) [2] or Extensible Messaging and Presence Protocol (XMPP) [3]. However, these solutions are centralized and typically rely on a central broker for message distribution. This centralized nature represents a drawback for their use in critical applications, as failures of the broker implies failure of message distribution. Additionally, even

1. MQTT standard: `http://mqtt.org/` (visited on 07/22/2018)
2. AMQP standard: `https://www.amqp.org/` (visited on 07/22/2018)
3. XMPP standard: `https://xmpp.org/` (visited on 07/22/2018)

though some broker redundancy can be introduced, this process commonly occurs manually, and every element of the system should be reconfigured in order to include redundant brokers addresses and configuration parameters.

In opposition, the Data Distribution Service (DDS) [4] standard is a brokerless, decentralized middleware aimed at the development of distributed real-time applications. The DDS is divided into the DDS Application Programming Interface (API), which guarantees source code portability independently of underlying vendor implementations, and the DDS Interoperability Wire Protocol (abbreviated as), which ensures low-level interoperability between different implementations.

Data exchange in the DDS relies on a fully distributed Global Data Space (GDS) with dynamic publisher and subscriber discovery. The GDS is accessed by declaring Domain Participants (DPs), which are indexed using integer IDs. Data flows between devices are based on topics, which are made of a name, a type and a predefined quality-of-service. Topics types are specified using a subset of the OMG Interface Definition Language (IDL), which adopts a C-like definition of data types using structures. This interface specification facilitates system integration as systems' components communicate using an agreed-upon data type. This comes in opposition with the MQTT protocol, where the payload format is not specified, and system components must use external knowledge to aggregate exchanged data.

Additionally, DDS features extremely flexible and precise QoS management through the definition of 22 QoS policies. QoS properties can be defined either for topics, publishers (also known as writers in the DDS lexicon) or subscribers (i.e., data readers). They are defined with respect to volability, infrastructure, delivery, user QoS, presentation, redundancy and transport. Furthermore, even though most QoS are only defined for specific instances of DDS elements, a number of properties can cause incompatibility issues if communicating elements do not share the same QoS property values. For instance, the *reliability* QoS should be the same for all system elements, and if a subscriber does not have the same reliability parameter value that a publisher, the two elements are not able to communicate. These flexible QoS management empowers us with the capability to further improve the communication reliability in our IoT system.

Furthermore, DDS implementations commonly offer data reading through direct query or by attaching a listener function to the data reader object. The listener function is simply a callback function assigned to the reader, which is called every time new data is available, while the query based mechanism is typically used to read data through polling by periodically interrogating the reader for the presence of new data. Both solutions offer advantages, but we chose the listener-based approach as it offers better event management, minimizes processor load, and because our global architecture is strongly event-driven.

The DDS protocol specifies hierarchical information scopes. The global information scope is called a domain, and inter-domains communication is not specified in the protocol and must be interceded in the user application. A domain can contain several partitions. Partitions are finer information scopes and provide a flexible mechanism to divide information into logical sets. Publisher and subscribers can then join partitions by either providing the partition full name, or by providing a regular expression matching all partitions they want to join.

4. DDS standard: `http://portals.omg.org/dds/` (visited on 07/22/2018)

In addition, DDS performance evaluation confirms its relevance for its use in critical IoT-based applications [CK16; BPM16]. Indeed, even though it introduces overhead in order to increase communication reliability, it achieves acceptable latency and packet loss rate. In particular, authors of [BPM16] found it performed better than MQTT and XMPP in terms of message throughput, while authors of [CK16] showed DDS provided better latency than MQTT in low-quality networks.

Because DDS is a standard technology, numerous implementations are available both under commercial or open-source licensing. Since we focus on adapting open source solutions, we have chosen OpenDDS and Vortex OpenSplice DDS. Additionally, Real-Time Innovation (RTI) offers a free version of their product (Connext DDS) for inclusion in open-source projects, but this product remains closed-source. OpenDDS focuses exclusively on a C++ based implementation of the DDS protocol. Even though bindings to the Java programming language are available through Java Native Interface (JNI), they are difficult to configure and are a suboptimal solution. In opposition, Vortex OpenSplice DDS Community (ADLINK Technology Inc., Taipei, Taiwan) implements full DDS standard in several languages such as Java, C, C++ or Python. Additionally, this DDS implementation was recently integrated to a new Eclipse project under the name Eclipse Cyclone DDS. This leads us to choose the ADLINK DDS implementation, as Eclipse projects are commonly well distributed and well maintained.

6.2.2 Asynchronous Reactive Systems Using Vert.x

As mentioned in the previous chapter, our system comprises asynchronous and synchronous capabilities, while being strongly event-driven. While events handling could occur manually or through polling in the synchronous case, it is more convenient to work with a global framework in order to have homogeneous event handling. Additionally, in order to avoid overhead, we require internal events (i.e., events used for the coordination of the elements in the feedback loop) not to use the DDS protocol.

To this end, we investigate a tool for the homogeneous implementation of reactive systems, which led us to the choice of the Vert.x[5] tool-kit. Using Vert.x, developers can implement flexible, scalable and lightweight reactive systems with an asynchronous non-blocking development paradigm. In addition, Vert.x provides non-blocking libraries for the integration of various tools and technologies, such as a basic Hypertext Transfer Protocol (HTTP) server, HTTP templating engines, or various database systems.

Because Vert.x is only a tool-kit and not a comprehensive framework, it is effectively lightweight (the Vert.x core only takes 650 kB of space) and runs on various hardware, ranging from powerful servers to low-cost single board computers. Additionally, even though the Vert.x core module needs the Java Virtual Machine (JVM) to be installed in order to be executed, interfaces to numerous programming languages are available (such as JavaScript, Kotlin or Ruby just to name a few).

The strength of Vert.x principally lies on its advanced computing model, namely, non-blocking asynchronous and event-based development. Particularly, concurrency is achieved by attaching handlers to events, which are managed by an advanced event loop running in the background and which executes the functions attached with the event it receives. One can trivially notice that this programming paradigm is ideal for the specification of reactive systems, which are by nature required to react to various

5. Eclipse Vert.x tool-kit: `https://vertx.io/` (visited on 07/23/2018)

sets of external events. Additionally, the use of an event-loop to manage external events improves system scalability. Indeed, in opposition with the classical synchronous and blocking approach by which every concurrent element of the system is assigned to a dedicated thread, the asynchronous non-blocking approach can handle more events using a minimal number of threads [Esc17]. Practically, Vert.x implements a variation of the traditional reactor pattern called the multi-reactor pattern, where several event loops are used to manage events and call appropriate handlers [Esc17]. This computational paradigm comes with a condition: the event loop should never be blocked. In order to tackle this strong requirement, and if blocking operations are still necessary, Vert.x provides a mechanism to manage blocking computations safely.

Furthermore, Vert.x reactive components are specified in *verticles*, which are an actor-like deployment model and can be seen as a self-contained reactive component. Verticles specify reactions to various events, and events handler are commonly represented as callbacks. They also provide a mean to increase the modularity of Vert.x-based solutions, as they can be used to implement self-contained modules with a single functional purpose. In addition, Vert.x offers an internal communication bus called an *event bus*. Essentially, this bus is the backbone of Vert.x-based asynchronous systems, and it uses a publish-subscribe communication paradigm. A single event bus is deployed in each Vert.x instance (i.e., instance of the Vert.x core managing events handling), and it can be used for communication between verticles.

Other frameworks or tool-kits for the implementations of asynchronous reactive systems can be found, such as Node.js, Netty or ReactiveX. Node.js is a JavaScript-only non-blocking asynchronous framework for the development of Web servers. Even though it is widely used, the poor availability of advanced debugging and code analysis tools for JavaScript prevents its adoption in critical context, where minimal code quality should be guaranteed. Netty is a low-level asynchronous networking framework, and Vert.x actually relies on it for its asynchronous event management implementation. However, Vert.x offers various high-level libraries that facilitates its integration with external software, and consequently presents a better comprise between ease-of-use and computational efficiency. Eventually, ReactiveX provides generic asynchronous frameworks based on the observer pattern. It also defines observable sequence of events, which are processed and associated with various actions. The principal advantage of ReactiveX is that it decreases the use of callbacks, which can reduce code legibility if overused. However, the libraries offered by ReactiveX are very generic, and do not offer integration with external software or Web server capabilities.

As mentioned in the introduction, we require a Web-based GUI in order to empower end-users with easy adaptation objectives specification. Vert.x offers full-fledged Web-server capabilities, which further motivated its adoption. In addition, Vert.x supports various Hypertext Markup Language (HTML) templating engines to dynamically generate Web pages according to internal server information. Indeed, the GUI needs to accurately represent the system being managed, and it thus needs to change depending on server-side information. With a templating engine, we are able to implement generic Web pages, that are then automatically generated according to internal information through the Thymeleaf template engine when the Vert.x server receives an appropriate HTTP request from a client.

6.2.3 The Drools Rule Engine

Generalities

Drools [6] is a Business Rule Management System (BRMS), providing both rules management and a rule execution engine. The rule execution engine is based on an improved version of the Rete algorithm. It provides a full-fledged approach to rule specification, and it is an open source project part of the WildFly application server. Drools defines two core concepts of rules management and execution [SMA16]:

- *Data model*, which designates the global knowledge on which the rules are applied. Practically, this is implemented under the form of a set of Java classes.
- *Rules syntax*: In Drools, rules have two sides, the left-hand side (LHS) and the right-hand side (RHS). Expressions in the LHS declare constraints on the data model (i.e., constraints on the classes' attributes defined in the data model), and they are also called *conditions*. Expressions on the RHS, also designated as *actions*, declare actions to be performed on the data model.

Practically, Drools uses the KIE (Knowledge is Everything) API for knowledge representation, management and execution. Rules are specified using the Drools Rule Language (DRL), and the basic structure of the language is given in Figure 6.1.

```
1  package example // package defining the knowledge base
2
3  import classes // import any Java classes
4
5  global // keyword defining global variables
6
7  rule "Example rule name"
8    when
9      // conditions
10   then
11     // actions
12 end
```

Figure 6.1 – Basic Drools rule syntax example

Each rule is designated by a unique name, given after the `rule` keyword. Conditions are given after the `when` keyword, and actions are specified using Java code after the `then` keyword. The package that implements the knowledge base is given after the `package` keyword. Eventually, it is possible to import any Java classes using the traditional `import` keyword, and global variables can also be defined after the `global` keyword. Such variables are typically implemented in order to use external application objects inside rules. They typically invoke remote services or log information in order to gain knowledge over the system execution. This syntax allows end-users to only focus on rule specification, without the need to specify the complete system flow to achieve desired behavior. A complete description of the Drools rule syntax is given herein bellow.

In Drools, business knowledge and facts are embedded in a `KieModule`. Each `KieModule` contains `KieBases`, which define the application's entire knowledge (i.e., rules, type models, functions, etc.). All the `KieBases` of a given `KieModule` are contained into a `KieContainer`. However, this data organization model represents offline data. Runtime data (i.e., dynamic business data) is provided to the rule engine us-

6. Drools documentation: `https://www.drools.org/` (visited on 07/22/2018)

ing `KieSessions`. `KieSessions` are the main mechanism to provide interactions between the rule engine and the data model, and they make use of definitions specified in `KieBases` to feed the rule engine with appropriate data. Practically, `KieSessions` can be instantiated directly, and the appropriate `KieBases` instanciation will be cached by the API.

Drools is able to manage dynamic business logic, where rules change frequently, using the `KieScanner` object. This object implements synchronous or asynchronous changes monitoring in the compiled business assets (i.e., compiled version of business rules). Change detection is based on a traditional versioning and follow the Apache Maven [7] specification. These capabilities are of prime interest in our framework, since such mechanism can be used to manage and dynamically react to changes in the adaptation objective rules without any system downtime.

Drools Rule Language

In addition to the simple rule syntax given in Figure 6.1, the DRL features advanced capabilities that improve the declarativity of complex and large scale business applications specification. More particularly, the DRL provides mechanisms to specify modifications of the data embedded in instances of the data model, but also interactions with the external world, advanced configuration parameters and rule execution control.

The main mechanism to refer to external tiers from rules are global variables, that can be specified after the keyword `global` in DRL. External tiers are typically services (e.g., Web services), and the global variable declaration syntax in DRL is similar to traditional Java variable declaration (i.e., **`Type`** `varName`).

As business application grow larger, rules can become very complex, with very long condition lists and actions. In order to improve maintainability, this must be avoided by separating complex rules into smaller and simpler rules. As the Drools approach aims to be as declarative as possible, there are no mechanisms that specify rule calls. The orchestration of several rules rather happens though the modification of the data in the working memory.

However, several imperative keywords such as `import`, `modify`, `update`, `delete` or `retract` were introduced. They empower developers with more latitude regarding the implementation of large-scale systems but are declarative in nature because they are used to specify the rules' flow of execution by, schematically, modifying data that will trigger the execution of other rules. Using these keywords, rules can be chained, and they can interact together through modifications of data in the rule engine session. As a consequence, this approach is closer to classical imperative programming languages, it improves the overall flexibility of systems specification.

CEP Capabilities

In addition to being an efficient and comprehensive rule engine, Drools also features CEP capabilities. Namely, Drools is able to process streams of events in order to detect changes requiring actions, and implements it through rule-based event processing. It defines two categories of events: *punctual events* and *interval events* [SMA16], the first being used to describe events that have a single occurrence in time, while the latter is

7. Apache Maven: `https://maven.apache.org` (visited on 07/22/2018)

used for events associated with a beginning date and an end date. Events are immutable (i.e., they cannot be changed after storage), and are associated with a life cycle (i.e., events that are too old to be relevant to the application should be removed from the knowledge base in order to save space).

Practically, information inserted in the knowledge base should be assigned the role *event* to allow the rule engine to apply event-based reasoning and process it. Optionally, events can be explicitly associated with a time stamp, a duration used to define interval events, and an expiration date.

When it comes to event processing, Drools defines a variety of temporal operators (e.g., `after`, `before`, `overlaps`, etc.). It also defines sliding windows, which can be specified with respect to length (i.e., the discrete number of events inserted into the knowledge session) or time. Often used in combination with sliding time windows is the `accumulate` keyword, which can be used to count facts or events based on specific filters.

In our specific application, CEP capabilities are particularly useful for the specification of advanced non-functional monitors, which can deduce non-functional state of the system through the analysis of events (either functional or non-functional) generated by the smart devices included in the controlled systems.

6.2.4 The Maven Software Management Tool

Because our prototype is required to be flexible and easily deployable on a variety of architectures, we chose to use a software management tool. When it comes to Java development, the two principal options are Maven and Gradle. The reliance of the Drools rule engine on Maven lead us to chose this software development over Gradle.

Maven provides full-fledged build automation for Java projects. Practically, it uses the "convention over configuration" paradigm, which encourages software developers to adopt specific conventions as means to decrease development time by reducing available options. Practically, Maven specifies a standard folder architecture that developers must comply with in order to use this build automation tool. In addition, Maven offers advanced dependency management through central repositories, and most of the dependencies can automatically be fetched from such remote repositories and placed into a given project. Maven also comes with numerous plugins to extend its capabilities, and they can be used to better control the build process. Please note that it is also possible to specify unitary tests within the build process, and that the project build will fail if test does not complete successfully.

In Maven, the entire build process is specified in an Extensible Markup Language (XML) file called `pom.xml`, placed at the root of the project directory. This file declaratively describes various project's parameters (such as project name, inclusion of sub-projects, project version, etc.), but also of the project's dependencies and build strategy (e.g., build only a minimal version or a self-contained version including all dependencies). Practically, projects can then be built and run using very simply Maven commands.

6.2.5 MongoDB Database

Even though we emphasize the importance of a distributed architecture, there is still a need for central information storage in case the system is shut down and must be

restarted with its previous configuration. To this end, we chose to integrate MongoDB in our solution due to its Structured Query Language (SQL)-like queries, its replication model favoring data consistency, and the possibility to organize documents in collections. Additional, independent benchmarking showed its optimal performance in terms of read and write operations [PND14].

Practically, we use JavaScript Object Notation (JSON)-based description of smart devices, which contains the synchronous specification of the node they implement in terms of inputs, outputs and automaton. We also used JSON to store rule-based adaptation objectives, and the structure of the data follows our language's grammar. In the current state of our prototype, rules are specified in terms of rule name, QoS name, predicates and adaptive actions.

6.3 Implementation Architecture

In this section, we give a comprehensive description of the implementation of our dynamic self-adaptation framework. We start with a global architectural overview of our implementation, and we then proceed to the complete description of all the elements of the implemented framework.

6.3.1 Global Architecture

The general architecture of our self-adaptation framework is given in Figure 6.2. As depicted in the figure, DDS is the backbone of our architecture, and it is used to implement all interactions described in the previous chapter, namely the adaptation objectives (called non-functional adaptation objectives in Figure 6.2), the failed monitors, the adaptive actions and the internal context. The three MAPE-K loops of our framework are communicating though these DDS interactions (which are, in fact, DDS topics). The adaptation objectives feedback loop is also interacting with the objectives directory, which stores predefined adaptation goals, which is itself interacting with our Web-based GUI. Indeed, we chose to interface interactions between the feedback loop and the GUI with the objectives directory, as it represents a central element storing all the existing rules in the system. In fact, when a user adds a new adaptation rule in the system, it is directly stored in the database. The objectives feedback loop then detects a database change event, and propagates this new rule to all the other parts of the framework.

The adaptation feedback loops interacts with the device directory. This database stores representations of smart devices in the IoT-based system. In our specific application, it stores the LTSs of all smart devices, that can be used to perform discrete controller synthesis when combined with a set of adaptation objectives. Eventually, the monitoring feedback loop interacts with the Drools rule engine. Indeed, we used advanced CEP capabilities of Drools to implement a declarative specification of complex software monitors, which use event streams from the system to deduce non-functional states.

Table 6.1 summarizes the software used to implement this prototype, along with their versions. Please note that for Vert.x and Drools, the use of newer version simply implies changing the version number in the Maven `pom.xml` file. By doing so, Maven will automatically update the version of these libraries. For all other software, updates should occur through the package management software (on GNU/Linux operating

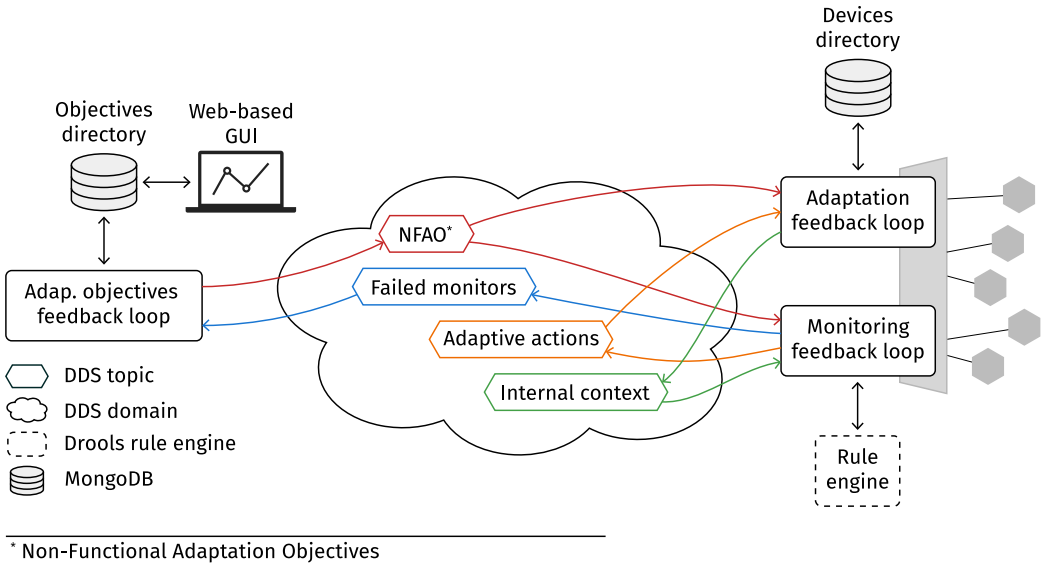


Figure 6.2 – Simplified implementation diagram

systems) or manually (on Windows). Additionally, Heptagon/BZR can be updated using opam, the OCaml package manager. Please note that backward compatibility should however be verified before updating to newer version of the libraries.

Table 6.1 – Prototype dependencies names and versions

Software Name	Version
Java JDK	1.8.0_172
Vortex OpenSplice DDS Community	6.7.180404
Maven	3.5.2
Drools	7.1.0.Final
Vert.x	3.5.1
Ocaml	6.05.00
Heptagon/BZR	1.4.00

6.3.2 Implementation Model

The Maven modules implementing the architecture given in Figure 6.2 are represented in the dependency diagram given in Figure 6.3. The overall system comprises a single parent module and 8 sub-modules. The parent module is called `org.adapiot`, and it is only used to define framework-wise dependencies and build strategies. The three feedback loops are implemented in the `org.adapiot.objectives`, `org.adapiot.monitoring` and `org.adapiot.nf_adaptation` sub-modules. The GUI is implemented in the `org.adapiot.gui` module. The `org.adapiot.dds` and `org.adapiot.configuration` are helper modules, implementing high-level functions to either configure the system at a global level or easily access DDS communication functions. Eventually, the `org.adapiot.logger` module implements framework-wise logging mechanism, while the `org.adapiot.device` is used to model devices. It is worth noting that the three feedback loops sub-modules are not interdependent, thus making them truly self-contained.

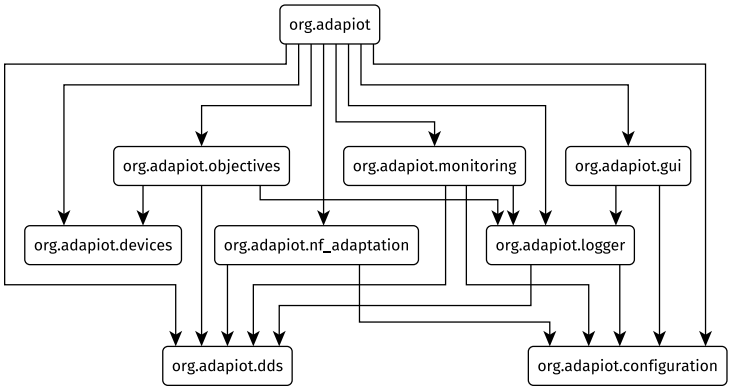


Figure 6.3 – Implementation dependency graph

In the remaining section, we focus on the implementation of the three feedback loops, as they are the backbone of our dynamic self-adaptation framework. These loops comprise 17 classes, and cross-modules communications are handled using the helper functions of each DDS-dedicated module.

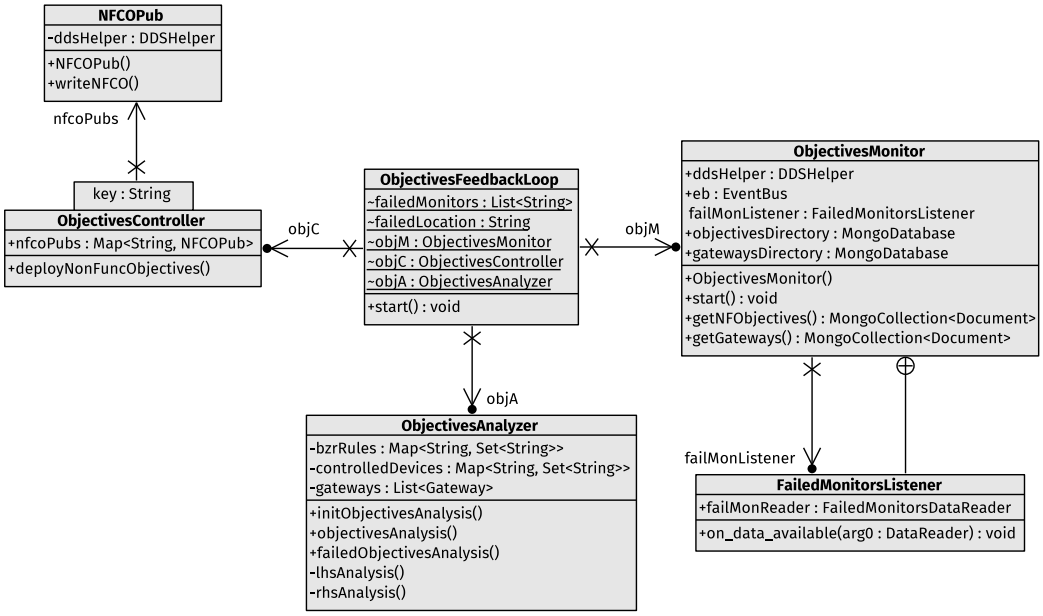


Figure 6.4 – Objectives feedback loop class diagram

The first module implements the objectives feedback loop with 6 classes, detailed in Figure 6.4 and Figure 6.5:

- The `ObjectivesMonitor` class, which implements a set of sensors that detect if new objectives are added by external users in the objectives database, or if changes in the monitoring infrastructure mandate changes in the adaptation objectives.
- The `ObjectivesAnalyzer` class, which determines if the objectives are specified with respect to the functional or non-functional properties of the system. This analysis is mandated by the different ways we handle functional and non-functional adaptation. While functional adaptation is managed by the Drools rule engine, non-functional adaptation rests on the use of automatically synthesized

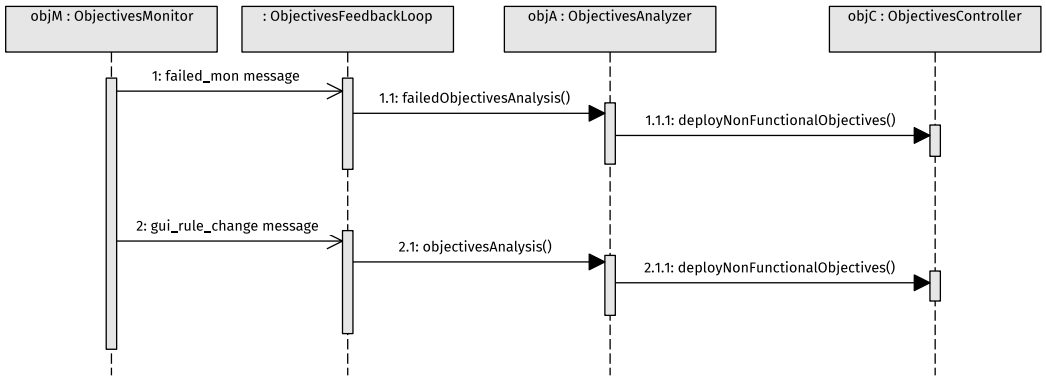


Figure 6.5 – Objectives feedback loop activity diagram

discrete controllers deployed on gateways, and the objectives for these separate kind of adaptation must be handled distinctively.

- The `ObjectivesController` class, which is in charge of objectives deployment in either the rule engine or in the gateways, depending on the nature of the new objectives. The deployment occurs through the publication of a message with the new objectives on the appropriate DDS topics.
- The `ObjectivesFeedbackLoop` class, which coordinates the operation of the three classes described herein above. This class is in fact a Vert.x verticle, and the coordination between the elements given above occurs asynchronously.
- The `NFCOPub` is a DDS data writer class, and it is used to publish the non-functional adaptation objectives on the appropriate DDS topic. This class implements the interaction from the objectives feedback loop to the adaptation and monitoring feedback loops.
- The `FailedMonitorListener` is used to asynchronously receive failed monitors, thus implementing the interaction between the monitoring feedback loop and the objective feedback loop.

Figure 6.5 illustrates the activity diagram, representing the interactions between classes from the objectives management sub-module, and only two events trigger objectives analysis and redeployment. These events are originating from the failed monitor topic or changes in the non-functional adaptation objectives database. They both trigger different actions. Events from the failed monitor topic cause the such monitors to be analyzed in order to remove the defected smart object from the adaptation goals, while events generated by changes in the database only trigger new objectives analysis and integration to the relevant gateways.

In summary, the objective feedback loop sub-module (called `org.adapiot.objectives`) handles the dynamic management of changing adaptation objectives. It communicates with adaptation and monitoring feedback loops though interactions using the DDS publishers and subscribers defined in the `NFCOPub` and `FailedMonitorListener` classes.

The next sub-module implements the non-functional adaptation feedback loop, and it contains 5 classes, displayed in Figure 6.6:

- The `AdaptationMonitor` class, which implements all the necessary monitors to perform non-functional adaptation. In particular, it subscribes to the non-functional adaptation objectives DDS topic. This class provides bindings between

smart devices' streaming services and synchronous controllers, handling the preservation of an acceptable QoS in the entire controlled sub-systems.

- The `BZRWrapper` which is a helper class to build Heptagon/BZR programs from non-functional adaptation objectives and smart devices description.
- The `AdaptationAnalyzer` class performs the actual discrete controller synthesis process. It is used to synthesize Java code implementing discrete controllers.
- The `AdaptationController` class generates a self-contained `jar` file, and deploys it on the appropriate gateways. This executable is remotely deployed using Secure Shell (SSH) File Transfer Protocol (SFTP), and remote operation is securely performed using SSH. Please note that all the necessary library for discrete controller operations are packaged within this file.
- Finally, the `AdapatationFeedbackLoop` class coordinates the operation between the adaptation monitor, adaptation analyzer, discrete controller synthesizer and rule engine runtime classes.

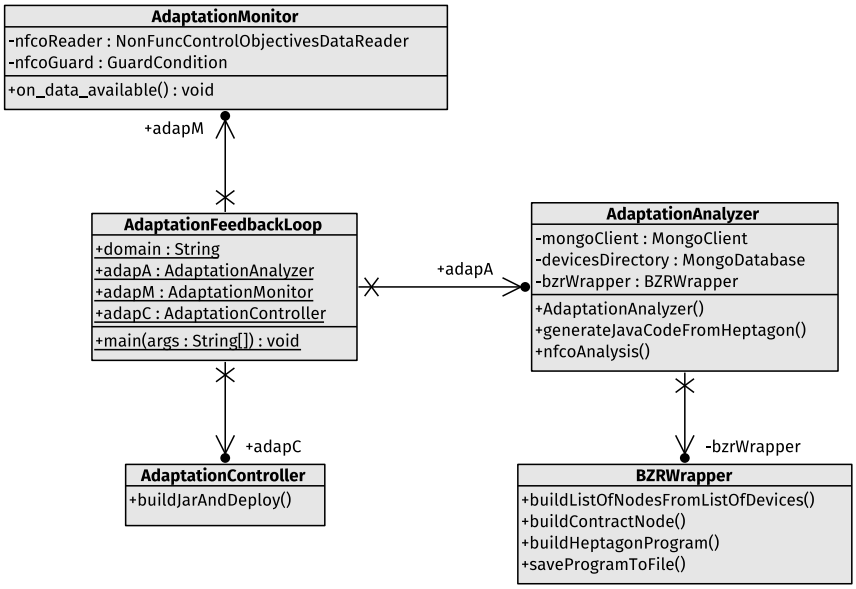


Figure 6.6 – Adaptation feedback loop class diagram

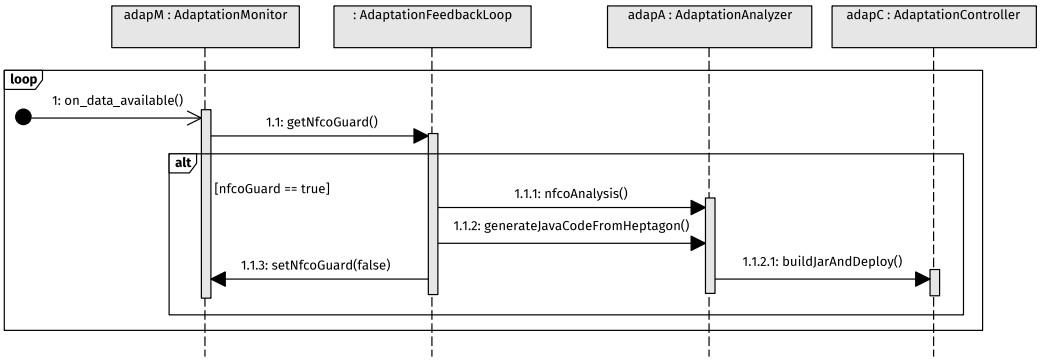


Figure 6.7 – Adaptation feedback loop activity diagram

Figure 6.7 details interactions between classes. One can notice that interactions are synchronous, because they are defined within a loop. This implies we periodically

query data availability in order for us to detect events mandating changes in the self-adaptation strategy. Elements triggering adaptation are the reception of new non-functional adaptation objectives, that trigger objectives analysis, Java code generation, and executable generation and deployment. These operations are blocking, and events should not occur during the computation of a new controller.

In sum, this module (called `org.adapiot.adaptation_fl`) implements full-fledged discrete controller generation for non-functional adaptation. Indeed, this feedback loop self-generates synchronous controllers in charge of self-adaption in response to changes in the external world in order to preserve satisfactory QoS in the controlled systems. This package communicates with the other feedback loops (namely the monitoring feedback loop and objectives feedback loop) through interactions as defined in Figure 6.2 and implemented as DDS communication topics in the DDS dedicated package.

The third module implements the monitoring feedback loop using 6 classes, as depicted on Figure 6.8:

- The `NFPMonitor` class subscribes to the non-functional objectives feedback loop in order to be able to deploy appropriate monitors.
- The `NFCOListener` class is used by the previous class to receive adaptation goals from the objectives feedback loop.
- The `NFPAnalyzer` class infers monitoring rules' relevant non-functional properties from a set of monitored system properties, that will be used in the global self-adaptation framework to perform preventive or corrective self-adaptation.
- The `NFPController` implements the non-functional property soft monitors using purely software probes, implemented as a Drools rule engine instance running rules computed in the previous class. These monitors are used when a non-functional property must be derived from other measured non-functional variables. This class subscribes to the appropriate feeds from smart devices, and failed monitors are published using the class described bellow.
- The `FailedMonPub` publishes failed monitors to the objectives feedback loop in order to trigger control-objectives self-adaptation.
- Finally, the `NFPFeedbackLoop` coordinates the interactions between the three `NFPMonitor`, `NFPAnalyzer`, and `NFPController` classes.

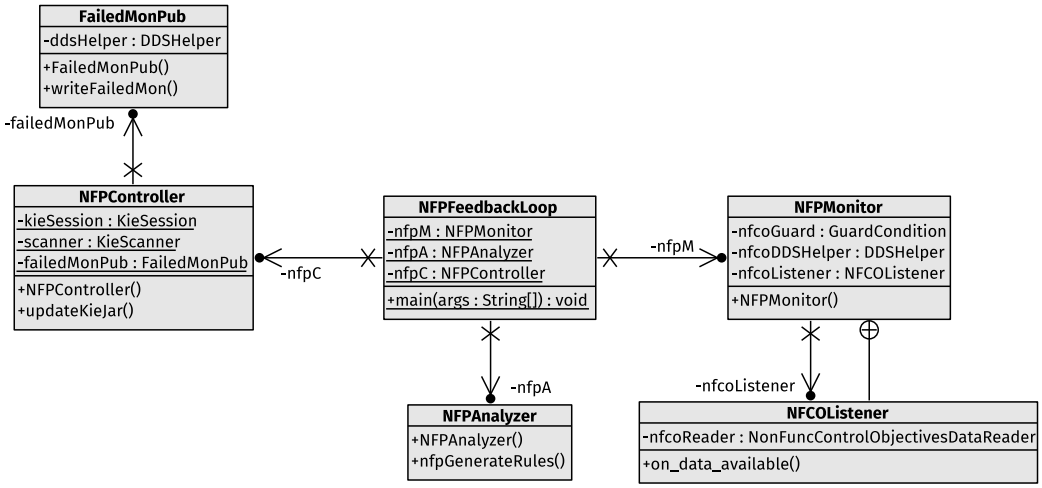


Figure 6.8 – Monitoring feedback loop class diagram

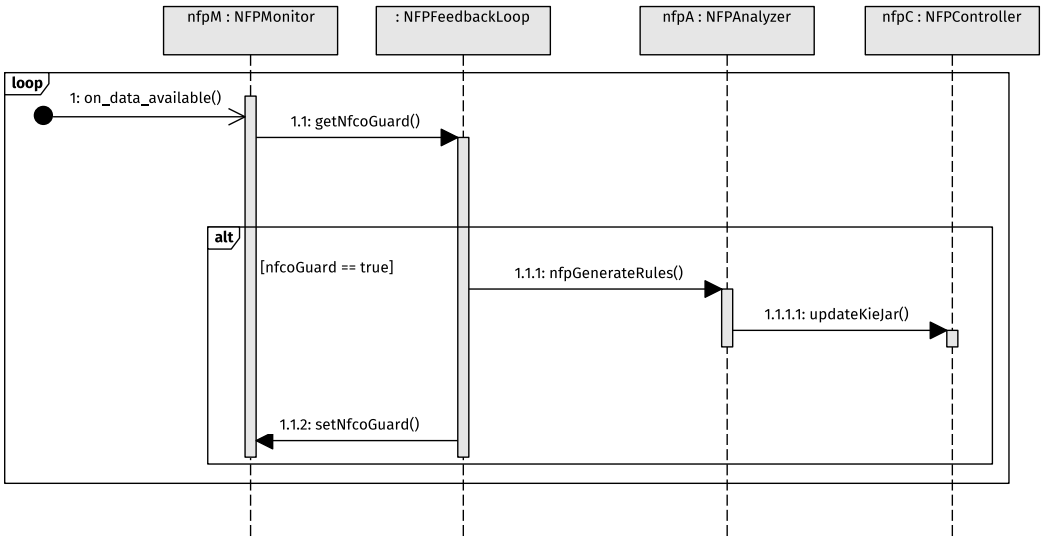


Figure 6.9 – Monitoring feedback loop activity diagram

Figure 6.9 is the activity diagram of the non-functional properties monitoring feedback loop. As the adaptation feedback loop, interactions are synchronous, and they are coordinated though a loop that periodically acquires inputs mandating self-adaptation. When such events are received (namely, non-functional adaptation objectives), new monitoring rules are synthesized and inserted into the running Drools KieSession, consequently deploying on-the-fly monitors on gateways.

In essence, the `org.adapiot.monitoring` module implements the monitoring feedback loop that handles dynamic changes of the monitoring infrastructure. It communicates with the adaptation and objectives feedback loops through the four defined interactions using appropriate DDS topics.

In conclusion, with this implementation model, we have a fully distributed self-adaptation framework implemented in 3 principal modules and communicating using DDS topics. Implementation was performed using traditional Java 8 in the Eclipse development environment. In addition, we also implemented a web-based GUI for end-users objectives specification. The GUI also notifies user about the self-adaptive system internal functional and non-functional states, and the implementation of such component is discussed in the next section.

6.3.3 Implementing the GUI

As mentioned in the introduction, we require our prototype to include a Web-based GUI to offer end-users the capability to easily read information about our system, but also to conveniently write new self-adaptation rules. The choice of Web-related technologies over native technologies such as Qt or GTK to implement the GUI was motivated by the interoperability of Web-based solutions, which only requires a simple Web browser to display the interface.

We thus implement a Vert.x-based server, which generates appropriate interface panels and interacts with the users. Because we wished our GUI to be generic and flexible, we used the Thymeleaf templating engine to dynamically generate Web pages according to information in the database from the server side. In addition, we used the Bootstrap JavaScript and CSS framework to provide responsive Web pages design (i.e.,

the GUI self-adapts to the device it is displayed on, meaning it is adapted to account for narrower screen if it is displayed on a phone for instance). Furthermore, we use the WebSocket protocol for real-time and event-based communication between clients and the server. This protocol provides clients/users with real-time functional and non-functional information regarding the system being controlled. It also forwards to the server relevant interactions from the user with the GUI (e.g., the addition of a new rule).

Practically, the server consists of a single Vert.x verticle reacting to appropriate internal and external events. From a server point of view, external events are generated by the clients through user interactions, and they only consist of rule addition, modification or deletion for our preliminary PoC. If the server detects a rule-related event, it propagates the action (i.e., addition, modification or deletion) to the MongoDB database, and the change of the rules directory is notified to the remaining of the system in order to take appropriate action. Events circulating from the server to clients are sensor states, but also relevant functional information such as patients' health data or system metrics.

6.3.4 Deployment Life Cycle

The architecture displayed in Figure 5.10 only gives a run-time representation of the system, and does not accurately capture the deployment life-cycle, which is displayed in Figure 6.10. This life cycle starts with adaptation objectives specification, which is subsequently used to configure the entire system.

In the IoT system's higher layers, the deployment targets specified in the rules (which can be seen as a filter attribute) are used to generate a deployment strategy. Indeed, for our PoC, we assume that rules are specified with respect to particular *locations*, which can be utilized to differentiate between patients' homes, since they are accessed through gateways used as single access points. Then, rules associated with particular locations are analyzed and transformed into synchronous programs, that are deployed on appropriate gateways according to the deployment targets analyzed by the objectives feedback loop. Eventually, QoS monitors or more advanced software probes are derived from the adaptation objectives specification and are deployed on the appropriate gateways. Please note that the adaptation objectives feedback loop is deployed as an application running on a server, while the adaptation and monitoring feedback loops are deployed on gateways.

However, this deployment life cycle is theoretical and applies to the final version of the system. For the sake of our proof of concept, we implemented a simpler life cycle, in which only the adaptation is dynamically deployed. In the current state of the PoC, the asynchronous manager analyses adaptation objectives to check if locations filter parameters are present. If no location is specified in the rule, we deploy the rule to all the gateways included in the system. The software monitors are hard coded, and they are only remotely enabled or disabled depending on the adaptation goals.

6.4 Implementation Evaluation

In this section, we present the case-study that we used for the preliminary evaluation of our framework. Based on this case study, we subsequently discuss experimental results, both in terms of user interface and quantitative evaluation of our framework.

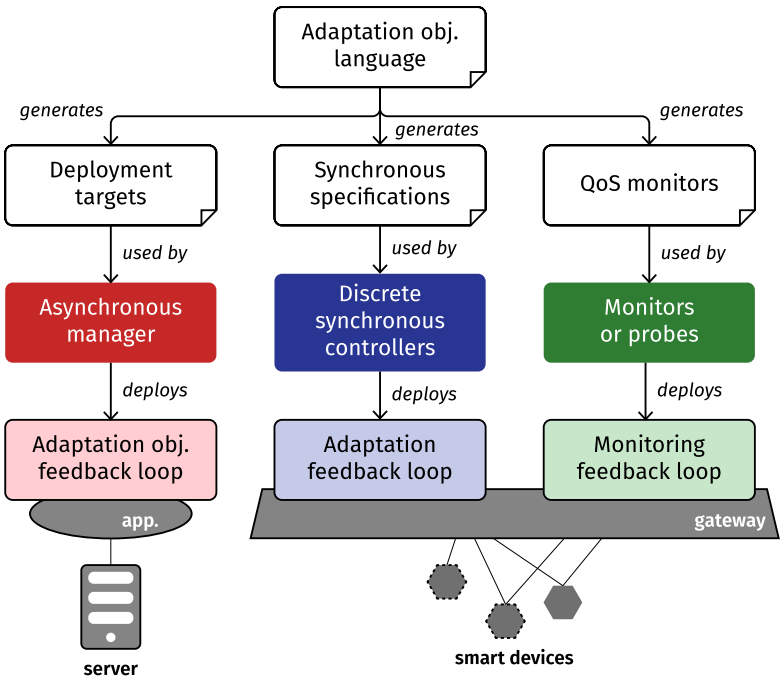


Figure 6.10 – Deployment life cycle

6.4.1 Case Study

Our proof of concept was built around a simple-yet-realistic case study. We suppose our dynamic self-adaptation framework is used for the management of a flock of smart houses coordinated from a single hospital, which follows the case-study extensively described in the previous chapter.

As a reminder, we point out that upon system initialization, we assume that each smart home contains a set of smart devices which are all behaving correctly:

- *Cardiorespiratory smart device*, which corresponds to the sensor described in the previous chapters. Initially, this sensor is in the normal state, and it is battery operated.
- *Photoplethysmography (PPG) smart device*, which measures PPG parameters. This sensor is initially turned off. Even though it is battery operated, we suppose its battery life to be much bigger than the one of the cardiorespiratory sensor, and we thus consider it as line-powered.
- *Gross position and fine position sensors*, which are both line-powered and both initially turned off.

When it comes to the adaptation goals that we deploy to achieve dynamic self-adaptation, we implement the adaptation rules given in Figure 5.9. Namely, such rules are specified for the preservation of satisfactory QoS for any state of the cardiorespiratory sensor, which is the most important smart device of the system as it is the only one directly measuring instantaneous cardiac parameters. Rules should be able to be specified directly by end-users, but also to be dynamically added or removed depending on the status of monitors.

When it comes to the functional aspects of the system, we only consider the remote notification of failed cardiorespiratory sensor. The functional aspects of the prototype was voluntarily less explored than the non-functional aspects, as most work oriented at

self-adaptation of IoT-based systems deal with functional adaptation, such as in [SLR17; Syl+17; Zha+13; Zha+14], and experimental results are introduced in the next section.

6.4.2 Experimental Results

In terms of experimental results, we first display the implemented GUI, from which both medical doctors or systems managers interact. Then, we also performed a quantitative evaluation of the scalability of our system, both with respect to number of rules and number of gateways included in the system.

GUI Presentation

Figure 6.11 introduces the first part of the GUI, and it is a simple login screen where users can chose to operate as normal users or system administrator. Normal users refer to medical practitioners which are only interested in functional and biomedical data from the system, while the system administrators are users who configure the system's self-adaptation behavior. Please note that in the current state of the PoC, no authentication mechanism is implemented, and this represent a direction for future improvements.

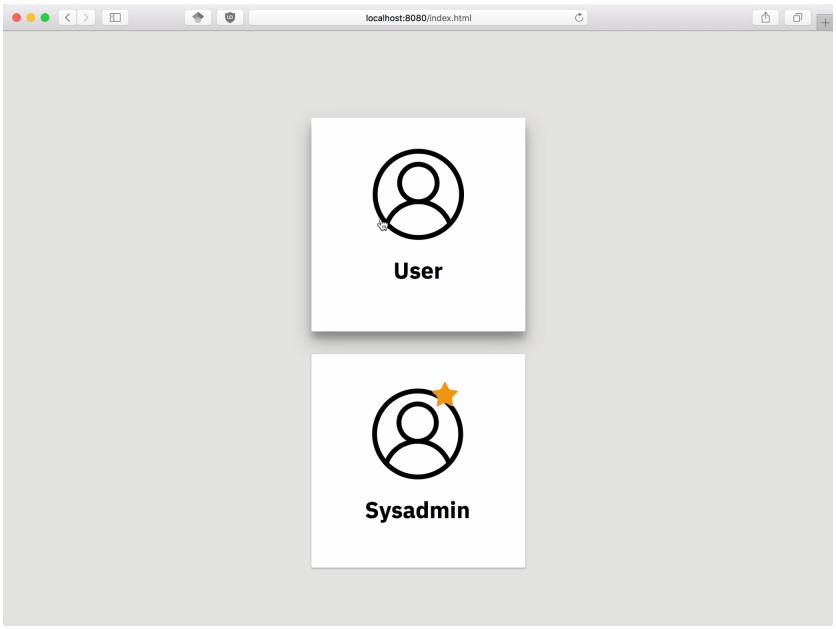


Figure 6.11 – GUI welcome screen

On Figure 6.12 is displayed the global panel of the normal user perspective. In this view, all homes of our IoT system are displayed on a map, associated with a color code depicting the severity of QoS degradation in a specific sub-system, or the presence of a critical health crisis. This information is delivered in real-time using the event-based WebSocket protocol. We do not differentiate between two events as in both cases, external human intervention is necessary in order to solve the problem. Please note that this Web-page is dynamically generated by the templating engine. In particular, homes are placed on the map depending on gateway information stored in the database. On the right-hand side of the view, end-users can access home-specific information

for all the listed homes. This menu is also dynamically generated trough Thymeleaf templating.

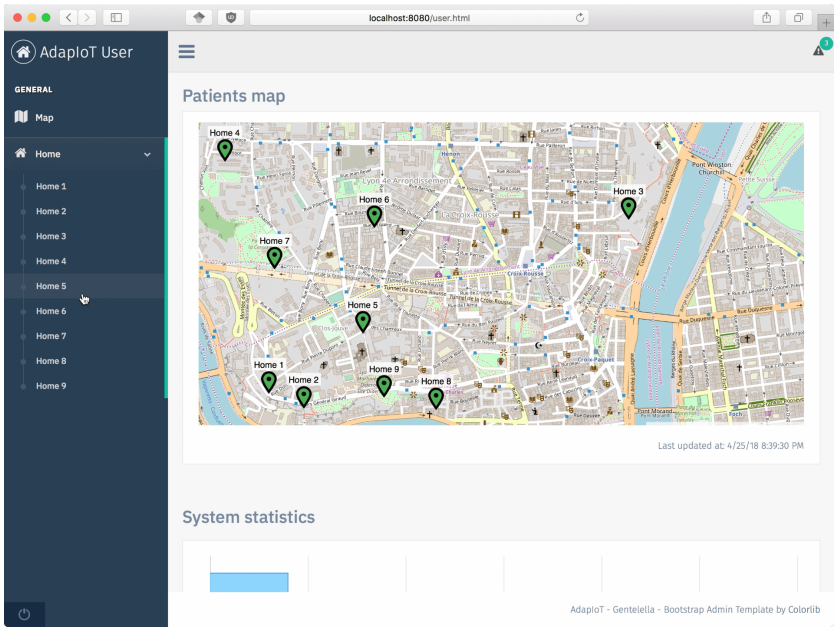


Figure 6.12 – Global view

Figure 6.13 details the home-specific view. In this perspective, end-users have access to patients' data. Since this panel does not introduce any QoS metrics measurements, we consider collected data to be functional. We emphasize on the fact that data in this perspective is displayed in real-time using the WebSocket protocols. This empowers end-users with the capability to continuously and remotely know the health status of monitored patients. In addition, since our interface is Web-based end-users can access this data from any device with Internet access.

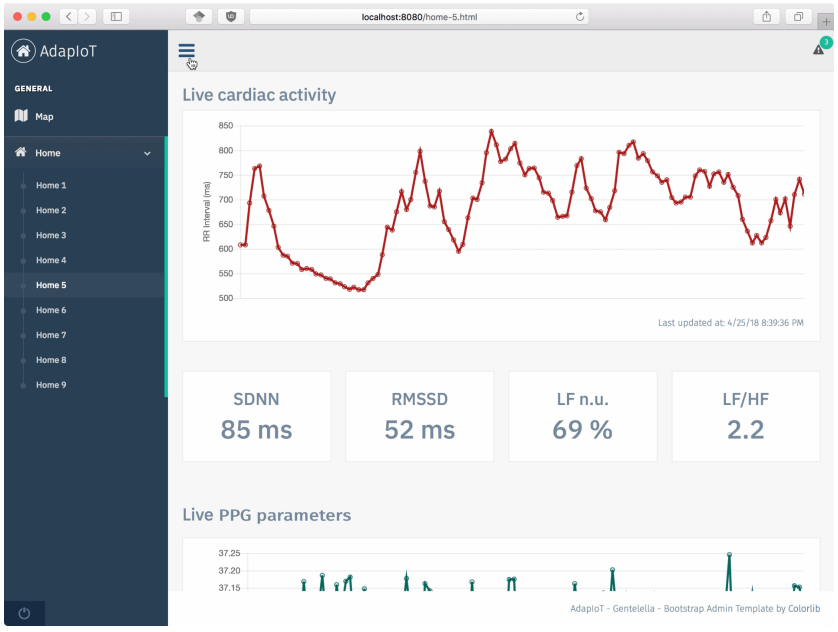


Figure 6.13 – Physiological data view

Eventually, Figure 6.14 illustrates the GUI for adaptation goals management. This perspective can be used by end-users to administrate our dynamic self-adaptation system, and more particularly to specify self-adaptive behavior through our rule-based language. On the left-hand side of this view, a menu is presented listing all the rules deployed in the system along with an option for the addition of a new rule. On the right-hand side, already-deployed rules are displayed, and they can be modified or deleted (which will trigger self-adaptation of the system). Eventually, on the lower part, we implemented a logger in order to display internal information about the system, and it can typically be used to follow the deployment of a new rule in the gateways.

It is worth noting that for our specific use-case, we only define smart devices assigned with the type `Sensor` (following the type notion introduced by our rule-based self-adaptation language). Then, device are further filtered using a filter attribute called deviceType, as illustrated in the *Heart Failsoft* rule of Figure 6.14. This two-fold specification of device types was selected because it facilitates the specification of a JSON-based parser, which is manually implemented for this first version of our PoC. Rules are also making use of the `$` binding operator to select devices corresponding to the correct sub-system when self-adaptation is necessary.

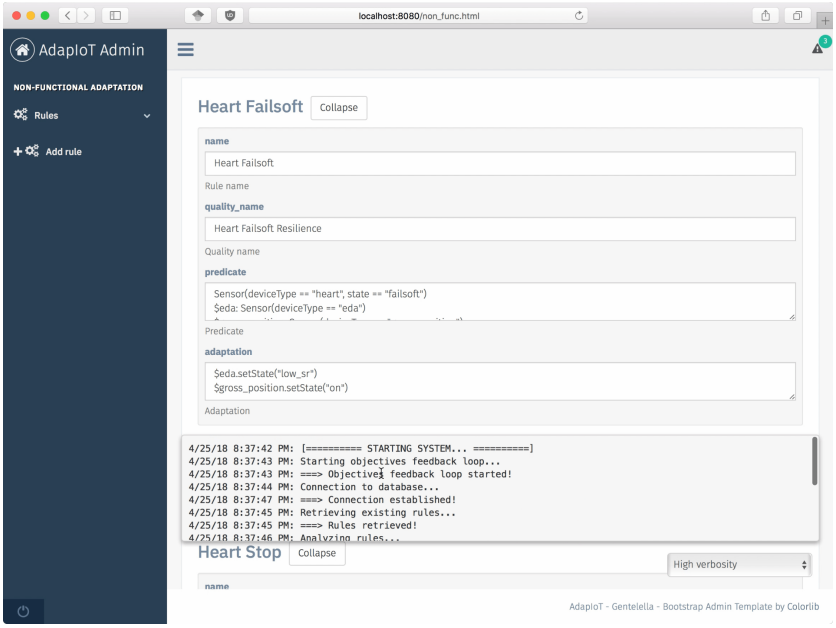


Figure 6.14 – System administrator view

This concludes our presentation of our prototype's Web-based GUI, which comprises two views of the system: a high-level and functional view oriented at the remote monitoring of various parameters such as patients' physiological signal or overall QoS of specific subsystems, and a lower-level view oriented at the specification of non-functional adaptation objectives.

Quantitative Results

Our next evaluation consisted in a quantitative characterization of our prototype. Indeed, we particularly emphasize the importance of scalability for systems targeted at IoT-based applications, as they can contain hundreds or even thousands of devices.

Even though technical solutions used in our PoC were selected with scalability in mind, experimentation regarding this characteristic is still necessary.

Our first experiment was realized to test the system scalability with respect to number of rules. Even though the number of devices, and consequently the number of rules, included to specific subsystems are never high, it is still worth characterizing our system with respect to the number of rules deployed on each sub-system. The reason behind the number of rules being always reasonably low comes from the synchrony hypothesis, which constrains number of devices and rules contained in synchronous sub-systems to be kept low, as a high number of devices and adaptation rules might break this hypothesis. Our experiment was conducted using rules illustrated in Figure 5.9. Practically, we measured time between rules insertion in the system and Java code (i.e., discrete controller) generation, and results are introduced in Figure 6.15.

To improve experimental consistency, we performed 10 measurements, and the plot given in Figure 6.15 displays their average and standard deviation. The experiment was performed on an Ubuntu 18.04 virtual machine assigned with 4 CPU threads, 8,196 MB of RAM, and 32 GB of disk space running on a server equipped with 12 GB of RAM and an Intel Xeon E5 CPU running at 3.7 GHz.

The first observation we can draw from this experiment is that, rather trivially, increasing the number of rules in the system increases deployment time (about 650 ms for a single rule up to more than 1100 ms when 5 rules are being deployed). More interestingly, the behavior depicted in Figure 6.15 is not linear, and deployment times for the addition of 3 to 5 rules are similar. This experiment is bounded to 5 rules because of the complexity of correct rule specification. Indeed, because discrete controller synthesis relies on a formal modeling and analysis of the systems' component, some behavior are not achievable. Practically, this can result in synthesis failure, which prevents the redeployment of a correct discrete controller in the gateways. Adding more rules to the system thus require a particular attention. In conclusion, the first experiment demonstrates acceptable system scalability as the number of rules increases, even though the amount of rules deployed in each sub-system is bound to be small. This observation is of particular interest if computationally powerful gateways are used, since it means they can process a higher number of event thus tackling potential synchrony hypothesis violation. In our case, we however consider cheap and lightweight gateways such as Raspberry Pis, and this constrains the size of the synchronous sub-systems.

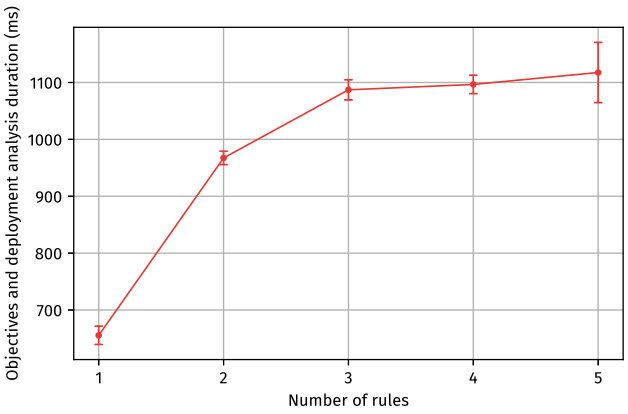


Figure 6.15 – Objectives analysis duration as a function of number of rules

Our second experiment concerns the evaluation of our framework's scalability with respect to the amount of homes (or gateways) being managed. The results are introduced in Figure 6.16. During this experiment, 5 rules were considered for system-wide deployment. Identical experimental setup than the one used in the previous experiment was used, and we also ran 10 consecutive trials as means of improving experimental consistency. In this particular experiment, we only measured the duration between new rules reception and rules deployment to all the relevant entities. Indeed, in the current system state, rules are analysis is performed for each subsystem (meaning each sub-system perform discrete controller synthesis), either in the cloud or directly on gateways. Indeed, since our prototype features remote operation capabilities using the SSH protocol, one can either perform analysis on the cloud and subsequently deploy the executable discrete controller on the gateways, or the gateways can directly subscribe to the non-functional adaptation objectives and perform synthesis and deployment locally. Because scalability experimentation using real-world large-scale infrastructure (i.e., in our case, using up to a hundred of Raspberry Pis) were not possible, we performed analysis in the cloud. As depicted on Figure 6.16, our system exhibits linear behavior, which means our dynamic self-adaptation system can scale up with respect to the number of managed synchronous sub-system, as it not subject to deployment time explosion.

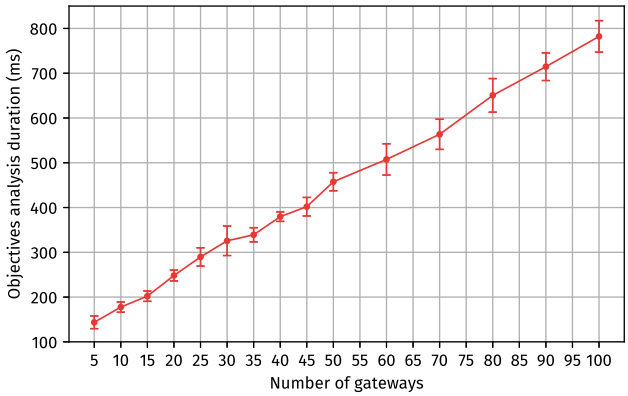


Figure 6.16 – Objectives analysis duration as a function of number of rules

In conclusion, we performed scalability experiment with respect to the number of rules and number of managed synchronous sub-systems. While number of rules for a given sub-systems should never be high as it might compromise the verification of the synchrony hypothesis, the measured behavior points toward good scalability when a moderate (e.g., 5 to 10) number of rules is considered. More interestingly, our system present linear behavior when it comes to the scalability with respect to the number of managed sub-systems, which points toward good scaling-up properties because we are able to avoid adaptation objectives analysis time explosion.

6.5 Conclusion

In this chapter, we describe the prototyping of our dynamic self-adaptation architecture along with a series of preliminary experiments to evaluate its scalability. We first start with the introduction of various external software and libraries used for the development

of our prototype. Such elements were selected because they can address *scalability*, *reliability*, and *interoperability*.

Practically, we chose DDS to provide brokerless publish-subscribe communications in the higher level of the system since it eliminates single points of failure by not relying on a central broker to achieve message distribution. For the asynchronous adaptation objective management and for the Web-server handling the GUI, we used the Vert.x tool-kit. Our motivation for the inclusion of this software stems from its capability to tackle the scalability issue by providing an implementation of the reactor pattern. We chose MongoDB for our database, as we do not need the advanced relationship modeling of SQL-based solution, and because it stores data as semi-structured JSON documents. When it comes to the rule engine, we used Drools principally because it features advanced CEP capabilities which makes it appropriate for IoT-based applications. Our entire framework was implemented using Java and the Maven software management framework in order to maximize compatibility. Our implementation results in a fully distributed hybrid self-adaptation system, which differentiates from other self-adaptive solution by its close interaction with hardware and its focus on non-functional properties.

Our framework's scalability was evaluated with respect to two criteria: the number of rules and the number of managed synchronous sub-systems (i.e., number of gateways). We found our system to be scalable for both criteria, as deployment time explosion did not occur during our experiment. Additionally, we presented the GUI empowering end-users with the capability to easily administrate self-adaptive behavior, but also by providing real-time information about various patients-related or system-related metrics.

One must however realize that our prototype is for the moment only a PoC, and it is consequently faced with some limitations. The principal limitation comes from the fact that our rule-based self-adaptation language is only implemented through a simple parser, which strongly limits features such as advanced code analysis, syntactic coloration, ability to generate binaries, etc. In the future, we plan to fully implement this language using tools such as Eclipse Xtext [8], which is a full-fledged framework for the implementation of Domain-Specific Language (DSL). Another limitation is the static nature of monitors deployment, which are, for the moment, hard coded and only remotely enabled or disabled. A potential research direction for monitors dynamic deployment also comes from the development of an exhaustive compiler for our rule-based language. Indeed, through advanced code analytics, relevant QoS monitors can be inferred and automatically deployed on the relevant targets. Yet another limitation is the lack of security considerations in the development of our prototype, and possible solutions are the use of already-developed security protocols associated with the framework and tools we used (e.g., the utilization of Hypertext Transfer Protocol Secure (HTTPS) rather than HTTP for the interface, the inclusion of access control and cryptographic techniques to the DDS publishers and subscribers, etc.).

8. Eclipse Xtext: `https://www.eclipse.org/Xtext/` (visited on 08/01/2018)

CHAPTER 7

Conclusion and Perspectives

Contents

7.1 Summary of the Contributions 145
7.1.1 Service-Oriented Design Method for Smart Devices 146
7.1.2 Dynamic Self-Adaptation Framework 147
7.2 Future Research Directions 148

7.1 Summary of the Contributions

In sum, this dissertation introduces a vertical approach to build self-adaptable smart devices for IoT-based systems. Indeed, in our state of the art, we demonstrate that traditional IoT-related approaches are typically horizontal, and usually focus either only on devices, communication protocols and/or services. The principal challenge of horizontal approaches is that they sometimes introduce a mismatch between concerns of each layer of the IoT architecture. For instance, high-level service-oriented frameworks often consider smart devices as black-boxes without any insights on internal resources availability, power consumption, or computational capabilities. While such approach are satisfactory for ad hoc visions of the IoT, they lack scalability and represents a strong limitation when broader IoT-based systems are considered. In addition, horizontal solutions often lack accurate representations of multidisciplinary perspectives when designing smart devices.

In our contribution, we consider a bottom-up approach to build IoT solutions and services, starting from the design of service-oriented smart devices and working our way towards devices integration in IoT systems. By taking this research direction, we are able to introduce lower-level concerns of smart devices such as resources limitation and deal with intrinsic properties of IoT-based systems such as self-adaptation capabilities. Indeed, since smart devices are in constant interaction with the physical world, they are required to be reliable and safe when reacting to changes in order to preserve expected behaviors. In addition, we emphasize on self-adaptation as a preliminary step towards embedded intelligence in IoT devices and systems.

Furthermore, we emphasize on the importance of a multidisciplinary approach for IoT solutions design. For instance, our state of the art established the relevance of

Service-Oriented Architectures (SOA) in IoT-related contexts. Indeed, by defining self-contained, flexible, modular and interoperable services, such architectural style provides solutions to some IoT-related challenges. However, SOAs are strongly software-oriented, and do not account for potential hardware concerns such as resources limitation. In this dissertation, we attempt at applying the service-oriented paradigm as low as the smart devices' scale. By doing so, we are not only able to improve smart devices modularity and interoperability, but we also facilitate their integration into higher-level resources-aware frameworks. This leads us to the definition of a service-oriented design method for smart devices.

Pursuing our bottom-up approach, we empower the integration layer of the IoT with self-adaptive capabilities. Namely, we focus on enhancing self-adaptive behavior with dynamic capabilities, particularly in respect to control objectives and monitoring infrastructure. Considering IoT-based systems are dynamic and IoT topology varies over time due to device mobility or device inclusion and exclusion, self-adaptive solutions must be able to appropriately manage such changes. Practically, we achieve full-fledged dynamic self-adaptation through the separation of concerns between control objectives management, actual self-adaptive behavior, and monitoring strategies.

7.1.1 Service-Oriented Design Method for Smart Devices

Our principal contributions with respect to the low layers of the IoT are the proposition of a service-oriented design method for smart devices along with its application to the design of a medical-grade cardiorespiratory sensor. Our design method adopts a V-shaped life cycle, and integrates formal specification and validation steps. Indeed, we emphasize on the importance of formal specification when it comes to the design of smart devices, as it provides necessary tools to ensure expected behavior of the devices being designed. In addition, smart devices present measuring, actuation and configuration as a set of low-level services called *hardware services*, which can be used to access internal information on devices resources. The design method starts with specifications of functional and non-functional needs, which are then used as the basis for the selection of appropriate Integrated Circuits (IC). Once the hardware is selected, developers are required to design a hybrid modular architecture based on hardware-software components, communicating serialized data through serial buses (i.e., using Direct Memory Access (DMA) channels in the ideal case, in order to minimize microcontroller load). Practically, such components can be seen as a hardware/software partition, and several components can run on the same IC. Components are then individually specified and verified through model checking, and these specifications are subsequently used for the verification of components integration. After successful verification, the hardware-software components are implemented and tested according to testing procedures defined with regard to functional and non-functional requirements specified in the early step of our design method. Once testing is passed, smart devices can be mass-produced for wide-scale usage.

In order to test our service-oriented design method, we apply it to the design of a medical-grade cardiorespiratory sensor. We set the functional requirements to computing Heart Rate (HR) and Heart Rate Variability (HRV) parameters from electrocardiogram (ECG) signal and the measurement of the Respiration Waveform (RWF). Non-functional requirements include an extended battery life (superior to 48 hours), wearable nature

(i.e., the smart device must be small-sized and lightweight), and the inclusion of external and internal self-adaptive capabilities. To meet functional requirements, careful hardware selection was performed, which resulted in the use of the PSoC5LP for its combination of an ARM Cortex-M3 microprocessor, a Digital Filter Block (DFB) and a small-scale programmable logic gate matrix, the BLE113 for handling Bluetooth Low Energy (BLE) communications and the ADS1292R for analog signal conditioning and conversions. We then proceed with the extensive testing of our sensor, which demonstrated that expected non-functional requirements were achieved. Additionally, our sensor was equipped with internal and external self-adaptive properties, as we foresee self-adaptation as being an enabler of the emergence of full-fledged intelligent IoT-based systems.

7.1.2 Dynamic Self-Adaptation Framework

Even though self-adaptation of software systems is well-studied, the unique characteristics of smart devices and IoT-based systems brings additional challenges that are not accurately managed by software-only solutions. Namely, such solutions are usually a poor fit for resources-constrained smart devices, as they are typically based on resources-consuming technologies such as ontological analysis, Web-based protocols or full-fledged server-oriented frameworks. Additionally, the dynamic nature of IoT networks and the fact that devices are in constant interaction with the ever-evolving physical world, require IoT-based systems to accurately react to changes. Furthermore, the critical nature of most IoT-based systems calls for the study of Quality of Service (QoS) preservation as a way to ensure consistent behavior in a wide range of operating conditions.

To this end, we introduce our IoT-targeted dynamic self-adaptation framework, which is able to deal with changing control objectives and monitoring probes through separation of concerns and by implementing three independent feedback loops: the objectives feedback loop, monitoring feedback loop and adaptation feedback loop. In addition, we use Discrete Controller Synthesis (DCS) and Synchronous Programming Languages (SPL) to provide safe and correct-by-construction controllers. Indeed, SPLs specify synchronous systems using Labeled Transition Systems (LTS), which provide a formal framework for the modeling of smart devices. Such models are then used in combination with advanced algorithms to generate correct-by-construction imperative code in Java implementing self-adaptive behavior.

Furthermore, we propose a rule-based language for the declarative specifications of self-adaptation strategies. Declarative approaches specify what a system should achieve rather than how it should achieve it. They consequently reduce the complexity of the specification of control objective, which in turns improves scalability by reducing the code base.

We then tackle the scalability issue by integrating asynchronous control objectives management while preserving synchronous adaptation and monitoring. Indeed, the volumes of events generated by large-scale IoT-based systems might cause violation of the synchrony hypothesis if the system running the controller is not powerful enough to compute reactions between two time occurrences of events. Through our hybrid approach, we are able to divide IoT-based system into synchronous sub-systems, which

comprise a small-enough number of devices to ensure synchrony hypothesis verification.

Eventually, the entire dynamic self-adaptation framework was implemented using standard technologies such as the Data Distribution Service (DDS) for communication, Vert.x for the implementation of asynchronous reactive control objectives management and Java as the overall implementation language. We performed preliminary evaluation of our prototype's scalability and results pointing towards good scalability were obtained, since no deployment time explosion was observed.

7.2 Future Research Directions

A first promising research direction is the establishment of a unified data and lifecycle model for smart devices. Indeed, in our dissertation, we use LTSs as models of devices self-adaptive behavior, which can be tied to a lifecycle model. However, data structure is not captured by this approach. A potential solution for representing both data and its evolution over time is to use artifacts. Artifacts, also known as business artifacts, are entities which combine both data and their management into cohesive and modular units [NC03]. They include informational models, consisting of attribute-value pairs, a set of services that manipulates attributes, a set of states describing artifacts evolution over time, and a set of transition rules that invoke services and results in changing artifacts current states. Artifact-based approaches feature many advantages and benefits including natural modularity and componentization of self-contained entities. Such characteristics are particularly relevant to IoT-based systems, are they are commonly strongly data-driven, but must still account for system evolution in order to be able to react to changes in either the execution environment or network topology. Recent works such as [ABA17] further extended the capabilities of artifacts with the integration of stream-based queries, which is relevant for IoT-based applications as most smart devices stream their data to external tiers. Additionally, matches can be established between artifacts and proclets [Hul+11], which are in turn equivalent to Petri nets [Van+01]. Such links can be used as the basis for formal verification of artifact models instances, which is particularly important for IoT-related applications considering the often-critical nature of this category of systems.

As we have discussed in our dissertation, there is a need for verified hybrid hardware and software, especially in the context of the Internet-of-Things, as IoT-controlled systems can be extremely critical. In addition to the critical nature of the IoT-systems, they can also account for hundreds or thousands of devices, making system management complex and time-consuming. From these observations, we identify the need for a verification framework which is simultaneously flexible and simple enough to model and verify any IoT-systems. A promising direction in terms of smart device verification can be found in the linear logic community. The linear logic, introduced by J.-Y. Girard in the late 1980s [Gir87], is capable of modeling resources and their consumption. It is thus of particular interest in IoT-related contexts as it offers a way to precisely model smart devices limited resources along with their evolution over time. Applications of linear logic targeted at automated services composition can be found in the literature, such as in [PFW12] or [RS04]. However, such approaches only focuses on software-centric solutions and only present examples with simple resources, such as the price of a service

invocation. Additionally, these two contributions rely on outdated automated or interactive provers. Interactive provers designate software tools that assist end users with drafting correct proofs. Users are still required to manually specify each step of the proof, and the prover warns users if a specific proof step is not valid. While this can be useful for reasonably-sized proofs, it becomes inapplicable in cases where hundreds of services or devices must be coordinated and verified. Such limitations can be tackled using automated theorem provers, but, to the best of our knowledge, no provers are available (the only exception being llprover [1], but it is strongly limited and cannot be used for the construction of complex proofs). Advanced automated provers are however being developed in the LLipIdO project [2], and results of such project could be used for smart devices verification. Different potential use of the linear logic is in the higher-levels of the IoT, and for the specification of smart services. Through proof automation, linear logic could indeed be used for the implementation of reliable and safe resource-aware smart services relying on the coordination of numerous smart devices.

Smart services could also benefit from the use of advanced semantic Web techniques to facilitate cross-device communication and interoperability. Indeed, semantic Web relies on machine-readable and -understandable data for easier data integration between a variety of actors. Practically, data is annotated in order to establish machine-understandable connections between various concepts (e.g., *Paris* can be annotated with the concept *city*). It is usually used in coordination with ontologies establishing definitions, relationships and properties between knowledge entities. Annotated data can be used in coordination with the relevant ontologies as a basis for advanced reasoning, which can be used to infer new knowledge or relationships between apparently unrelated data. When it comes to the IoT, such tools are of particular interest as they provide solutions to extreme devices and data heterogeneity (e.g., two temperature sensors might stream temperature values in Fahrenheit and Celsius, and annotation can be used to implement automatic conversion without the need for human intervention). The coordinated use of semantic Web techniques and ontologies has already been used in IoT-related contexts, such as in [Ter+17] where authors propose self-adaptation of IP-connected and resources-unconstrained devices using a dedicated reasoner, or in [Ala+15] where authors improve interoperability of their framework based on an ontological description of the system. In addition, the recent standardization of the Semantic Sensor Network (SSN) ontology [3] pushes toward its adoption in a wide range of semantic-oriented application for the IoT. However, semantic Web software often rely on verbose knowledge description using the Web Ontology Language (OWL) and Resource Description Framework (RDF) data models, which limit their use in resources-constrained devices. More recently, initiatives such as JavaScript Object Notation (JSON) for Linked Data (JSON-LD) facilitate the annotation of JSON documents with relationship or contextual information, which easily translates to traditional RDF descriptions used by most semantic reasoners. JSON-LD is of particular interest in our application, as sensor model are stored as JSON semi-structured data. In combination with tools such as JSON Schema, which implements comprehensive specification of expected JSON

1. Linear logic automated prover llprover: `http://bach.istc.kobe-u.ac.jp/llprover/` (visited on 08/02/2018)

2. LLipIdO prover project: `http://perso.ens-lyon.fr/olivier.laurent/llipido/` (visited on 08/02/2018)

3. SSN ontology: `http://w3c.github.io/sdw/ssn/` (08/02/2018)

data structure, this can be the basis of advanced high-level reasoning, thus increasing global system intelligence and being another step towards the implementation of true smart services.

Another potential research direction is the extension of self-adaptation to the lower layers of the IoT. Indeed, in this dissertation, we only consider self-adaptation through the remote coordination of smart devices, and self-adaptive behavior is consequently implemented with respect to the coordination of devices. However, this approach does not include the capability to change smart-devices adaptive behavior per se. Indeed, self-adaptation is implemented through SPLs, which model smart devices as LTSs that are subsequently used for discrete controller synthesis. In our specific application, we use only code generation capabilities for the implementation of the discrete controller, and devices' LTSs are implemented manually. Nevertheless, Heptagon/BZR is also able to generate code for each node comprised in the system, meaning it generates correct-by-construction LTSs for each devices. Directly using this automatically-generated code would benefit smart devices from adding a degree of certitude with respect to devices' behavior.

Another interesting research direction is to investigate discrete controller synthesis at the device hardware scale, meaning smart devices are specified as synchronous modules and their firmware consist of the coordination of these modules, and this approach also provides guarantees on final behavior. The practicality of this solution has been presented in works such as [BMM13] or [GG10]. However, all the methods detailed herein above imply statically implemented LTSs, which limits self-adaptive capabilities. This static implementation also implies that the self-adaptation must be considered a priori, which might be limiting if system designers want to manage new and unforeseen use-cases. This challenge points towards the need for remote firmware flashing through Over-the-Air (OTA) updates. The integration of such mechanism with our dynamic self-adaptation framework and rule-based adaptation goal specification language represent an interesting research directions, and simultaneously brings both research and technical challenges such as the heterogeneity between OTA-updatable and non-updatable devices, the inclusion of low-level concerns in the adaptation languages, etc.

A final noteworthy research direction can be found in IoT cybersecurity. Indeed, the IoT brings many challenges making traditional security obsolete, such as number of devices, limited resources, highly distributed systems, etc. In addition, the trustless essence of the IoT infrastructure requires distributed security solutions, which do not rely on certification from central authorities. The heterogeneity of devices, services and people coming into play in large scale IoT-based system calls for advanced identity management frameworks in order to enable trust relations as means of implementing collaboration between entities to achieve business-specific objectives. Recently, many emerging technologies such as the blockchain can be used to tackle these challenges, and authors of [Zhu+17] propose a blockchain-based identity management system, which provides unique identities to smart devices and link them to their owners' identity. Another application of the blockchain can be found in the securing of smart devices access control as described in [OAA16], which solves potential privacy concerns by providing end-users the capability to specify their own access control policies.

ANNEXE A

Résumé Long en Français

Des Services Intelligents à Partir d'Objets Connectés Réutilisables et Adaptables

Applications aux Réseaux Non-Intrusifs de Capteurs Biomédicaux Portables

Table des Matières

A.1 Introduction 151
A.2 Méthode de Conception Orientée Service pour Objets Intelligents . . . 156
A.2.1 Description Générale 156
A.2.2 Application de la Méthode à un Capteur Cardiorespiratoire . . . 160
A.3 Infrastructure d'Auto-Adaptation Dédiée aux Objets Intelligents 162
A.3.1 Présentation de l'Infrastructure Hybride 164
A.3.2 Implémentation de l'Infrastructure d'Auto-Adaptation 166
A.4 Conclusion 168

B Cardiorespiratory Sensor Schematic 171

A.1 Introduction

L'Internet des Objets (IdO) désigne l'interconnexion d'une multitude d'objets physiques ainsi que leur intégration à des infrastructures à grande échelle afin de répondre à des besoins complexes. Ce domaine est en pleine expansion, et des analystes estiment que 20 milliards d'objets intelligents seront en circulation d'ici 2020 [Cer+15], et que l'impact économique de ces objets sera de 2,9 mille milliards de dollars d'ici la même date. En plus de cet impact économique, l'internet des objets va également causer de nombreux changements sociaux. En effet, son utilisation dans des domaines tels que la santé [BXA17], la régulation de trafic [CZ16] où la ville intelligente [Zan+14] laisse entrevoir des révolutions en termes de réponse à des problèmes complexes ou l'interaction entre les mondes physique et numérique est nécessaire.

Bien que chaque application utilisant l'internet des objets soit différente des autres, des exigences en termes de *fiabilité*, *sureté* et *sécurité* sont à la racine de chaque système basé sur l'internet des objets :

— *Fiabilitié* : Un système est considéré comme fiable si les exigences fonctionnelles sont vérifiées pendant tout le cycle d'exécution de celui-ci. Plus particulièrement, cela implique que dans des cas d'utilisation normaux, aucun dysfonctionnement ne doit survenir. Ce besoin est amplifié par la nature critique de la plupart des systèmes basés sur l'internet des objets (p. ex. les systèmes de santé ou de gestion de trafic), dans lesquels un dysfonctionnement pourrait causer des accidents aux conséquences désastreuses.

— *Sureté* : Celle-ci est définie comme la capacité d'un système à ne pas causer de dommage à son environnement. Étant donné que l'internet des objets est en constante interaction avec le monde physique, cette propriété revêt une importance particulière considérant les dommages potentiellement coûteux, aussi bien économiquement qu'humainement, qu'un malfonctionnement des systèmes pourrait avoir (p. ex. un système de gestion de trafic pourrait causer de graves accident de la circulation si sa sureté n'était pas garantie).

— *Sécurité* : Un système sécurisé est robuste aux attaques malicieuses internes et externes. La sécurité peut être définie comme une propriété fonctionnelle du système (*sécurité par conception*), ou bien être implémentée en détectant en temps réel diverses menaces et attaques (dans ce qui s'appelle la *sécurité à l'exécution*).

En général, l'internet des objets repose sur l'intégration et la coordination de nombreux objets intelligents pour concevoir des systèmes à large échelle répondant à une variété de besoins pratiques. Les objets intelligents sont définis comme des objets (c.-à-d. appartenant de facto au monde physique) équipés de capacités de calcul, de communication et plus largement de comportements intelligents. La multiplication rapide de la disponibilité de cette catégorie d'objets est causée par de nombreux progrès dans des domaines variés tels que les microcontrôleurs, les protocoles de communication sans fil et basse consommation, la gestion d'énergie, etc. Ces techniques ont permis de faciliter la conception d'objets intelligents, qui peuvent maintenant être aisément intégrés à des infrastructures massives en utilisant des protocoles de communication pair-à-pair ou utilisés dans l'internet.

Afin de clarifier les différents éléments entrant en jeux dans la création de tels systèmes, nous adoptions une vision par couches de l'internet des objets, comme illustré dans la FIGURE A.1. La couche de plus bas niveau, désignée comme la « couche objets intelligents », traite de l'étude et de la conception des dits objets. Au niveau intermédiaire se trouve la « couche d'intégration », qui se concentre sur la communication entre les objets intelligents et des logiciels de haut niveau. Finalement, la « couche applicative » a pour sujet l'interfaçage de l'internet des objets à des technologies plus traditionnelles couramment en utilisation dans les entreprises.

Bien que l'internet des objets soit le sujet de nombreuses études, celui-ci présente défis à la fois en termes de recherche mais également de technologies, tels que :

— *Les caractéristiques non-conventionnelles des objets intelligents*, qui peuvent être résumées par :

  — *Capacités limitées* : Bien que l'étendue de ce qui peut être considéré comme un objet intelligent est vaste, les ressources à disposition des objets intel-

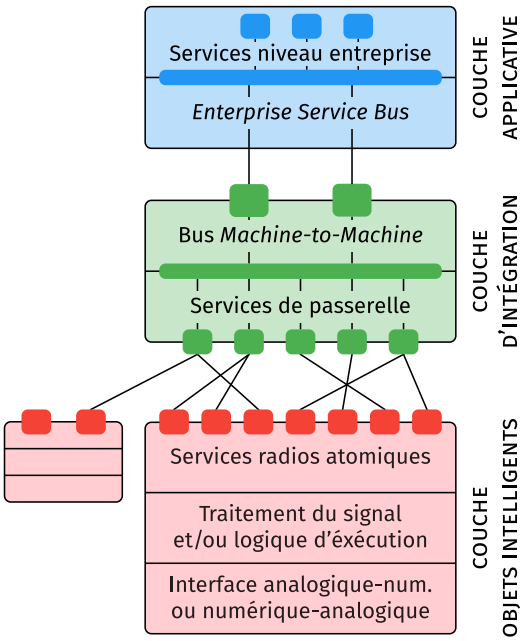


FIGURE A.1 – Représentation par couches de l'internet des objets

ligents sont habituellement faibles, aussi bien en termes de puissance de calcul que de stockage et de mémoire vive. En effet, les microcontrôleurs communément utilisés dans les objets connectés fonctionnent seulement avec une vitesse d'horloge variant de quelques mégahertz à quelques centaines de mégahertz, et embarquent une mémoire de stockage ainsi qu'une mémoire vive de l'ordre de quelques dizaines de kilooctets à quelques mégaoctets. Ces ressources limitées se retrouvent également pour la capacité des batteries, qui dépasse rarement les quelques centaines de milliampère-heures. Ces faibles aptitudes ont de nombreuses implications en matière de protocoles de communication ou de capacités à implémenter des algorithmes complexes. En effet, les protocoles de communication traditionnellement utilisés dans l'internet des objets reposent en général sur la pile TCP/IP, et celle-ci introduit des opérations de calcul additionnelles, ce qui résulte en un besoin en performance et donc une consommation énergétique plus importante. Si de fortes contraintes en termes de consommation sont présentes au sein d'un objet intelligent particulier, les concepteurs de celui-ci devront probablement se tourner vers des protocoles plus économes et n'utilisant pas la pile TCP/IP.

— *Hétérogénéité matérielle* : En plus des ressources limitées des objets intelligents, l'internet des objets doit également faire face à l'extrême hétérogénéité du matériel traditionnellement utilisé pour implémenter ces objets. En effet, de nombreux constructeurs proposent une multitude de circuits intégrés implémentant diverses fonctions. Par exemple, les microcontrôleurs utilisent généralement des architectures 8 bit (p. ex. Atmel AVR ou Intel 8051), 16 bit (p. ex. famille MSP430 de Texas Instruments) ou 32 bit (p. ex. famille Cortex-M d'ARM), ce qui implique des jeux d'instruction différents et des capacités de calcul variées. En plus de l'hétérogénéité des microcon-

trôleurs viennent s'ajouter tous les périphériques disponibles sous forme de circuit intégrés, tels que des convertisseurs analogique-numérique et numérique-analogique, des processeurs de traitement du signal, des *front-end* analogiques, etc. Cette grande variété de composants communique au travers de différents bus et en utilisant des formats de donnée hétérogènes, ce qui vient ajouter une difficulté supplémentaire au développement d'objets intelligents en empêchant l'utilisation d'outils d'aide au design ou au choix de composant.

- *Méthodes de conception hybrides matériel-logiciel* : L'hétérogénéité du matériel entrant dans la création d'objets intelligents nécessite l'étude et la proposition de méthodes de design permettant de traiter à la fois de la variété du matériel ainsi que de permettre d'établir des garanties en matière de fiabilité, sureté et sécurité. Les méthodes de conception sont généralement définies comme une série d'étapes organisées décrivant le cycle de vie de la conception d'un système [WR97], et elles incluent en général des outils pour la validation et la vérification des systèmes en train d'être développés. Néanmoins, ces méthodes doivent être pratiques et faciles à utiliser, en particulier pour les objets intelligents, étant donné les fortes contraintes sur les temps de développement afin de proposer une mise sur le marché la plus rapide possible. Un défi additionnel pour les méthodes de conception appliquées au design de tels objets et le fait que les technologies utilisées durant leur création change rapidement, ce qui motive l'étude de méthodes suffisamment génériques pouvant être appliquées à une variété de matériels et de logiciels.

- *L'hétérogénéité globale de l'internet des objets* : En plus de l'hétérogénéité matérielle exposée précédemment, l'internet des objets doit également faire face à une prolifération d'infrastructures, de protocoles et de standards. En effet, de nombreux protocoles de communication plus ou moins haut niveau sont utilisés dans notre contexte d'étude, et ils peuvent être catégorisés vis-à-vis de leur implémentation de la pile TCP/IP. En effet, celle-ci n'est pas implémentée par certains protocoles comme le *Bluetooth Low Energy (BLE)* ou le Zigbee, qui sont basées sur une pile spécifique. Le principal avantage de ces protocoles repose sur leurs faibles consommations d'énergie [Dem+13], qui est en général beaucoup moins importante que les protocoles basés sur la pile TCP/IP classique. Ceux-ci sont également nombreux, et on peut citer *Message Queuing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP)* ou *Hypertext Transfer Protocol (HTTP)* comme quelques exemples de protocoles communément utilisés dans l'internet des objets. La principale différence entre ceux-ci est la façon d'accédé aux données : dans MQTT, les données sont publiées sur des *topics* en suivant le paradigme publication-abonnement, alors que CoAP et HTTP sont de type *RESTful*. Dans les protocoles de type RESTful, l'accès aux données est synchrone et se fait au travers d'*Uniform Resource Identifier (URI)* [PZL08], ce qui implique des architectures de systèmes de type client-serveur. Cette différence en termes d'accès aux données combinée à l'existence de protocoles non-IP est un facteur d'hétérogénéité, et des passerelles doivent être implémentées afin d'assurer leur inter-compatibilité.

— *La pluridisciplinarité de l'internet des objets* : En effet, les problématiques de recherche apportées par celui-ci sont une combinaison de défis de bas niveau et de haut niveau. Par exemple, la conception et l'implémentation d'objets intelligents sont étudiées par des ingénieurs et chercheurs dans le domaine du génie électrique, du traitement du signal, et même du génie mécanique (si des interactions physiques complexes sont nécessaires). Les couches les plus hautes, comme celles traitant de l'interconnexion des objets intelligents mais aussi de leur intégration à des services intelligents de haut niveau, sont traditionnellement étudiées par les informaticiens. De nombreuses autres spécialités peuvent être impliquées dans l'étude de l'internet des objets, et quelques exemples sont l'analyse de données de masse, l'ergonomie ou encore la sociologie. Le principal défi causé par la forte pluridisciplinarité de l'internet des objets se rapporte souvent au fait que les spécialistes d'un domaine spécifique considèrent exclusivement les problématiques se rapportant à celui-ci. Cette stratégie, bien qu'efficace pour résoudre des problématiques très précises, ne permet pas l'étude de l'internet des objets à une échelle plus globale. C'est pour cela que nous mettons en avant l'importance d'adopter une vision pluridisciplinaire de celui-ci. En effet, la meilleure compréhension des besoins et du vocabulaire de chaque discipline qu'apportent ce type de vision, permet la création de systèmes basés sur l'internet des objets plus cohérents, aussi bien à l'échelle locale qu'à l'échelle globale.

Ces trois défis de l'internet des objets nous ont amené à considérer la problématique de recherche suivante : *comment concevoir des objets intelligents* auto-adaptables *et* réutilisables *afin de faciliter leur intégration avec des services intelligents tout en représentant leurs caractéristiques uniques ?* Cette question permet de regrouper les défis de l'internet des objets au travers des notions de réutilisabilité et d'auto-adaptation. En effet, le premier permet l'étude des problématiques d'interopérabilité entre objets intelligents, mais il illustre également le besoin de meilleures méthodes de conception afin d'intégrer ces notions très en amont du cycle de vie de l'objet (c.-à-d. dès son design). L'auto-adaptation permet d'augmenter l'intelligence des objets en leur permettant de réagir aux contextes internes et externes, et peut donc être perçue comme une première étape vers l'établissement de systèmes intelligents à toutes les échelles, de l'objet aux services haut niveaux. Plus particulièrement, nous détaillons deux axes de recherche :

— *L'amélioration de la conception des objets intelligents*, qui amende également leur intégration à des services intelligents. En effet, en considérant l'interopérabilité dès les phases de design, nous sommes capables d'améliorer l'interopérabilité logicielle et matérielle de tels objets. Celle-ci se fait au travers de l'utilisation d'architectures orientées services dès les niveaux les plus bas de l'internet des objets, tout en préservant la possibilité d'exprimer les contraintes en termes de ressources des objets intelligents.

— *L'auto-adaptation pour les objets intelligents*, qui peut être perçue comme un moyen d'améliorer la fiabilité et la sureté des systèmes de l'internet des objets. En effet, le fait pour les objets intelligents de pouvoir s'adapter à diverses conditions de fonctionnement les rend plus robuste aux changements constants du monde physique dans lequel ils évoluent, ce qui rend les systèmes basés sur de tels objets plus fiables et plus sûrs.

Dans le reste de ce résumé, nous allons détailler les contributions et résultats clefs des recherches réalisées au cours de cette thèse. La section A.2 détaille notre méthode de

conception orientée service pour objets intelligents ainsi que son application à un capteur cardiorespiratoire médical. Ensuite, la section A.3 décrit notre infrastructure pour l'auto-adaptation des objets connectés, ainsi que son implémentation et évaluation. Finalement, nous concluons ce résumé par la section A.4 en identifiant des directions de recherche potentielles qui pourraient faire suite à ces travaux.

A.2 Méthode de Conception Orientée Service pour Objets Intelligents

Bien que de nombreuses méthodes de conception soient communément décrites dans la littérature scientifique, elles présentent pour la plupart des inconvénients lorsqu'appliquées au design d'objets intelligents. En effet, les méthodes de conception de haut niveau et orientées service, telles que SOMA [Ars+08] ou SOAD [ZKG04], ou bien celles décrites dans [PH06] et [CK07], sont fortement orientées vers le design de solutions purement logicielles, et ne permettent pas de capturer la nature hybride des objets intelligents utilisés dans l'internet des objets. Des méthodes plus bas niveau, dédiées à la conception de systèmes embarqués, traitent du design et de l'implémentation de ceux-ci. Celles-ci sont nombreuses, et de nombreux exemples peuvent être cités, comme [dNOW07 ; Mhe+14] qui traitent de la conception de systèmes électromécaniques, ou [Kei+09 ; Döm+08] qui mettent l'accent sur la synthèse automatique de systèmes hybrides basée sur la co-conception matérielle-logicielle [Wol94 ; Wol03]. Celle-ci est définie comme l'étude simultanée du logiciel et du matériel des systèmes embarqués afin de décider de la forme de l'implémentation des fonctions du système afin d'obtenir des performances optimales. Cependant, ces méthodes de co-conception matérielle-logicielle reposent souvent sur l'hypothèse qu'il est possible de synthétiser le matériel du système considéré. Bien que cela soit vrai pour le design de systèmes-sur-puce, ce n'est pas le cas pour la conception d'objets intelligents, qui doit être perçue comme l'intégration matérielle de plusieurs circuits intégrés plutôt que comme la fabrication d'un de ces composants lui-même.

Par conséquent, il y a donc un manque en termes de méthode de conception qui ne soit à la fois pas exclusivement orientée vers le logiciel, ni trop bas niveau comme le sont les méthodes de co-conception matérielle-logicielle. De plus, afin de répondre à la problématique de la réutilisation des objets connectés, nous souhaitons conserver une dimension orientée service. En effet, en définissant des modules autonomes, interopérables, réutilisables et modulaires sous le terme de *services* les chercheurs en informatique ont défini les architectures orientées services. Ces architectures sont utilisées dans l'implémentation de systèmes complexes, alors définis comme une coordination de services répondant aux besoins d'interopérabilité et de flexibilité. De plus, les services sont souvent définis comme indépendants de leur technologie d'implémentation, ce qui les rend particulièrement pertinents dans un contexte relatif à l'internet des objets considérant la grande hétérogénéité matérielle et logicielle des objets intelligents le constituant.

A.2.1 Description Générale

Notre méthode de conception orientée service pour objets connectés est illustrée FIGURE A.2. Celle-ci adopte un cycle de vie en forme de V, ce qui permet de mettre l'accent

sur l'importance des tests dès le début du processus de conception en spécifiant des procédures de tests à diverses échelles (tests unitaires, d'intégration ou d'acceptation). Notre méthode est divisée en trois parties majeures (illustrées en bleu, rouge et vert) et 9 étapes (numérotées de 0 à 8). Les trois parties majeures traitent de l'analyse des besoins et contraintes de l'objet intelligent en cours de conception ainsi que de la définition d'une architecture hybride modulaire (en bleu) ; de la spécification et vérification formelle à la fois des modules hybrides ainsi que de leur intégration (en rouge) ; et enfin de l'implémentation et des tests de l'objet connecté (en vert).

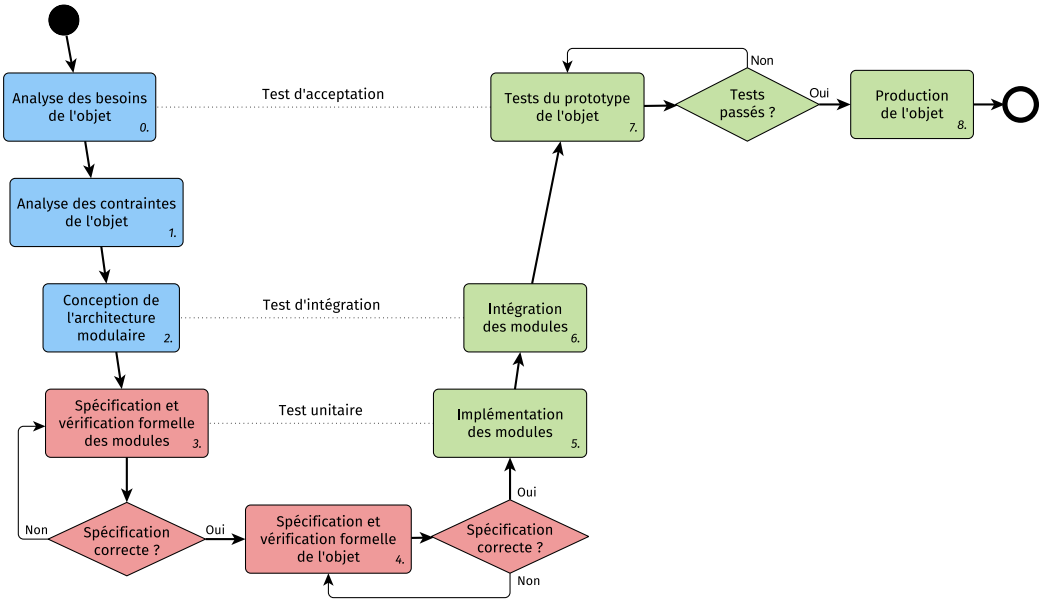


FIGURE A.2 – Cycle de vie de la méthode de conception orientée service

Plus particulièrement, nous définissons neuf étapes :

0. *Définition et analyse des besoins de l'objet intelligent* : Dans cette étape, les concepteurs spécifient (eux-mêmes ou au travers de la discussion avec des clients potentiels) les besoins fonctionnels de l'objet. Ces besoins peuvent être spécifiés au travers d'outils d'analyse fonctionnels, et se concentrent exclusivement sur les propriétés fonctionnelles de l'objet. Cette analyse répond aux questions suivantes : *qu'est-ce que* fait l'objet intelligent ? *Comment* le fait-il ? *À qui* celui-ci s'adresse-t-il ? Les réponses à ces questions peuvent ensuite être utilisés afin de sélectionner les composants électroniques permettant de réaliser les fonctions de l'objet intelligent.
1. *Définition et analyse des contraintes de l'objet intelligent* : Une fois les besoins fonctionnels établis, les besoins non-fonctionnels de l'objet doivent être déterminés. Ceci est réalisé au travers de la définition et l'analyse des contraintes associées à l'objet en cours de conception, et nous les définissons en termes de *taille, capacités de calcul, autonomie, connectivité* et enfin *capacité de stockage*. Ces critères vont être utilisés pour filtrer les composants sélectionnés lors de l'étape précédente afin de répondre aux besoins non-fonctionnels. En effet, des relations de corrélation ou d'anticorrélation existent entre ces paramètres (p. ex. un augmentation de la taille de l'objet intelligent entraine une augmentation de l'autonomie car les concepteurs peuvent utiliser des batteries plus grandes et

de plus forte capacité), et des compromis doivent être établis afin d'obtenir une solution optimale. Au terme de cette étape, ls concepteurs établissent donc une liste des composants qu'ils utiliseront pour la réalisation de l'objet intelligent.

2. *Conception de l'architecture modulaire hybride* : Dans cette étape, les concepteurs spécifient une architecture modulaire basée sur les composants sélectionnés, et un exemple d'une telle architecture est donné FIGURE A.3. L'idée derrière cette étape de conception est d'établir une architecture hybride basée sur des modules matériels-logiciels réalisant une fonctionnalité unique. Ce découpage de l'objet connecté permet d'améliorer l'interopérabilité en maximisant les possibilités d'utilisations ultérieures des modules déjà implémentés. En plus, celui-ci se rapproche de la notion de service des architectures orientées service, qui sont utilisées pour leur flexibilité et leur modularité.
3. *Spécification et vérification des modules* : Comme mentionné dans l'introduction, obtenir des garanties en termes de sureté et de fiabilité est capital pour les systèmes basés sur l'internet des objets étant donné leur nature généralement critique. Pour améliorer ces garanties, nous appuyons l'importance de l'utilisation de méthodes formelles, qui permettent de tester des propriétés diverses des systèmes étudiés à partir d'un modèle mathématique de ceux-ci. Nous avons par conséquent choisi d'intégrer une étape de spécification et de vérification des modules définis dans l'architecture modulaire de l'étape précédente. Nous n'imposons cependant pas de langage de spécification formel en particulier, et laissons à la discrétion des concepteurs le choix de cet outil parmi la variété de logiciels disponibles, comme TLA+ [Lam93], UPPAAL [LPY97] ou KeYmaera X [Pla18].
4. *Spécification et vérification de l'objet* : Une fois les modules spécifiés et validés, l'objet intelligent lui même doit être sujet au même procédé. En effet, celui-ci peut être vu comme l'intégration des modules hybrides, et la vérification de cette intégration permet d'ajouter un degré de certitude supplémentaire vis-à-vis du comportement final de l'objet intelligent.
5. *Implémentation des modules* : Une fois l'objet complètement spécifié et vérifié, celui-ci peut être réalisé en suivant la spécification, et en adoptant par conséquent une approche module par module. Nous insistons sur l'importance du fait que les concepteurs respectent au mieux la spécification formelle définie dans les étapes précédentes. En effet, c'est sur celle-ci que reposent les garanties en termes de sureté et de fiabilité de l'objet connecté en cours de conception, et s'écarter de la spécification annule ces garanties et peut introduire diverses failles ou erreurs.
6. *Intégration des modules* : Une fois tous les modules implémentés, leur intégration peut être réalisée. À l'issue de celle-ci, un prototype de l'objet intelligent est produit.
7. *Tests du prototype de l'objet* : Le prototype produit à l'étape précédente est utilisé pour réaliser une variété de tests afin de vérifier que l'objet est conforme aux exigences initiales. En effet, bien que les étapes de spécification et vérification formelles permettent d'examiner certaines propriétés de l'objet en cours de conception (comme la *liveness* ou la sureté de celui-ci), toutes ne peuvent pas être l'objet de cet examen. En effet, des propriétés telles que l'autonomie restent difficiles à vérifier a priori, et nécessitent souvent des mesures expérimentales afin d'être validées. De plus, de possibles erreurs lors de l'implémentation peuvent être présentes. La somme de ces constatations nous motive à insister particulièrement

sur la nécessité de procédures de tests avancées, permettant à la fois d'ajouter des garanties vis-à-vis du comportement final de l'objet, mais également de vérifier sa pertinence dans une variété de conditions réelles. Les erreurs détectées par le biais des tests peuvent ensuite être, dans une certaine mesure, corrigées avant la production finale et en masse de l'objet intelligent.

8. *Production de l'objet* : Cette dernière étape de notre méthodologie décrit la production en masse de l'objet intelligent développé. Ainsi, une fois les procédures de tests passées, les concepteurs peuvent faire appel à une variété de prestataires pour la fabrication des circuits imprimés de cet objet, l'encapsulation ou bien la distribution de celui-ci.

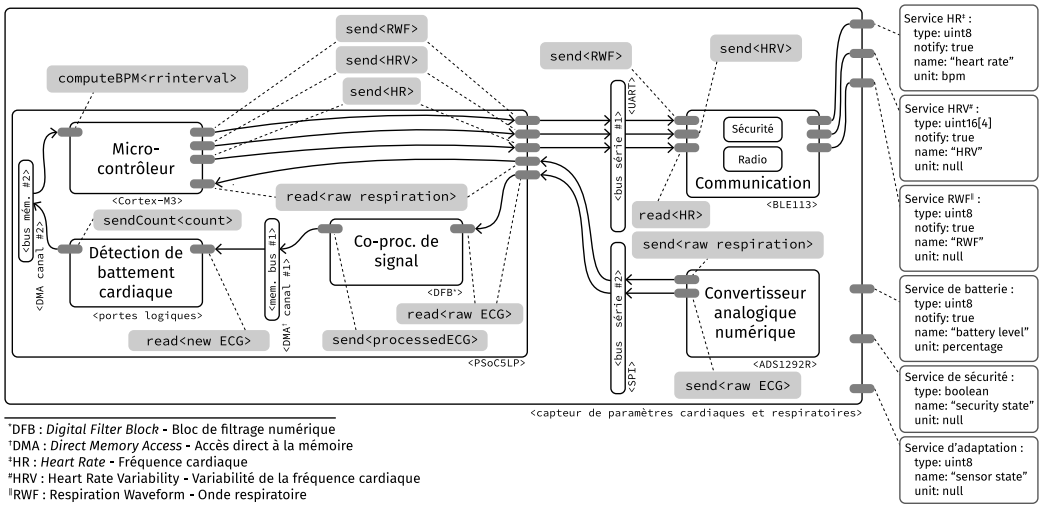


FIGURE A.3 – Architecture modulaire hybride du capteur cardiorespiratoire

En plus de ces étapes de conception détaillées, nous définissons également une liste de recommandations permettant la définition des modules matériels-logiciels autonomes exposant des interfaces bien définies :

— *Découpler les modules logiciels et matériels* : Cela permet de réduire la charge de calcul du microcontrôleur en implémentant les différentes fonctions efficacement sur des composants dédiés.

— *Utiliser des accès mémoire directs et des interruptions pour la communication inter-modules* : Ces deux éléments permettent également une réduction de la charge de calcul, étant donné que les données transitent directement par les bus appropriés sans intervention du microcontrôleur et que la détection de nouvelles données se fait par interruption plutôt que par attente active.

— *Annoter les échanges de données* : En annotant les échanges entre les modules hybrides, ceux-ci deviennent auto-descriptifs et améliorent l'autonomie des modules, qui ne reposent plus sur une spécification haut niveau des formats de données. Pour des raisons de compacité, des protocoles de sérialisation binaires tels que MessagePack, CBOR ou BSON peuvent être utilisés.

— *Spécifier des services matériels dans le composant radio* : Ces services exposent les fonctionnalités bas niveau des objets intelligents au monde externe, et sont définis vis-à-vis des catégories *actionnement, mesure* et *configuration.*

— *Activer la sécurité dans les composants* : Ce dernier point permet d'améliorer la sécurité globale du système en activant des accélérateurs de chiffrage matériels présents dans la plupart des composants actuels.

A.2.2 Application de la Méthode à un Capteur Cardiorespiratoire

Afin de confirmer l'aspect pratique de notre méthodologie orientée service, nous l'avons appliqué à la conception d'un capteur cardiorespiratoire intelligent, qui doit être capable de transmettre en temps réel la fréquence cardiaque, les paramètres de variabilité cardiaque et l'onde respiratoire. Celui-ci doit également disposer d'une précision de grade médical, d'une autonomie de plus de 48 heures, et être suffisamment léger pour pouvoir être attaché à une électrode collée sur le patient sans arracher celle-ci. Ce capteur devra pouvoir être utilisé dans des conditions ambulatoires. De plus nous souhaitons intégrer des capacités d'auto-adaptation afin d'augmenter l'intelligence de l'objet, et nous définissons les états suivants : *initial*, qui représente l'initialisation de tous les composants du capteur ; *normal*, dans lequel le capteur fonctionne en temps réel et calcule tous les paramètres ; *dégradé*, dans lequel le capteur transmet la moyenne de la fréquence cardiaque seulement toutes les 5 minutes pour des raisons d'économies d'énergie ; *détaché*, dans lequel le capteur se place lui-même s'il auto-détecte sa déconnexion de la peau du patient ; et enfin *stop*, qui décrit un état d'hibernation du capteur ou seul un *reset* matériel peut le redémarrer. Un diagramme représentant ces états et les transitions associées est donné FIGURE A.4. Afin d'avoir une estimation la plus précise possible des paramètres cardiaques, ceux-ci sont déterminés à partir de l'électrocardiogramme. Afin d'améliorer l'interopérabilité de notre objet, et à cause de son utilisation répandue dans les objets intelligents contraints, nous utilisons le protocole *BLE*. Bien que de nombreux capteurs semblables existent dans la littérature scientifique, comme ceux décrits dans [Mag+14; Tuo+17; Izu+15; Mas+15], à notre connaissance aucun de ces capteurs ne disposent à la fois de capacités de transmission en temps réels des paramètres combinées à la possibilité d'être configurés à distance.

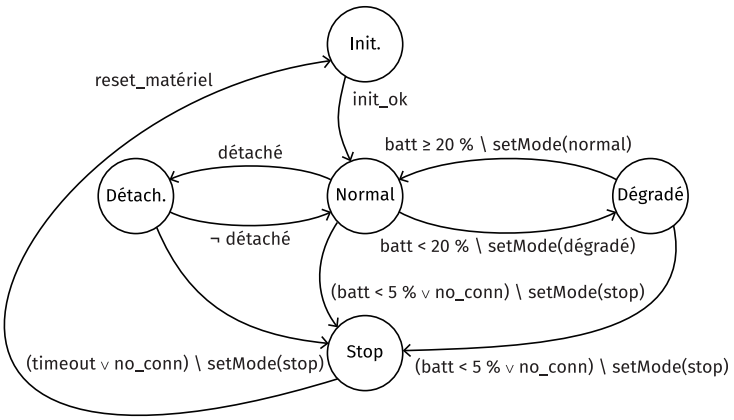


FIGURE A.4 – Diagramme d'auto-adaptation

À partir des besoins fonctionnels et non-fonctionnels détaillés dans le paragraphe précédent, nous avons sélectionné le PSoC5LP (Cypress Semiconductor, San Jose, CA) comme système sur puce principal accompagné des composants intégrés ADS1292R (Texas Instruments, Dallas, TX) pour l'acquisition et la conversion analogique-numéri-

que et BLE113 (Silicon Labs, Austin, TX) pour la communication sans fil. En effet, le PSoC5LP embarque un co-processeur de traitement du signal ainsi qu'une matrice de portes logiques programmables, ce qui nous permet d'implémenter la plupart du traitement du signal en temps réel de l'électrocardiogramme sur ces composants dédiés, et nous permet par conséquent de réduire la consommation électrique globale et donc d'améliorer l'autonomie de notre capteur. Le circuit intégré ADS1292R dispose d'une résolution et d'une fréquence d'échantillonnage de l'électrocardiogramme permettant d'acquérir celui-ci avec des qualités comparables aux standards médicaux. Finalement, le circuit BLE113 permet de gérer les communications sans fil de façon déportée, et par conséquent de réduire la charge de calcul du microcontrôleur. Le capteur est alimenté par une batterie de capacité 300 mAh. Une application Android a également été conçue afin de faciliter la collecte de données.

Nous avons donc conçu, spécifié et vérifié ce capteur cardiorespiratoire en suivant l'architecture modulaire illustrée FIGURE A.3, et une photo du capteur est montrée FIGURE A.5.(a.). Dans la suite de ce résumé, nous mettons en particulier l'accent sur les tests réalisés permettant la vérification des propriétés non-fonctionnelles, mais également la pertinence de l'utilisation ambulatoire de notre objet intelligent. Plus particulièrement, nous avons étudié l'impact des paramètres de la connexion *BLE* sur la consommation électrique du capteur, comme illustré FIGURE A.5.(b.), ce qui nous a permis de vérifier l'autonomie maximale visée.

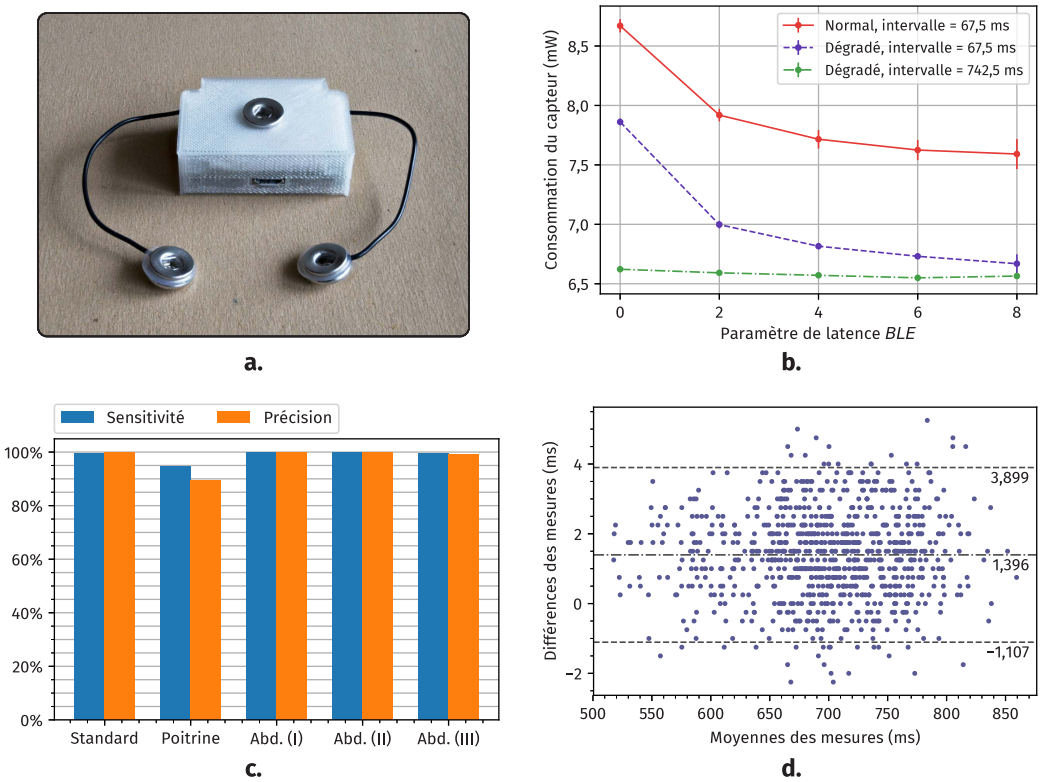


FIGURE A.5 – (a.) Capteur développé encapsulé, **(b.)** Résultats des tests de consommation, **(c.)** Sensitivité et précision du capteur, **(d.)** Diagramme de Bland-Altman avec l'instrumentation PowerLab utilisée comme référence

Afin de vérifier la pertinence de l'utilisation de notre objet intelligent dans des contextes ambulatoires, nous avons testé l'impact de la nature et de la position des

électrodes sur la sensibilité et la précision du capteur. Cette étude a été réalisée sur 8 sujets volontaires, et les résultats sont présentés FIGURE A.5.(c.). Ceux-ci illustrent que les positions standards et abdominales présentent toutes des sensibilités et précisions supérieures à 98 %, alors que la position où le capteur est placée sur la partie supérieure de la poitrine ne présente pas d'aussi bons résultats.

Finalement, nous avons évalué quantitativement la concordance entre notre capteur et une instrumentation de référence au travers d'un diagramme de Bland-Altmand, et les résultats sont illustrés FIGURE A.5.(d.). En résumé, il est important de remarquer que les données de notre capteur présentent un décalage positif d'environ 1,3 ms, et celui-ci est vraisemblablement causé par la tolérance de l'horloge interne de l'ADS1292R, qui est seulement de 1,5 %. Néanmoins, les erreurs maximales aussi bien positivement que négativement représentent systématiquement des variations inférieures à 1 % entre notre capteur et l'instrumentation de référence, et cela renforce le fait que notre objet intelligent est bien doté d'une précision de grade médical.

En conclusion, cette partie présente deux contributions : une méthode de conception orientée service pour objets intelligents, et l'application de cette méthode afin de concevoir un capteur cardiorespiratoire temps réel et auto-adaptable. L'originalité de ces contributions vient principalement de l'adoption d'une vision orientée service pour les couches les plus basses de l'internet des objets, qui permet d'améliorer la flexibilité et la modularité des objets conçus. De plus, l'addition de capacités d'auto-adaptation améliore l'intelligence de ces objets, qui peuvent ainsi être intégrés à des infrastructures auto-adaptables à plus large échelle, que nous décrivons dans la section suivante.

A.3 Infrastructure d'Auto-Adaptation Dédiée aux Objets Intelligents

Lorsqu'il s'agit de l'adaptation tant matérielle que logicielle, de nombreuses contributions ont été proposées. Les premières concernent exclusivement l'adaptation matérielle, et proviennent de la théorie classique du contrôle, qui repose principalement sur l'utilisation de boucles fermées afin de contrôler la dynamique de systèmes électromécaniques [Oga10]. La théorie du contrôle définit des architectures de boucles et de contrôleurs classiques, comme les régulateurs proportionnels, intégraux, dérivés [Sko03], qui sont utilisés pour le contrôle d'une variété de matériels (p. ex. moteurs, systèmes électrochimiques, etc.). Ces solutions reposent néanmoins sur une modélisation précise des systèmes contrôlés, traditionnellement sous la forme d'équations différentielles, et nécessitent donc une caractérisation expérimentale de ceux-ci. Ceci limite fortement leurs capacités à être utilisés pour des systèmes hybrides présentant une dynamique à la fois discrète et continue. Des solutions de la communauté des systèmes cyber-physiques permettent de contrôler ce type de systèmes [Lee08], mais elles présentent le même type de limites que les solutions uniquement continues (c.-à-d. l'effort nécessaire pour développer des modèles précis des systèmes à contrôler).

Les informaticiens étudient également l'auto-adaptation de systèmes logiciels, et ont proposé une variété de solutions afin d'équiper ceux-ci de capacités de réaction aux sollicitations à la fois internes et externes. Une contribution majeure pour l'adaptation

de tels systèmes est la boucle *MAPE-K*[1] [KC03]. Celle-ci définit une architecture de référence pour la création de composants logiciels autonomes, et les boucles *MAPE-K* sont par conséquent extrêmement souples et flexibles, et elles permettent aux systèmes logiciels de s'auto-configurer, s'auto-optimiser ou bien encore s'auto-guérir [KC03]. Des contributions telles que [Vil+13] proposent la coordination de boucles *MAPE-K* pour la gestion dynamique de l'auto-adaptation. Plus particulièrement, l'aspect dynamique est introduit par le fait que les systèmes proposés dans [Vil+13] sont capables de réagir aux changements d'objectifs d'adaptation ou d'infrastructure de supervision durant l'exécution. Cependant, les contributions basées sur des boucles de type *MAPE-K* sont fréquemment exclusivement logicielles, et ne sont par conséquent pas capables de représenter les contraintes intrinsèques à l'internet des objets telles que les limitations en termes de ressources ou la nature hybrides des objets intelligents le constituant.

Finalement, nous avons isolé une dernière catégorie de contributions intéressantes dans le contexte de l'internet des objets, produite par la communauté dédiée à l'étude des systèmes réactifs. Ceux-ci sont définis comme des systèmes réagissant à une variété de sollicitations extérieures, et de nombreux outils permettant leur étude sont proposés. C'est particulièrement le cas des systèmes synchrones, qui sont définis dans ce cadre comme des systèmes assez puissants pour que le temps de calcul d'une réponse à une sollicitation externe soit négligeable comparé au flux de sollicitations nécessitant une réaction [Gam10]. Si cette hypothèse est vérifiée, les systèmes peuvent être modélisés sous la forme de systèmes de transition d'états étiquetés, et il existe des techniques permettant la synthèse automatique de contrôleurs discrets et synchrones pour la coordination de plusieurs sous-systèmes de ce type. Celle-ci est généralement appelée synthèse de contrôleurs discrets, et elle consiste en la génération automatique de code impératif implémentant le contrôleur discret vérifiant les objectifs d'adaptation spécifiés lors du processus de synthèse.

En conclusion, les solutions existantes ne satisfont pas tous les besoins de l'auto-adaptation appliquée à l'internet des objets. En effet, les solutions issues de la théorie du contrôle classique ou des systèmes cyber-physiques nécessitent la modélisation exhaustive des systèmes à contrôler, ce qui n'est pas toujours réalisable pour les systèmes de l'internet des objets. De plus, les solutions orientées logiciel comme les boucles *MAPE-K* ne permettent pas de prendre en compte les contraintes des objets intelligents, mais permettent d'établir des architectures flexibles et modulaires. Finalement, les outils relatifs à la synthèse de contrôleurs discrets pour les systèmes réactifs synchrones apportent sont pertinent dans le contexte de l'internet des objets. En effet, ceux-ci utilisent des modèles simples à base de systèmes de transition d'états étiquetés, qui peuvent être appliqués facilement aux objets intelligents. De plus, l'aspect automatique et formel de ces outils permettent de produire du code correct par construction, ce qui ajoute un degré de certitude vis-à-vis du comportement attendu des systèmes synthétisés. Notre contribution se situe donc à l'interface de ces trois champs, et propose une solution pour l'auto-adaptation dynamique de systèmes de l'internet des objets basée sur la synthèse de contrôleurs discrets intégrée à des boucles de type *MAPE-K*, elles-mêmes interagissant à la manière de boucles fermées issues du la théorie du contrôle classique.

1. MAPE-K : *Monitor, Analyze, Plan, Execute and Knowledge* - Supervise, analyse, planifie et exécute en utilisant une base de connaissances partagée.

A.3.1 Présentation de l'Infrastructure Hybride

Notre infrastructure dynamique d'auto-adaptation pour l'internet des objets est introduite FIGURE A.6. Celle-ci se concentre particulièrement sur la préservation d'une qualité de service satisfaisante durant tout le cycle d'exécution du système, et met donc l'accent sur les propriétés non-fonctionnelles. Notre démarche a d'abord été d'établir une architecture entièrement synchrone basée sur la séparation des préoccupations entre la gestion des objectifs d'adaptation, le processus d'adaptation et l'infrastructure de supervision du système. Ainsi, comme illustré FIGURE 5.1.(a.), nous définissons trois boucles de contrôle basées sur la topologie *MAPE-K* : la boucle de contrôle des objectifs, celle d'adaptation et enfin celle de supervision des propriétés non-fonctionnelles. Ces trois boucles communiquent au travers de quatre interactions, qui représentent les *objectifs de contrôle*, les *superviseurs invalidés*, le *contexte interne* et enfin des *actions d'adaptation* préventives. Le processus d'adaptation est implémenté au travers d'un contrôleur discret, généré automatiquement en utilisant des outils de synthèse automatique. Ainsi, nous sommes capables de gérer des changements dans les objectifs de contrôle, mais également dans l'infrastructure de supervision, ce qui est particulièrement pertinent dans le contexte de l'internet des objets considérant que celui-ci représente un réseau dynamique que les objets intelligents peuvent joindre ou quitter à tout moment, mais aussi considérant le fait que l'internet des objets est en constante interaction avec le monde physique, qui lui-même est en évolution permanente.

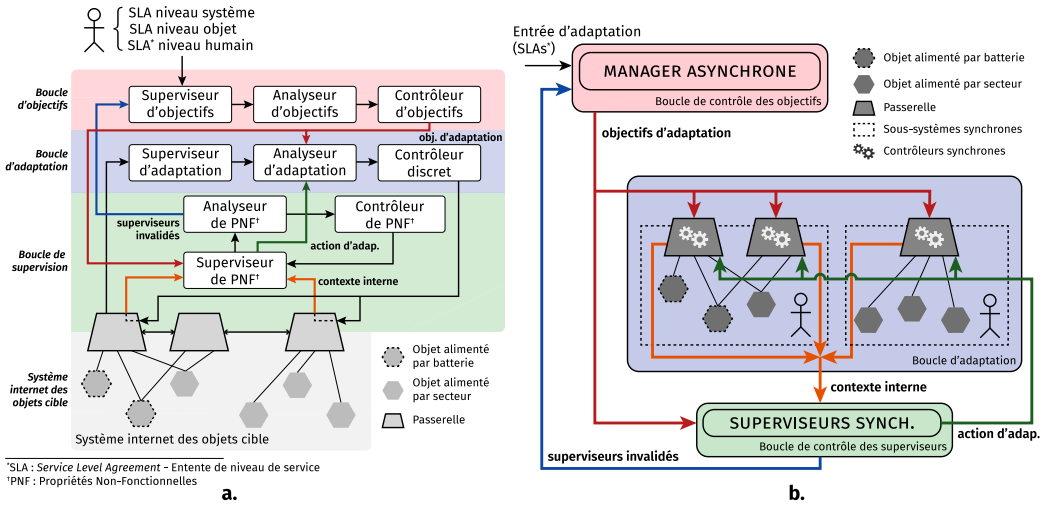


FIGURE A.6 – Versions synchrone (**a.**) et hybride (**b.**) de l'infrastructure d'auto-adaptation

En plus de cette infrastructure synchrone d'auto-adaptation dynamique, nous avons également conçu un langage de spécification d'objectifs d'adaptation déclaratif à base de règles, et un exemple est illustré FIGURE A.7. En effet, les langages impératifs présentent de fortes limites en termes de maintenance et d'évolutivité car ils sont utilisés pour spécifier *comment* un système réalise des objectifs prédéfinis. En spécifiant *quels* sont les objectifs à atteindre, les solutions déclaratives abrègent le processus de développement car il n'est plus nécessaire de caractériser l'intégralité du déroulement de l'exécution. Notre système déclaratif nous permet de réduire la base de code nécessaire à la spécification de l'auto-adaptation de systèmes internet des objets complexes, mais abaisse également le niveau d'entrée en permettant à des non-experts de spécifier

eux-mêmes les systèmes en fonctions de leurs exigences. De plus, nous divisons la spécification des objectifs en trois catégories : niveau objet, niveau système et niveau humain. Cela permet de différencier les niveaux de granularité nécessaires à la caractérisation de systèmes cohérents à la fois à l'échelle globale mais également à l'échelle locale (c.-à-d. au niveau des objets). De plus, les variations inter-patients dans le cas de systèmes de santé requièrent la possibilité d'adapter le système à leur différentes exigences. Ces spécifications sont réalisées en termes de qualité de service à préserver, et les utilisateurs finaux caractérisent *quelles* sont les configurations du système permettant la préservation de ces qualités.

```
RULE "R1" // Nom de la règle
  ON HeartNormal // Nom de la qualité de service associée
  IF HRSensorType(State == "normal") // Préconditions
     $fpos: FPosSensorType()
  DO $ppg.setState("off") // Action d'adaptation
     $fpos.setState("off")
END

RULE "R2"
  ON HeartStopped
  IF HRSensorType(State == "stop")
     $ppg: PPGSensorType()
     $fpos: FPosSensorType()
  DO $ppg.setState("high")
     $fpos.setState("on")
END
```

FIGURE A.7 – Exemple de règles spécifiées en utilisant notre langage

Bien que la solution synchrone accompagnée de son langage d'auto-adaptation à base de règles représente une contribution intéressante, elle est fortement limitée par son aspect exclusivement synchrone, qui peut poser des problèmes d'évolutivité. Bien que l'hypothèse des systèmes synchrones soit vérifiée pour des infrastructures de l'internet des objets de taille modeste, ce n'est plus le cas si les systèmes dépassent une certaine taille. En effet, bien que les passerelles utilisées dans l'internet des objets soient en général bien plus puissantes que les objets qu'elles contrôlent, si ceux-ci deviennent trop nombreux, la masse d'évènements à gérer par ces passerelles peut devenir trop importante, ce qui peut causer une invalidation de la vérification de cette hypothèse. Pour palier à cette limitation d'évolutivité, nous introduisons la solution hybride présentée FIGURE 5.1.(b.). Celle-ci introduit une gestion asynchrone des objectifs d'adaptation, qui sont ensuite retransmis aux sous-systèmes synchrones de supervision des propriétés non-fonctionnelles et d'adaptation. La conversion asynchrone vers synchrone est implémentée au travers de tampons stockant les évènements asynchrones de façon séquentielle, qui sont ensuite lus et gérés par les systèmes synchrones à chaque itération de leur boucle principale. Notre système d'auto adaptation dynamique final consiste donc en la coordination asynchrone de sous-systèmes d'adaptation et de supervision synchrones. En plus de la définition théorique résumée ci-dessus, nous avons implémenté une preuve de concept afin d'illustrer l'aspect pratique de notre infrastructure et de réaliser des expériences, notamment afin d'évaluer son évolutivité.

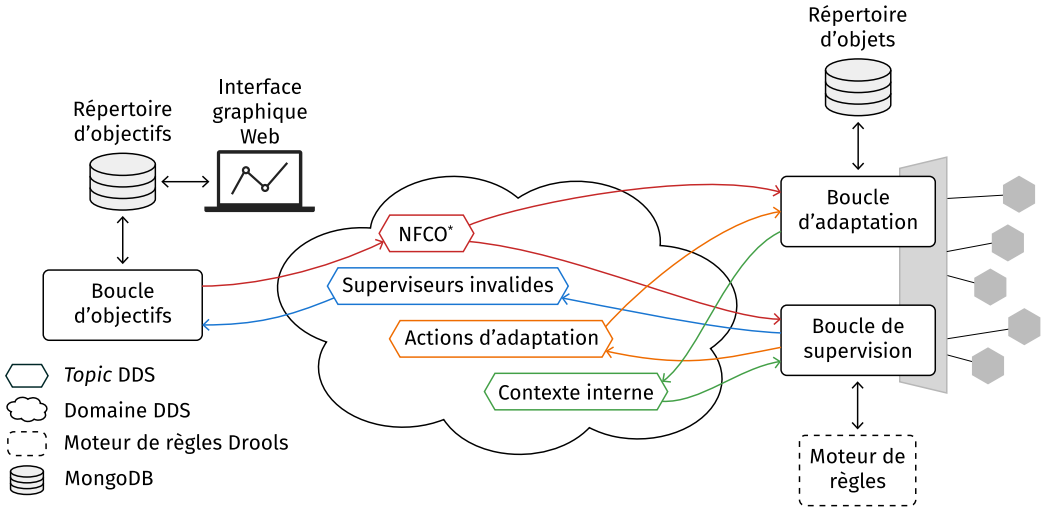
A.3.2 Implémentation de l'Infrastructure d'Auto-Adaptation

Dans cette section, nous résumons l'implémentation et l'évaluation de notre infrastructure d'auto-adaptation dynamique pour l'internet des objets. L'architecture de notre preuve de concept est présentée FIGURE A.8. Afin de maximiser l'interopérabilité, la fiabilité et la robustesse de notre solution, nous avons choisi le standard *Data Distribution Service (DDS)* pour implémenter toutes les communications de celle-ci. En effet, *DDS* est spécifiquement conçu pour une utilisation en milieu contraint et critique, et son architecture sans *broker* central élimine ce potentiel point de panne unique (dans le sens *single point of failure*). De plus, ce standard présente de bonne performances en termes d'évolutivité et d'utilisation en milieu contraint [CK16; BPM16], ce qui le rend particulièrement pertinent pour une application dans le domaine de l'internet des objets. En résumé, le standard *DDS* est de type publication-abonnement, et les données circulent sur des *topics* désignés par leur nom. Contrairement à *MQTT*, qui est également très utilisé pour les solutions dans le domaine de l'internet des objets, les données circulant sur les *topics DDS* sont préalablement caractérisées en utilisant un langage de spécification d'interface, et elles sont par conséquent typées. En plus, ce standard permet de partitionner le domaine d'information global en plusieurs sous-domaines imbriqués, ce qui permet potentiellement d'adopter un modèle de système hiérarchique (p. ex. ville, bâtiment, pièce) pouvant être utilisé pour filtrer les données circulant dans le système. *DDS* propose également une gestion très précise de la qualité de service en offrant 22 paramètres configurables. Les communications implémentées en utilisant ce protocole sont à la fois les interactions entre les boucles, mais également celles provenant des objets intelligents. Si un objet intelligent n'utilise pas un protocole TCP/IP, des convertisseurs vers *DDS* peuvent aisément être créés.

Nous avons utilisé la librairie Vert.x pour l'implémentation de la boucle de contrôle des objectifs asynchrone. En effet, celle-ci implémente le *reactor pattern*, qui permet de créer des systèmes asynchrones réactifs évolutifs. De plus, cette librairie est extrêmement légère (la partie centrale ne dépassant pas la centaine de kilooctets), et son exécution repose sur la *Java Virtual Machine (JVM)*, ce qui la rend extrêmement portable. Vert.x est basée sur une boucle d'évènements centrale, qui ne doit jamais être bloquée afin de garantir des performances optimales. Les concepteurs utilisant Vert.x doivent donc se concentrer sur la construction d'infrastructures exclusivement asynchrones et utiliser au maximum des appels de fonction non-bloquants.

En plus de ces deux outils, nous avons utilisé la base de données NoSQL MongoDB pour l'implémentation des répertoires d'objets et d'objectifs. Celle-ci permet de stocker des documents en utilisant le langage *JavaScript Object Notation (JSON)*, et les bonnes performances en lecture et écriture [PND14] rendent son utilisation pour l'internet des objets pertinente. En ce qui concerne le moteur de règles, nous avons choisi d'utiliser Drools, principalement pour sa flexibilité (il est accompagné d'un langage de spécification des règles très avancé, mais également de la possibilité de définir ses propres langages), ainsi que pour ses facultés de traitement des événements complexes, qui permettent d'analyser sans difficulté une masse d'évènements pour en détecter ceux qui sont significatifs et y attacher une logique applicative.

Étant donné que la plupart des outils à la base de notre prototype sont basés sur le langage Java, nous l'avons utilisé dans sa version 1.8 pour implémenter la totalité de la preuve de concept. Nous avons également implémenté une interface graphique



*NFCO : *Non-Functional Control Objectives* - Objectifs de contrôle non-fonctionnels

FIGURE A.8 – Architecture de l'implémentation de l'infrastructure d'auto-adaptation

Web, afin que les utilisateurs finaux puissent aisément spécifier les règles d'adaptation nécessaires à leur application spécifique. Les boucles de supervision et d'adaptation ont été implémentées de façon synchrone en utilisant une simple boucle logicielle infinie. En ce qui concerne la synthèse de contrôleurs discrets et synchrones, nous avons choisi d'utiliser le langage de programmation synchrone Heptagon/BZR [DR10] accompagné du synthétiseur de contrôleurs Sigali [Mar+00; DMR10; DRM13]. Ce langage est capable de générer du code Java, qui peut être compilé et déployé sur une variété de passerelles (p. ex. Raspberry Pi, Intel Edison, etc.).

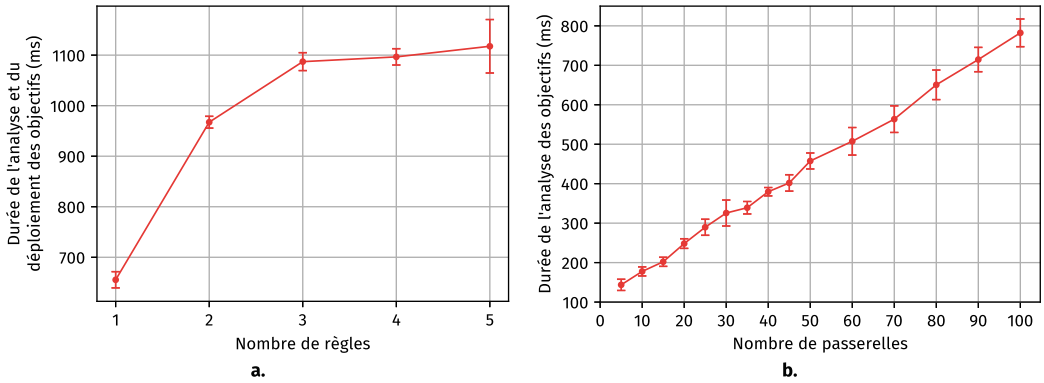


FIGURE A.9 – Étude de l'évolutivité du système par rapport au nombre de règles (**a.**) et du nombre de passerelles (**b.**)

Une fois la preuve de concept implémentée, nous avons testé son évolutivité par rapport à deux critères : le nombre de règles déployées dans chaque sous-système et le nombre de sous-systèmes (c.-à-d. le nombre de passerelles). Les résultats sont présentés FIGURE A.9. En effet, bien que le nombre de règles déployées dans un sous-système ne dépasse jamais quelques dizaines étant donné que la taille de ceux-ci est limitée par l'hypothèse des systèmes synchrones, il est important de vérifier qu'une augmentation du nombre de règles n'entraîne pas une explosion du temps de déploiement. La FIGURE A.9.(a.) vient confirmer ce fait, et on peut observer que la pente du temps de

déploiement des objectifs tend à diminuer lorsque le nombre d'objectifs augmente, ce qui confirme l'évolutivité de notre système vis-à-vis du nombre de règles. C'est aussi le cas pour l'évolutivité de l'infrastructure par rapport au nombre de passerelles à gérer. En effet, comme illustré FIGURE A.9.(b.), le temps d'analyse des objectifs de contrôle évolue linéairement avec le nombre de passerelles. Nous n'observons par conséquent pas d'explosion du temps d'analyse, ce qui encore une fois est un signe encourageant pour l'évolutivité de notre système et sa capacité à gérer une masse d'objets intelligents importante.

A.4 Conclusion

En conclusion, nous avons développé au cours de cette thèse deux axes de recherche principaux. Le premier consiste en l'amélioration de la conception des objets intelligents en intégrant une dimension orientée service et formelle à leur cycle de développement. Celle-ci permet d'améliorer l'interopérabilité de ces objets, mais également de minimiser les problèmes apportés par la forte hétérogénéité du matériel utilisé pour les implémenter et d'améliorer leur fiabilité en se basant sur leur vérification formelle. Notre contribution principale pour cet axe est la proposition d'une méthode de conception orientée service pour l'internet des objets intégrant une dimension formelle, ainsi que son application à la création d'un capteur cardiorespiratoire de grade médical.

Le deuxième axe de recherche aborde la problématique de l'auto-adaptation pour les objets intelligents et l'internet des objets. En effet, celle-ci représente une première étape vers une intelligence globale de ce type de systèmes, mais l'auto-adaptation permet également d'améliorer la fiabilité et la sureté de ceux-ci en les rendant capables de réagir à une variété de situations internes ou externes durant leur exécution. Dans cette partie, nous proposons une infrastructure d'auto-adaptation dynamique pour l'internet des objets, capable à la fois de gérer des changements dans les objectifs d'adaptation mais également dans le système de supervision. Nous présentons aussi une implémentation d'une preuve de concept de cette infrastructure, que nous avons évalué vis-à-vis de son évolutivité.

Bien que nos travaux introduisent une certaine verticalité dans la résolution des défis de l'internet des objets, ils restent restreints aux couches relativement basses de celui-ci, et présente par conséquent des limitations, qui mettent en avant quelques directions de recherche intéressantes :

— *La création d'un modèle unifié pour l'internet des objets* : Une piste intéressante pour l'unification des représentations des objets intelligents est l'utilisation d'artéfacts métiers, qui permettent la modélisation simultanée des données des objets ainsi que de leur cycle de vie [NC03]. Des contributions comme [ABA17] proposent de les utiliser dans le contexte de l'internet des objets, et les adapter à notre vision de celui-ci représente une direction de recherche potentielle.

— *La définition d'un cadre formel exhaustif* : Dans nos contributions, nous utilisons deux formalismes différents pour la spécification formelle des objets lors de leur création et de leur utilisation. Un modèle unifié serait bénéfique dans le sens ou il réduirait le temps de développement, et la logique linéaire représente une piste intéressante pour la modélisation complète des objets intelligents. En effet, ses capacités à représenter les ressources et leur consommation représentent

un avantage indéniable dans ce contexte, et elle a déjà été utilisée pour la composition des services Web dans [PFW12; RS04]. Le manque d'outils de preuve automatiques pour la logique linéaire représente néanmoins un inconvénient à son utilisation.

— *L'ajout d'une couche sémantique* : Celle-ci permettrait d'améliorer la compatibilité entre les objets connectés en fournissant un moyen de raisonner sur les données produites par ceux-ci et en permettant d'inférer de façon intelligente des équivalences entre des sources de données hétérogènes. Des contributions telles que [Ter+17] proposent d'utiliser des raisonneurs automatiques dans un contexte de l'internet des objets, mais les technologies utilisées restent de relativement haut niveau, et leurs applications aux couches les plus basses de celui-ci restent un défi.

— *L'extension de l'auto-adaptation aux couches basses* : Ici, notre infrastructure dynamique d'auto-adaptation est limitée à la coordination d'objets intelligents au travers de leur reconfiguration. Une extension aux couches les plus basses, ou les objets ne sont plus seulement reconfigurés mais ou leur micrologiciel est changé à distance, permettrait de répondre à des cas d'auto-adaptation plus larges. Ceci est réalisable techniquement étant donné que les langages de programmation synchrones permettent de générer du code C bas niveau, et cette vision a été adoptée pour le développement de pilotes pour objets contraints [BMM13].

— *L'intégration de la sécurité* : Celle-ci représente une lacune importante de nos travaux, considérée la sensibilité des données collectées par l'internet des objets, mais également son utilisation pour de nombreuses applications critiques. La sécurité est cependant un sujet d'étude très large, et la taille importante des systèmes dans ce contexte ainsi que l'hétérogénéité des objets connectés apportent de nouveaux défis à résoudre pour les chercheurs en sécurité. Une direction de recherche intéressante se situe dans le domaine de la sécurité décentralisée et distribuée dans la *blockchain*, et des contributions telles que [Zhu+17] ou [OAA16] décrivent des solutions pour la gestion de l'identité ou du contrôle d'accès dans l'internet des objets. Intégrer de telles solutions à notre vision de celui-ci serait extrêmement bénéfique et améliorerait considérablement la sécurité de nos systèmes.

APPENDIX B

Cardiorespiratory Sensor Schematic

This appendix illustrates the complete cardiorespiratory sensor schematic which was used for Printed Circuit Board (PCB) routing and hardware implementation.

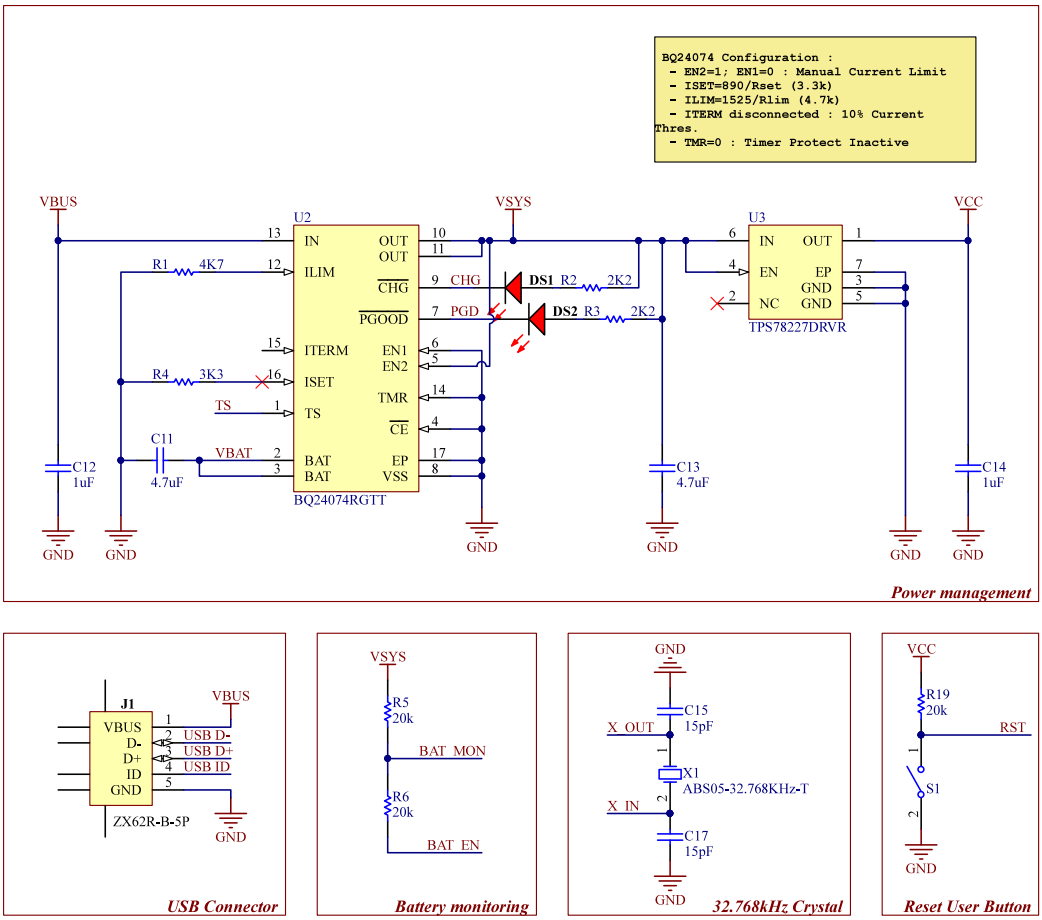


Figure B.1 – Power management and hardware connectivity

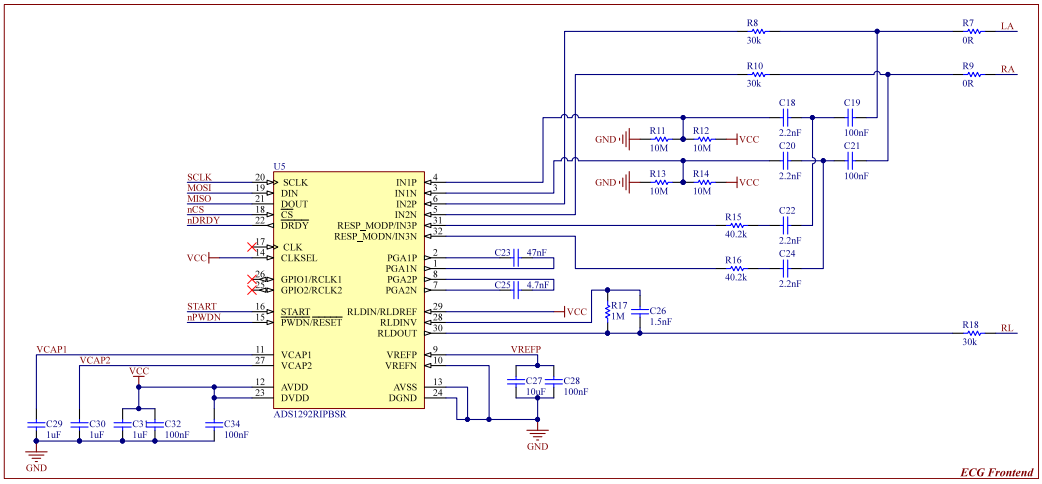


Figure B.2 – Analog front end interfacing

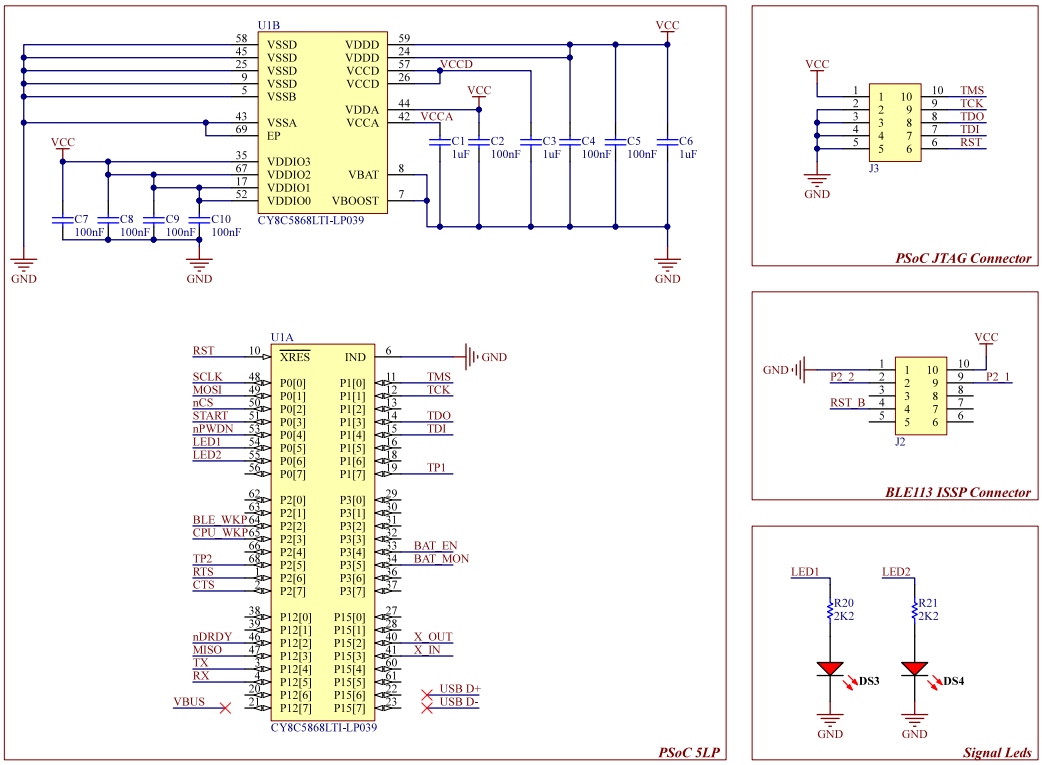


Figure B.3 – PSOC5LP and additional hardware connectivity

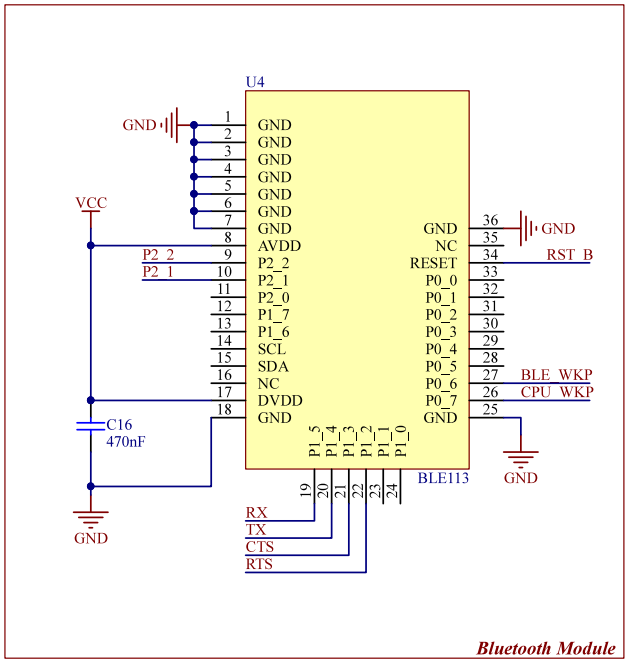


Figure B.4 – BLE113 interfacing

Bibliography

[ABA17] M. Abi Assaf, Y. Badr, and Y. Amghar. "A Continuous Query Language for Stream-Based Artifacts." *Proceedings of the 28th International Conference on Database and Expert Systems Applications (DEXA)*. Springer, 2017, pp. 80–89.

[AL12] O. Adeluyi and J.-A. Lee. "Realtime Detection of ECG Signal R-Peaks Using a Lightweight R-READER Algorithm." *International Journal of Intelligent Information Prosessing*, vol. 3, no. 3, 2012, pp. 1–12.

[Ala+14] M. B. Alaya, Y. Banouar, T. Monteil, C. Chassot, and K. Drira. "OM2M: Extensible ETSI-Compliant M2M Service Platform with Self-Configuration Capability." *Procedia Computer Science*, vol. 32, 2014, pp. 1079–1086.

[Ala+15] M. B. Alaya, S. Medjiah, T. Monteil, and K. Drira. "Toward Semantic Interoperability in oneM2M Architecture." *IEEE Communications Magazine*, vol. 53, no. 12, 2015, pp. 35–41.

[AlF+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications." *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, 2015, pp. 2347–2376.

[Alo+04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. "Web Services." *Web Services: Concepts, Architectures and Applications*. Springer, 2004, pp. 123–149.

[Alt14] Altera. *Bare-Metal, RTOS, or Linux? Optimize Real-Time Performance with Altera SoCs*. 2014, pp. 1–12.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. "Symbolic Controller Synthesis for Discrete and Timed Systems." *Proceedings of the 3rd International Workshop on Hybrid Systems*. Springer, 1995, pp. 1–20.

[Ang+16] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos. "Model Predictive Control for Software Systems with CobRA." *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2016, pp. 35–46.

[Arm+05] S. W. Arms, C. P. Townsend, D. L. Churchill, J. H. Galbreath, and S. W. Mundell. "Power Management for Energy Harvesting Wireless Sensors." *Smart Structures and Materials: Smart Electronics, MEMS, BioMEMS, and Nanotechnology*. SPIE, 2005, pp. 1–9.

[Ars+08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. "SOMA: A Method for Developing Service-Oriented Solutions." *IBM Systems Journal*, vol. 47, no. 3, 2008, pp. 377–396.

[ARS15] P. Arcaini, E. Riccobene, and P. Scandurra. "Modeling and Analyzing Mape-K Feedback Loops for Self-Adaptation." *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2015, pp. 13–23.

[AS87] B. Alpern and F. B. Schneider. "Recognizing Safety and Liveness." *Distributed Computing*, vol. 2, no. 3, 1987, pp. 117–126.

[ASD17] R. Abid, G. Salaün, and N. De Palma. "Asynchronous Synthesis Techniques for Coordinating Autonomic Managers in the Cloud." *Science of Computer Programming*, vol. 146, 2017, pp. 87–103.

[ASL93] P. J. Antsaklis, J. A. Stiver, and M. Lemmon. "Hybrid System Modeling and Autonomous Control Systems." *Hybrid Systems*. Springer, 1993, pp. 366–392.

[Ath99] D. Atherton. "PID Controller Tuning." *Computing & Control Engineering Journal*, vol. 10, no. 2, 1999, pp. 44–50.

[AWB14] F. J. Acosta Padilla, F. Weis, and J. Bourcier. "Towards a Model@Runtime Middleware for Cyber Physical Systems." *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing (MW4NG)*. ACM, 2014, pp. 1–6.

[Bab+11] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad. "Proposed Embedded Security Framework for Internet of Things." *Proceedings of the 2nd International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)*. IEEE, 2011, pp. 1–5.

[Bag+17] M. Bagheri, I. Akkaya, E. Khamespanah, N. Khakpour, M. Sirjani, A. Movaghar, and E. A. Lee. "Coordinated Actors for Reliable Self-Adaptive Systems." *Proceedings of the 13th International Conference on Formal Aspects of Component Software (FACS)*. Springer, 2017, pp. 241–259.

[Bar06] F. Barbier. "MDE-Based Design and Implementation of Autonomic Software Components." *Proceedings of the 5th International Conference on Cognitive Informatics (ICCI)*. IEEE, 2006, pp. 163–169.

[Bau+13] M. Bauer, M. Boussard, N. Bui, J. De Loof, C. Magerkurth, S. Meissner, A. Nettsträter, J. Stefa, M. Thoma, and J. W. Walewski. "IoT Reference Architecture." *Enabling Things to Talk*. Springer, 2013, pp. 163–211.

[BB91] A. Benveniste and G. Berry. "The Synchronous Approach to Reactive and Real-Time Systems." *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1270–1282.

[BBF09] G. Blair, N. Bencomo, and R. B. France. "Models@run.Time." *Computer*, vol. 42, no. 10, 2009, pp. 22–27.

[BdS91] F. Boussinot and R. de Simone. "The ESTEREL Language." *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1293–1304.

[Bel08] M. Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. 1st Edition. John Wiley & Sons, 2008.

[Bem+02] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos. "The Explicit Linear Quadratic Regulator for Constrained Systems." *Automatica*, vol. 38, no. 1, 2002, pp. 3–20.

[Ben+00] D. Benitez, P. Gaydecki, A. Zaidi, and A. Fitzpatrick. "A New QRS Detection Algorithm Based on the Hilbert Transform." *Proceedings of the 2000 International Conference on Computer in Cardiology*. IEEE, 2000, pp. 379–382.

[Ben+05] B. Benatallah, M.-S. Hacid, A. Leger, C. Rey, and F. Toumani. "On Automating Web Services Discovery." *The VLDB Journal*, vol. 14, no. 1, 2005, pp. 84–96.

[Ben12] L. Bengtsson. "Direct Analog-to-Microcontroller Interfacing." *Sensors and Actuators A: Physical*, vol. 179, 2012, pp. 105–113.

[BIT17] B. Billet, V. Issarny, and G. Texier. "Composing Continuous Services in a CoAP-Based IoT." *Proceedings of the 6th International Conference on AI & Mobile Services (AIMS)*. IEEE, 2017, pp. 46–53.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. "The Palladio Component Model for Model-Driven Performance Prediction." *Journal of Systems and Software*, vol. 82, no. 1, 2009, pp. 3–22.

[BL03] B. Batson and L. Lamport. "High-Level Specifications: Lessons from Industry." *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, 2003, pp. 242–261.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. "Synchronous Programming with Events and Relations: The SIGNAL Language and Its Semantics." *Science of Computer Programming*, vol. 16, no. 2, 1991, pp. 103–149.

[BM14] N. Berthier and H. Marchand. "Discrete Controller Synthesis for Infinite State Systems with ReaX." *IFAC Proceedings Volumes*, vol. 47, no. 2, 2014, pp. 46–53.

[BMM13] N. Berthier, F. Maraninchi, and L. Mounier. "Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems." *ACM Transactions on Embedded Computing Systems*, vol. 12, 1s 2013, pp. 1–26.

[BMP07] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*. Morgan Kaufmann, 2007. 462 pp.

[Bou+16] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. "Execution Framework of the GEMOC Studio (Tool Demo)." *Proceedings of the 9th International Conference on Software Language Engineering (SLE)*. Amsterdam, Netherlands: ACM, 2016, pp. 84–89.

[BPM16] Z. B. Babovic, J. Protic, and V. Milutinovic. "Web Performance Evaluation for Internet of Things Applications." *IEEE Access*, vol. 4, 2016, pp. 6974–6992.

[Bro+18] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber. "A Component Architecture for the Internet of Things." *Proceedings of the IEEE*, 2018, pp. 1–16.

[Bru+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. "The FRACTAL Component Model and Its Support in Java." *Software: Practice and Experience*, vol. 36, no. 11-12, 2006, pp. 1257–1284.

[Bru+09] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. "Engineering Self-Adaptive Systems Through Feedback Loops." *Software Engineering for Self-Adaptive Systems*. Berlin: Springer, 2009, pp. 48–70.

[BXA17] S. B. Baker, W. Xiang, and I. Atkinson. "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities." *IEEE Access*, vol. 5, 2017, pp. 26521–26544.

[Cai+00] X. Cai, M. Lyu, Kam-Fai Wong, and Roy Ko. "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes." *Proceedings of the 7th Asia-Pacific Software Engeering Conference (APSEC)*. IEEE, 2000, pp. 372–379.

[Cal15] A. Calabria. *Understanding Lead-Off Detection in ECG*. Texas Instruments, 2015, pp. 1–16.

[Can+14] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz, and L. Gurgen. "ECA Rules for IoT Environment: A Case Study in Safe Design." *Proceedings of the 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE, 2014, pp. 116–121.

[Car+09] A. Cardenas, S. Amin, B. Sinopoli, A. Giani, A. Perrig, and S. Sastry. "Challenges for Securing Cyber Physical Systems." *Proceedings of the 1st Workshop on Future Directions in Cyber-Physical Systems Security (CPSSW)*. 2009, pp. 1–7.

[Car+17] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. "MOSES: A Platform for Experimenting with QoS-Driven Self-Adaptation Policies for Service Oriented Systems." *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017, pp. 409–433.

[Cas+11] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. "Web Services for the Internet of Things through CoAP and EXI." *Proceedings of the 2011 International Conference on Communications Workshops (ICC)*. IEEE, 2011, pp. 1–6.

[CCC95] Cuiwei Li, Chongxun Zheng, and Changfeng Tai. "Detection of ECG Characteristic Points Using Wavelet Transforms." *IEEE Transactions on Biomedical Engineering*, vol. 42, no. 1, 1995, pp. 21–28.

[CD99] I. I. Christov and I. K. Daskalov. "Filtering of Electromyogram Artifacts from the Electrocardiogram." *Medical Engineering & Physics*, vol. 21, no. 10, 1999, pp. 731–736.

[CDP17] T. Cerny, M. J. Donahoo, and J. Pechanec. "Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems." *Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS)*. ACM, 2017, pp. 228–235.

[CDR14] J. Cano, G. Delaval, and E. Rutten. "Coordination of ECA Rules by Verification and Control." *Proceedings of the 16th International Conference on Coordination Models and Languages (COORDINATION)*. Ed. by E. Kühn and R. Pugliese. Springer, 2014, pp. 33–48.

[Cer+15] P. Cerwall, P. Jonsson, S. Carson, R. Möller, and S. Bävertoft. *Ericsson Mobility Report*. Ericsson, 2015.

[CG79] D. Clarke and P. Gawthrop. "Self-Tuning Control." *Proceedings of the Institution of Electrical Engineers*, vol. 126, no. 6, 1979, p. 633.

[Cic+17] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, and R. Spalazzese. "Model-Driven Engineering for Mission-Critical IoT Systems." *IEEE Software*, vol. 34, no. 1, 2017, pp. 46–53.

[CK07] S. H. Chang and S. D. Kim. "A Service-Oriented Analysis and Design Approach to Developing Adaptable Services." *Proceedings of the 2007 International Conference on Services Computing (SCC)*. IEEE, 2007, pp. 204–211.

[CK16] Y. Chen and T. Kunz. "Performance Evaluation of IoT Protocols under a Constrained Wireless Access Network." *Proceedings of the 2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*. IEEE, 2016, pp. 1–7.

[CM12] G. Cugola and A. Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." *ACM Computing Surveys*, vol. 44, no. 3, 2012, pp. 1–62.

[Com+13] B. Combemale, J. Deantoni, R. France, F. Boulanger, S. Mosser, M. Pantel, B. Rumpe, R. Salay, and M. Schindler. "Report on the First Workshop on the Globalization of Modeling Languages." *Proceedings of the 1st International Workshop on the Globalization of Modeling Languages (GEMOC)*. Miami, Florida, USA: CEUR, 2013, pp. 3–13.

[Cos+16] B. Costa, P. F. Pires, F. C. Delicato, W. Li, and A. Y. Zomaya. "Design and Analysis of IoT Applications: A Model-Driven Approach." *14th International Conference on Dependable, Autonomic and Secure Computing, 14th International Conference on Pervasive Intelligence and Computing, 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2016, pp. 392–399.

[CPP05] J.-L. Colaço, B. Pagano, and M. Pouzet. "A Conservative Extension of Synchronous Data-Flow with State Machines." *Proceedings of the 5th International Conference on Embedded Software (EMSOFT)*. ACM, 2005, pp. 173–182.

[CRI12] M. Caporuscio, P.-G. Raverdy, and V. Issarny. "ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking." *IEEE Transactions on Services Computing*, vol. 5, no. 1, 2012, pp. 86–98.

[CS17] F. Ciccozzi and R. Spalazzese. "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering." *Intelligent Distributed Computing X*. Vol. 678. Springer, 2017, pp. 67–76.

[CT05] G. D. Clifford and L. Tarassenko. "Quantifying Errors in Spectral Estimates of HRV Due to Beat Replacement and Resampling." *IEEE Transactions on Biomedical Engineering*, vol. 52, no. 4, 2005, pp. 630–638.

[CZ16] M. Chiang and T. Zhang. "Fog and IoT: An Overview of Research Opportunities." *IEEE Internet of Things Journal*, vol. 3, no. 6, 2016, pp. 854–864.

[dDeu+06] S. de Deugd, R. Carroll, K. E. Kelly, B. Millett, and J. Ricker. "SODA: Service Oriented Device Architecture." *IEEE Pervasive Computing*, vol. 5, no. 3, 2006, pp. 94–96.

[Dem+13] A. Dementyev, S. Hodges, S. Taylor, and J. Smith. "Power Consumption Analysis of Bluetooth Low Energy, ZigBee and ANT Sensor Nodes in a Cyclic Sleep Scenario." *Proceedings of the 1st International Wireless Symposium (IWS)*. IEEE, 2013, pp. 1–4.

[Den+06] G. Denaro, M. Pezzé, D. Tosi, and D. Schilling. "Towards Self-Adaptive Service-Oriented Architectures." *Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB)*. ACM, 2006, pp. 10–16.

[DG97] G. De Micheli and R. K. Gupta. "Hardware/Software Co-Design." *Proceedings of the IEEE*, vol. 85, no. 3, 1997, pp. 349–365.

[DMR10] G. Delaval, H. Marchand, and E. Rutten. "Contracts for Modular Discrete Controller Synthesis." *Proceedings of the 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (CASES)*. ACM, 2010, pp. 57–66.

[dNOW07] F. A. M. do Nascimento, M. F. Oliveira, and F. R. Wagner. "ModES: Embedded Systems Design Methodology and Tools Based on MDE." *Proceedings of the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*. IEEE, 2007, pp. 67–76.

[Döm+08] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. "System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design." *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, 2008, pp. 1–13.

[Dom+16a] J. Domaszewicz, S. Lalis, A. Pruszkowski, M. Koutsoubelias, T. Tajmajer, N. Grigoropoulos, M. Nati, and A. Gluhak. "Soft Actuation: Smart Home and Office with Human-in-the-Loop." *IEEE Pervasive Computing*, vol. 15, no. 1, 2016, pp. 48–56.

[Dom+16b] M. Domingo-Prieto, T. Chang, X. Vilajosana, and T. Watteyne. "Distributed PID-Based Scheduling for 6TiSCH Networks." *IEEE Communications Letters*, vol. 20, no. 5, 2016, pp. 1006–1009.

[DR10] G. Delaval and E. Rutten. "Reactive Model-Based Control of Reconfiguration in the Fractal Component-Based Model." *Proceedings of the 13th International Conference on Component-Based Software Engineering (CBSE)*. Springer, 2010, pp. 93–112.

[Dra+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: Yesterday, Today, and Tomorrow." *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[DRM13] G. Delaval, E. Rutten, and H. Marchand. "Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler." *Discrete Event Dynamic Systems*, vol. 23, no. 4, 2013, pp. 385–418.

[DST16] K. Dhondge, R. Shorey, and J. Tew. "HOLA: Heuristic and Opportunistic Link Selection Algorithm for Energy Efficiency in Industrial Internet of Things (IIoT) Systems." *Proceedings of the 8th International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, 2016, pp. 1–6.

[DTD17] D. Dragan, D. Tudose, and D. Dragomir. "Enablement of CoAP Stack on Sparrow Wireless Sensor Network." *Proceedings of the 21st International Conference on Control Systems and Computer Science (CSCS)*. IEEE, 2017, pp. 625–629.

[Dut+10] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. "From Data Center Resource Allocation to Control Theory and Back." *Proceedings of the 3rd International Conference on Cloud Computing (CLOUD)*. IEEE, 2010, pp. 410–417.

[Ecl18] Eclipse Foundation. *Eclipse Modeling Framework*. 2018. URL: `https://www.eclipse.org/modeling/emf/` (visited on 06/16/2018).

[EHB93] R. Ernst, J. Henkel, and T. Benner. "Hardware-Software Cosynthesis for Microcontrollers." *IEEE Design & Test of Computers*, vol. 10, no. 4, 1993, pp. 64–75.

[EK16] A. Engel and A. Koch. "Heterogeneous Wireless Sensor Nodes That Target the Internet of Things." *IEEE Micro*, vol. 36, no. 6, 2016, pp. 8–15.

[Eli+15] J. Eliasson, J. Delsing, H. Derhamy, Z. Salcic, and K. Wang. "Towards Industrial Internet of Things: An Efficient and Interoperable Communication Framework." *Proceedings of the 2015 International Conference on Industrial Technology (ICIT)*. IEEE, 2015, pp. 2198–2204.

[Eme96] E. A. Emerson. "Automated Temporal Reasoning About Reactive Systems." *Logics for Concurrency*. Springer, 1996, pp. 41–101.

[ERA10] M. Eisenhauer, P. Rosengren, and P. Antolin. "HYDRA: A Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence Systems." *Proceedings of the 20th Tyrrhenian Workshop on Digital Communications*. Springer, 2010, pp. 367–373.

[Esc17] C. Escoffier. *Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design*. O'Reilly Media, 2017.

[Est07] J. A. Estefan. *Survey of Model-Based Systems Engineering (MBSE) Methodologies*. INCOSE, 2007, pp. 1–12.

[Fan+14] Y. J. Fan, Y. H. Yin, L. D. Xu, Y. Zeng, and F. Wu. "IoT-Based Smart Rehabilitation System." *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, 2014, pp. 1568–1577.

[Fer+10] G. Fertitta, A. D. Stefano, G. Fiscelli, and G. C. Giaconia. "A Low Power and High Resolution Data Logger for Submarine Seismic Monitoring." *Microprocessors and Microsystems*, vol. 34, no. 2-4, 2010, pp. 63–72.

[Fil+12] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. "Reliability-Driven Dynamic Binding Via Feedback Control." *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2012, pp. 43–52.

[FIM07] S. Friedenthal, L. Izumi, and A. Meilich. "Object-Oriented Systems Engineering Method (OOSEM) Applied to Joint Force Projection (JFP), a Lockheed Martin Integrating Concept (LMIC)." *Proceedings of the INCOSE International Symposium*. 2007, pp. 1471–1491.

[Fou+12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. "A Dynamic Component Model for Cyber Physical Systems." *Proceedings of the 15th Symposium on Component Based Software Engineering (CBSE)*. ACM, 2012, pp. 135–144.

[Fur00] S. Furber. *ARM System-on-Chip Architecture*. 2nd Edition. Addison-Wesley Longman, 2000.

[Gam10] A. Gamatié. "Synchronous Programming: Overview." A. Gamatie. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010, pp. 21–39.

[Gar+13] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes." *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, 2013, pp. 89–107.

[Gar+16] M. Garriga, C. Mateos, A. Flores, A. Cechich, and A. Zunino. "RESTful Service Composition at a Glance: A Survey." *Journal of Network and Computer Applications*, vol. 60, 2016, pp. 32–53.

[Gar17] Gartner. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, up 31 Percent from 2016*. 2017. URL: `https://www.gartner.com/newsroom/id/3598917` (visited on 05/30/2018).

[GD93] R. Gupta and G. De Micheli. "Hardware-Software Cosynthesis for Digital Systems." *IEEE Design & Test of Computers*, vol. 10, no. 3, 1993, pp. 29–41.

[GEM18] GEMOC Initiative. *Arduino Modeling*. 2018. URL: `https://github.com/gemoc/arduinomodeling` (visited on 06/18/2018).

[GFA11] N. Gamez, L. Fuentes, and M. A. Aragüez. "Autonomic Computing Driven by Feature Models and Architecture in FamiWare." *Proceedings of the 5th European Conference on Software Architecture (ECSA)*. Springer, 2011, pp. 164–179.

[GG10] A. Gamatié and T. Gautier. "The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems." *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, 2010, pp. 641–657.

[Gir87] J.-Y. Girard. "Linear Logic." *Theoretical Computer Science*, vol. 50, no. 1, 1987, pp. 1–101.

[GL11] Q. Gu and P. Lago. "Guiding the Selection of Service-Oriented Software Engineering Methodologies." *Service Oriented Computing and Applications*, vol. 5, no. 4, 2011, pp. 203–223.

[GLS17] H. Garavel, F. Lang, and W. Serwe. "From LOTOS to LNT." *ModelEd, TestEd, TrustEd*. Springer, 2017, pp. 3–26.

[GTD12] K. Gama, L. Touseau, and D. Donsez. "Combining Heterogeneous Service Technologies for Building an Internet of Things Middleware." *Computer Communications*, vol. 35, no. 4, 2012, pp. 405–417.

[Gub+13] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions." *Future Generation Computer Systems*, vol. 29, no. 7, 2013, pp. 1645–1660.

[Gui+10] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. "Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services." *IEEE Transactions on Services Computing*, vol. 3, no. 3, 2010, pp. 223–235.

[Guo+15] B. Guo, Z. Wang, Z. Yu, Y. Wang, N. Y. Yen, R. Huang, and X. Zhou. "Mobile Crowd Sensing and Computing: The Review of an Emerging Human-Powered Sensing Paradigm." *ACM Computing Surveys*, vol. 48, no. 1, 2015, pp. 1–31.

[Ha+07] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. "PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems." *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, 2007, pp. 1–25.

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The Synchronous Data Flow Programming Language LUSTRE." *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1305–1320.

[Har+16] A. F. Harris III, V. Khanna, G. Tuncay, R. Want, and R. Kravets. "Bluetooth Low Energy in Dense IoT Environments." *IEEE Communications Magazine*, vol. 54, no. 12, 2016, pp. 30–36.

[Har+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, vol. 16, no. 4, 1990, pp. 403–414.

[Hea02] S. Heath. "What Is an Embedded System?" *Embedded Systems Design*. Elsevier, 2002, pp. 1–14.

[Hel+04] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. "Model Construction." *Feedback Control of Computing Systems*. John Wiley & Sons, 2004, pp. 29–64.

[Hen+09] E. Henje Blom, E. M. G. Olsson, E. Serlachius, M. Ericson, and M. Ingvar. "Heart Rate Variability Is Related to Self-Reported Physical Activity in a Healthy Adolescent Population." *European Journal of Applied Physiology*, vol. 106, no. 6, 2009, pp. 877–883.

[HLR17] M. Hussein, S. Li, and A. Radermacher. "Model-Driven Development of Adaptive IoT Systems." *Proceedings of the 4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp)*. CEUR-WS, 2017, pp. 17–23.

[Hof14] H.-P. Hoffmann. *Systems Engineering Best Practices with the Rational Solution for Systems and Software Engineering Deskbook Release 4.1*. IBM, 2014.

[HP85] D. Harel and A. Pnueli. "On the Development of Reactive Systems." *Logics and Models of Concurrent Systems*. Springer, 1985, pp. 477–498.

[HPI14] S. Hachem, A. Pathak, and V. Issarny. "Service-Oriented Middleware for Large-Scale Mobile Participatory Sensing." *Pervasive and Mobile Computing*, vol. 10, 2014, pp. 66–82.

[Hsi+14] C.-M. Hsieh, F. Samie, M. S. Srouji, M. Wang, Z. Wang, and J. Henkel. "Hardware/Software Co-Design for a Wireless Sensor Network Platform." *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2014, pp. 1–10.

[Hul+11] R. Hull, A. Nigam, P. N. Sukaviriya, R. Vaculin, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. Linehan, and S. Maradugu. "Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events." *Proceedings of the 5th International Conference on Distributed Event-Based System (DEBS)*. ACM, 2011, pp. 51–62.

[Ift+17] M. U. Iftikhar, G. S. Ramachandran, P. Bollansée, D. Weyns, and D. Hughes. "DeltaIoT: A Self-Adaptive Internet of Things Exemplar." *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 76–82.

[Iss+16] V. Issarny, G. Bouloukakis, N. Georgantas, and B. Billet. "Revisiting Service Oriented Architecture for the IoT: A Middleware Perspective." *Proceedings of the 14th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2016, pp. 3–17.

[IW14] M. U. Iftikhar and D. Weyns. "ActivFORMS: Active Formal Models for Self-Adaptation." *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 125–134.

[Izu+15] S. Izumi, K. Yamashita, M. Nakano, H. Kawaguchi, H. Kimura, K. Marumoto, T. Fuchikami, Y. Fujimori, H. Nakajima, T. Shiga, and M. Yoshimoto. "A Wearable Healthcare System with a 13.7 μA Noise Tolerant ECG Processor." *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, no. 5, 2015, pp. 733–742.

[JCL11] J. C. Jensen, D. H. Chang, and E. A. Lee. "A Model-Based Design Methodology for Cyber-Physical Systems." *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2011, pp. 1666–1671.

[Jee+10] E. Jee, S. Wang, J. K. Kim, J. Lee, O. Sokolsky, and I. Lee. "A Safety-Assured Development Approach for Real-Time Software." *Proceedings of the 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2010, pp. 133–142.

[JM12] T. T. Johnson and S. Mitra. "Parametrized Verification of Distributed Cyber-Physical Systems: An Aircraft Landing Protocol Case Study." *Proceedings of the 3rd International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2012, pp. 161–170.

[JP12] I. Jayaram and C. Purdy. "Using Constraint Graphs to Improve Embedded Systems Design." *Proceedings of the 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2012, pp. 594–597.

[JVT13] S. L. Joshi, R. A. Vatti, and R. V. Tornekar. "A Survey on ECG Signal Denoising Techniques." *Proceedings of the 2013 International Conference on Communication Systems and Network Technologies (CSNT)*. IEEE, 2013, pp. 60–64.

[Kar+09] L. Karam, I. Alkamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans. "Trends in Multicore DSP Platforms." *IEEE Signal Processing Magazine*, vol. 26, no. 6, 2009, pp. 38–49.

[KC03] J. O. Kephart and D. M. Chess. "The Vision of Autonomic Computing." *Computer*, vol. 36, no. 1, 2003, pp. 41–50.

[Kei+09] J. Keinert, M. Streubūhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. "SystemCoDesigner—an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications." *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, 2009, pp. 1–23.

[KGH14] F. Kaup, P. Gottschling, and D. Hausheer. "PowerPi: Measuring and Modeling the Power Consumption of the Raspberry Pi." *Proceedings of the 39th Annual IEEE Conference on Local Computer Networks (LCN)*. IEEE, 2014, pp. 236–243.

[KHS10] M. Kranz, P. Holleis, and A. Schmidt. "Embedded Interaction: Interacting with the Internet of Things." *IEEE Internet Computing*, vol. 14, no. 2, 2010, pp. 46–53.

[KL93] A. Kalavade and E. A. Lee. "A Hardware-Software Codesign Methodology for DSP Applications." *IEEE Design & Test of Computers*, vol. 10, no. 3, 1993, pp. 16–28.

[KMS08] T.-H. Kim, I. Maruta, and T. Sugie. "Robust PID Controller Tuning Based on the Constrained Particle Swarm Optimization." *Automatica*, vol. 44, no. 4, 2008, pp. 1104–1110.

[Kou+16] E. Kougianos, S. P. Mohanty, G. Coelho, U. Albalawi, and P. Sundaravadivel. "Design of a High-Performance System for Secure Image Communication in the Internet of Things." *IEEE Access*, vol. 4, 2016, pp. 1222–1242.

[Kyu+13] R. Kyusakov, J. Eliasson, J. Delsing, J. van Deventer, and J. Gustafsson. "Integration of Wireless Sensor and Actuator Nodes With IT Infrastructure Using Service-Oriented Architecture." *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, 2013, pp. 43–51.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam93] L. Lamport. "Hybrid Systems in TLA+." *Hybrid Systems*. Springer, 1993, pp. 77–102.

[Lee+07] H. B. Lee, J. S. Kim, Y. S. Kim, H. J. Baek, M. S. Ryu, and K. S. Park. "The Relationship Between HRV Parameters and Stressful Driving Situation in the Real Road." *Proceedings of the 6th International Special Topic Conference on Information Technology Applications in Biomedicine*. IEEE, 2007, pp. 198–200.

[Lee+12] I. Lee, O. Sokolsky, S. Chen, J. Hatcliff, E. Jee, B. Kim, A. King, M. Mullen-Fortino, S. Park, A. Roederer, and K. K. Venkatasubramanian. "Challenges and Research Directions in Medical Cyber–Physical Systems." *Proceedings of the IEEE*, vol. 100, no. 1, 2012, pp. 75–90.

[Lee08] E. A. Lee. "Cyber Physical Systems: Design Challenges." *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE, 2008, pp. 363–369.

[Lee10] E. A. Lee. "CPS Foundations." *Proceedings of the 47th Design Automation Conference (DAC)*. ACM, 2010, pp. 737–742.

[LH18] D. Li and Q. He. *ArduBlock: A Block Programming Language for Arduino*. 2018. URL: `https://github.com/taweili/ardublock` (visited on 06/19/2018).

[LL15] M. Lohstroh and E. A. Lee. "An Interface Theory for the Internet of Things." *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 2015, pp. 20–34.

[LLP12] H. Li, L. Lai, and H. V. Poor. "Multicast Routing for Decentralized Control of Cyber Physical Systems with an Application in Smart Grid." *IEEE Journal on Selected Areas in Communications*, vol. 30, no. 6, 2012, pp. 1097–1107.

[Lom76] N. R. Lomb. "Least-Squares Frequency Analysis of Unequally Spaced Data." *Astrophysics and Space Science*, vol. 39, no. 2, 1976, pp. 447–462.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. "UPPAAL in a Nutshell." *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, 1997, pp. 134–152.

[LS17] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd Edition. MIT Press, 2017.

[LSS07] J.-S. Lee, Y.-W. Su, and C.-C. Shen. "A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi." *Proceedings of the 33rd Annual Conference of the IEEE Industrial Electronics Society (IECON)*. IEEE, 2007, pp. 46–51.

[LXZ15] S. Li, L. D. Xu, and S. Zhao. "The Internet of Things: A Survey." *Information Systems Frontiers*, vol. 17, no. 2, 2015, pp. 243–259.

[Mag+14] M. Magno, C. Spagnol, L. Benini, and E. Popovici. "A Low Power Wireless Node for Contact and Contactless Heart Monitoring." *Microelectronics Journal*, vol. 45, no. 12, 2014, pp. 1656–1664.

[Mal+10] A. Malik, Z. Salcic, P. S. Roop, and A. Girault. "SystemJ: A GALS Language for System Level Design." *Computer Languages, Systems & Structures*, vol. 36, no. 4, 2010, pp. 317–344.

[Mar+00] H. Marchand, P. Bournai, M. L. Borgne, and P. L. Guernic. "Synthesis of Discrete-Event Controllers Based on the Signal Environment." *Discrete Event Dynamic Systems*, vol. 10, no. 4, 2000, pp. 325–346.

[Mas+15] B. Massot, T. Risset, G. Michelet, and E. McAdams. "A Wireless, Low-Power, Smart Sensor of Cardiac Activity for Clinical Remote Monitoring." *Proceedings of the 17th International Conference on E-Health Networking, Application & Services (HealthCom)*. IEEE, 2015, pp. 488–494.

[Mas+16] B. Massot, T. Risset, G. Michelet, and E. McAdams. "Mixed Hardware and Software Embedded Signal Processing Methods for In-Situ Analysis of Cardiac Activity." *Proceedings of the 9th International Joint Conference on Biomedical Engineering Systems and Technologies*. SciTePress, 2016, pp. 303–310.

[May14] D. Q. Mayne. "Model Predictive Control: Recent Developments and Future Promise." *Automatica*, vol. 50, no. 12, 2014, pp. 2967–2986.

[Men+11] D. Menasce, H. Gomaa, S. Malek, and J. Sousa. "SASSY: A Framework for Self-Architecting Service-Oriented Systems." *IEEE Software*, vol. 28, no. 6, 2011, pp. 78–85.

[Mhe+14] F. Mhenni, J.-Y. Choley, O. Penas, R. Plateaux, and M. Hammadi. "A SysML-Based Methodology for Mechatronic Systems Architectural Design." *Advanced Engineering Informatics*, vol. 28, no. 3, 2014, pp. 218–231.

[Mic18] Microsoft. *The Component Object Model*. 2018. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms694363(v=vs.85).aspx` (visited on 06/16/2018).

[Mit+17a] S. Mitsch, M. Gario, C. J. Budnik, M. Golm, and A. Platzer. "Formal Verification of Train Control with Air Pressure Brakes." *Proceedings of the 2nd International Conference on Reliability, Safety and Security of Railway Systems, RRSRail*. Springer, 2017, pp. 173–191.

[Mit+17b] S. Mitsch, K. Ghorbal, D. Vogelbacher, and A. Platzer. "Formal Verification of Obstacle Avoidance and Navigation of Ground Robots." *The International Journal of Robotics Research*, vol. 36, no. 12, 2017, pp. 1312–1340.

[Mod18] Modkit LLC. *Modkit*. 2018. URL: `https://www.modkit.com/` (visited on 06/19/2018).

[Mor+09] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. "Models@run.Time to Support Dynamic Adaptation." *Computer*, vol. 42, no. 10, 2009, pp. 44–51.

[MR01] F. Maraninchi and Y. Rémond. "Argos: An Automaton-Based Synchronous Language." *Computer Languages*, vol. 27, no. 1, 2001, pp. 61–92.

[MR98] F. Maraninchi and Y. Rémond. "Mode-Automata: About Modes and States for Reactive Systems." *Proceedings of the 7th European Symposium on Programming (ESOP)*. Springer, 1998, pp. 185–199.

[MS04] E. M. Maximilien and M. P. Singh. "A Framework and Ontology for Dynamic Web Services Selection." *IEEE Internet Computing*, vol. 8, no. 5, 2004, pp. 84–93.

[MVF00] J. Muttersbach, T. Villiger, and W. Fichtner. "Practical Design of Globally Asynchronous Locally Synchronous Systems." *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*. IEEE, 2000, pp. 52–59.

[MY11] A. Malinowski and H. Yu. "Comparison of Embedded System Design for Industrial Applications." *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, 2011, pp. 244–254.

[Nai+10] G. Nain, F. Fouquet, B. Morin, O. Barais, and J.-M. Jezequel. "Integrating IoT and IoS with a Component-Based Approach." *Proceedings of the 36th Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2010, pp. 191–198.

[Nat17] National Instruments. *Best Practices in PCB Design: Routing*. 2017. URL: http://www.ni.com/tutorial/6880/en/ (visited on 06/27/2018).

[NC03] A. Nigam and N. S. Caswell. "Business Artifacts: An Approach to Operational Specification." *IBM Systems Journal*, vol. 42, no. 3, 2003, pp. 428–445.

[New14] C. Newcombe. "Why Amazon Chose TLA+." *Proceedings of the 4th International Conference on Abstract State Machines, B, TLA, VDM, and Z (ABZ)*. Springer, 2014, pp. 25–39.

[NM14] R. Negenborn and J. Maestre. "Distributed Model Predictive Control: An Overview and Roadmap of Future Research Opportunities." *IEEE Control Systems*, vol. 34, no. 4, 2014, pp. 87–97.

[Nod18] Node-RED. *Node-RED: Flow-Based Programming for the Internet of Things*. 2018. URL: https://nodered.org/ (visited on 06/19/2018).

[NSB10] D. Nunan, G. R. H. Sandercock, and D. A. Brodie. "A Quantitative Systematic Review of Normal Values for Short-Term Heart Rate Variability in Healthy Adults." *Pacing and Clinical Electrophysiology*, vol. 33, no. 11, 2010, pp. 1407–1417.

[OAA16] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman. "FairAccess: A New Blockchain Based Access Control Framework for the Internet of Things: Fairaccess: A New Access Control Framework for Iot." *Security and Communication Networks*, vol. 9, no. 18, 2016, pp. 5943–5964.

[Obj18a] Object Management Group. *Business Process Model and Notation*. 2018. URL: http://www.bpmn.org/ (visited on 06/16/2018).

[Obj18b] Object Management Group. *CORBA Component Model*. 2018. URL: https://www.omg.org/spec/CCM/About-CCM/ (visited on 06/16/2018).

[Obj18c] Object Management Group. *Model Driven Architecture*. 2018. URL: https://www.omg.org/mda/ (visited on 06/16/2018).

[Obj18d] Object Management Group. *Systems Modeling Language*. 2018. URL: http://www.omgsysml.org/ (visited on 06/16/2018).

[Obj18e] Object Management Group. *Unified Modeling Language*. 2018. URL: http://www.uml.org/ (visited on 06/16/2018).

[Oct18] Octoblu. *Octoblu: Integration of Things*. 2018. URL: https://octoblu.github.io/ (visited on 06/19/2018).

[Oga10] K. Ogata. *Modern Control Engineering*. 5th Edition. Prentice Hall, 2010.

[OSG07] OSGi Alliance. *About the OSGi Service Platform: Technical Whitepaper*. OSGi, 2007, pp. 1–20.

[PA16] E. Pencheva and I. Atanasov. "Engineering of Web Services for Internet of Things Applications." *Information Systems Frontiers*, vol. 18, no. 2, 2016, pp. 277–292.

[Pat+12] T. Patikirikorala, L. Wang, A. Colman, and J. Han. "Hammerstein–Wiener Nonlinear Model Based Predictive Control for Relative QoS Performance and Resource Management of Software Systems." *Control Engineering Practice*, vol. 20, no. 1, 2012, pp. 49–61.

[Pel03] C. Peltz. "Web Services Orchestration and Choreography." *Computer*, vol. 36, no. 10, 2003, pp. 46–52.

[Pen+12] X. Peng, B. Chen, Y. Yu, and W. Zhao. "Self-Tuning of Software Systems Through Dynamic Quality Tradeoff and Value-Based Feedback Control Loop." *Journal of Systems and Software*, vol. 85, no. 12, 2012, pp. 2707–2719.

[Per+14a] C. Perera, P. P. Jayaraman, A. Zaslavsky, D. Georgakopoulos, and P. Christen. "MOSDEN: An Internet of Things Middleware for Resource Constrained Mobile Devices." *Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2014, pp. 1053–1062.

[Per+14b] C. Perera, C. H. Liu, S. Jayawardena, and Min Chen. "A Survey on Internet of Things From Industrial Market Perspective." *IEEE Access*, vol. 2, 2014, pp. 1660–1679.

[PFW12] P. Papapanagiotou, J. Fleuriot, and S. Wilson. "Diagrammatically-Driven Formal Verification of Web-Services Composition." *Proceedings of the 7th International Conference on Theory and Application of Diagrams (Diagrams)*. Springer, 2012, pp. 241–255.

[PH06] M. P. Papazoglou and W.-J. V. D. Heuvel. "Service-Oriented Design and Development Methodology." *International Journal of Web Engineering and Technology*, vol. 2, no. 4, 2006, p. 412.

[PL03] R. Perrey and M. Lycett. "Service-Oriented Architecture." *Proceedings of the Symposium on Applications and the Internet Workshops (SAINT Workshops)*. IEEE, 2003, pp. 116–119.

[Pla10] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010.

[Pla18] A. Platzer. *KeYmaera X: An aXiomatic Tactical Theorem Prover for Hybrid Systems*. 2018. URL: `http://www.ls.cs.cmu.edu/KeYmaeraX/` (visited on 06/21/2016).

[PNC15] C. L. Phillips, H. T. Nagle, and A. Chakrabortty. *Digital Control System Analysis & Design*. 4th Edition. Pearson, 2015.

[PND14] T. A. M. Phan, J. K. Nurminen, and M. Di Francesco. "Cloud Databases for Internet-of-Things Data." *Proceedings of the 2014 International Conference on Internet of Things (iThings), Green Computing and Communications (GreenCom), and Cyber, Physical and Social Computing (CPSCom)*. IEEE, 2014, pp. 117–124.

[Pnu86] A. Pnueli. "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends." *Current Trends in Concurrency*. Springer, 1986, pp. 510–584.

[Poi+98] A. Poigné, M. Morley, O. Maffeïs, L. Holenderski, and R. Budde. "The Synchronous Approach to Designing Reactive Systems." *Formal Methods in System Design*, vol. 12, no. 2, 1998, pp. 163–187.

[PR89a] A. Pnueli and R. Rosner. "On the Synthesis of an Asynchronous Reactive Module." *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 372. Springer, 1989, pp. 652–671.

[PR89b] W. H. Press and G. B. Rybicki. "Fast Algorithm for Spectral Analysis of Unevenly Sampled Data." *The Astrophysical Journal*, vol. 338, 1989, pp. 277–280.

[PR90] A. Pneuli and R. Rosner. "Distributed Reactive Systems Are Hard to Synthesize." *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1990, pp. 746–757.

[PT85] J. Pan and W. J. Tompkins. "A Real-Time QRS Detection Algorithm." *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, 1985, pp. 230–236.

[Pto14] C. Ptolemaeus, ed. *System Design, Modeling, and Simulation Using Ptolemy II*. 2014.

[PvdH07] M. P. Papazoglou and W.-J. van den Heuvel. "Service Oriented Architectures: Approaches, Technologies and Research Issues." *The VLDB Journal*, vol. 16, no. 3, 2007, pp. 389–415.

[PZL08] C. Pautasso, O. Zimmermann, and F. Leymann. “RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision.” *Proceedings of the 17th International Conference on World Wide Web (WWW)*. ACM, 2008, p. 805.

[Raj+10] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. “Cyber-Physical Systems: The Next Computing Revolution.” *Proceedings of the 47th Design Automation Conference (DAC)*. ACM, 2010, pp. 731–736.

[Rat03] Rational Software Corporation. *Rational Unified Process for Systems Engineering*. IBM, 2003.

[Ray17] P. P. Ray. “A Survey on Visual Programming Languages in Internet of Things.” *Scientific Programming*, vol. 2017, 2017, pp. 1–6.

[Raz+13] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. “Lithe: Lightweight Secure CoAP for the Internet of Things.” *IEEE Sensors Journal*, vol. 13, no. 10, 2013, pp. 3711–3720.

[Raz+16] A. Raza, A. A. Ikram, A. Amin, and A. J. Ikram. “A Review of Low Cost and Power Efficient Development Boards for IoT Applications.” *Proceedings of the 2016 Future Technologies Conference (FTC)*. IEEE, 2016, pp. 786–790.

[RGS13] O. Rafique, M. Gesell, and K. Schneider. “Generating Hardware Specific Code at Different Abstraction Levels Using Averest.” *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2013, pp. 90–92.

[RKS17] U. Raza, P. Kulkarni, and M. Sooriyabandara. “Low Power Wide Area Networks: An Overview.” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, 2017, pp. 855–873.

[RL15] M. Rana and L. Li. “Microgrid State Estimation and Control for Smart Grid and Internet of Things Communication Network.” *Electronics Letters*, vol. 51, no. 2, 2015, pp. 149–151.

[Rod15] A. Rodrigues da Silva. “Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model.” *Computer Languages, Systems & Structures*, vol. 43, 2015, pp. 139–155.

[Rom+13] D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen. “The DigiHome Service-Oriented Platform.” *Software: Practice and Experience*, vol. 43, no. 10, 2013, pp. 1205–1218.

[Rou+09] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. “MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments.” *Software Engineering for Self-Adaptive Systems*. Springer, 2009, pp. 164–182.

[RS04] J. Rao and X. Su. “Toward the Composition of Semantic Web Services.” *Proceedings of the 2n 2nd International Workshop on Grid and Cooperative Computing (GCC)*. Springer, 2004, pp. 760–767.

[RW89] P. J. G. Ramadge and W. M. Wonham. “The Control of Discrete Event Systems.” *Proceedings of the IEEE*, vol. 77, no. 1, 1989, pp. 81–98.

[SB17] K. Schneider and J. Brandt. “Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems.” *Handbook of Hardware/Software Codesign*. Springer, 2017, pp. 29–58.

[Sca09] R. Scattolini. “Architectures for Distributed and Hierarchical Model Predictive Control – a Review.” *Journal of Process Control*, vol. 19, no. 5, 2009, pp. 723–731.

[Sca82] J. D. Scargle. “Studies in Astronomical Time Series Analysis II - Statistical Aspects of Spectral Analysis of Unevenly Spaced Data.” *The Astrophysical Journal*, vol. 263, 1982, pp. 835–853.

[Sch06] D. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering.” *Computer*, vol. 39, no. 2, 2006, pp. 25–31.

[Sch10a] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.

[Sch10b] P. R. Schaumont. “System on Chip.” *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010, pp. 3–31.

[Sch10c] P. R. Schaumont. “The Nature of Hardware and Software.” *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010, pp. 3–31.

[SCL16] G. Schermann, J. Cito, and P. Leitner. “All the Services Large and Micro: Revisiting Industrial Practice in Services Computing.” *Proceedings of the 14th International Conference on Service-Oriented Computing Workshops (ICSOC Workshops)*. Springer, 2016, pp. 36–47.

[Seg06] R. Segala. “Probability and Nondeterminism in Operational Models of Concurrency.” *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR)*. Springer, 2006, pp. 64–78.

[Sey+16] N. Seydoux, K. Drira, N. Hernandez, and T. Monteil. “IoT-O, a Core-Domain IoT Ontology to Represent Connected Devices Networks.” *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*. Springer, 2016, pp. 561–576.

[SHA17] R. Seiger, S. Herrmann, and U. ABmann. “Self-Healing for Distributed Workflows in the Internet of Things.” *Proceedings of the 2017 International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 72–79.

[SHG12] S. Sridhar, A. Hahn, and M. Govindarasu. “Cyber-Physical System Security for the Electric Power Grid.” *Proceedings of the IEEE*, vol. 100, no. 1, 2012, pp. 210–224.

[Shi+16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. “Edge Computing: Vision and Challenges.” *IEEE Internet of Things Journal*, vol. 3, no. 5, 2016, pp. 637–646.

[Sic+15] S. Sicari, A. Rizzardi, L. Grieco, and A. Coen-Porisini. “Security, Privacy and Trust in Internet of Things: The Road Ahead.” *Computer Networks*, vol. 76, 2015, pp. 146–164.

[Sil+15] L. Silva, H. Almeida, A. Perkusich, and M. Perkusich. “A Model-Based Approach to Support Validation of Medical Cyber-Physical Systems.” *Sensors*, vol. 15, no. 11, 2015, pp. 27625–27670.

[Sim15] P. A. Simpson. “Embedded Design.” *FPGA Design*. Springer, 2015, pp. 157–178.

[Sko03] S. Skogestad. “Simple Analytic Rules for Model Reduction and PID Controller Tuning.” *Journal of Process Control*, vol. 14, no. 4, 2003, pp. 291–309.

[SLR17] A. N. Sylla, M. Louvel, and E. Rutten. “Design Framework for Reliable and Environment Aware Management of Smart Environment Devices.” *Journal of Internet Services and Applications*, vol. 8, no. 1, 2017, pp. 1–22.

[SMA16] M. Salatino, M. D. Maio, and E. Aliverti. *Mastering JBoss Drools 6*. Packt Publishing, 2016.

[Sol+13] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou, and C.-H. Lung. “Smart Home: Integrating Internet of Things with Web Services and Cloud Computing.” *Proceedings of the 5th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2013, pp. 317–320.

[Spi+09] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. M. S. de Souza, and V. Trifa. "SOA-Based Integration of the Internet of Things in Enterprise Services." *Proceedings of the 7th International Conference on Web Services (ICWS)*. IEEE, 2009, pp. 968–975.

[Sta+13] M. J. Stanovich, I. Leonard, K. Sanjeev, M. Steurer, T. P. Roth, S. Jackson, and M. Bruce. "Development of a Smart-Grid Cyber-Physical Systems Testbed." *Proceedings of the 4th Innovative Smart Grid Technologies Conference (ISGT)*. IEEE, 2013, pp. 1–6.

[Sta14] J. A. Stankovic. "Research Directions for the Internet of Things." *IEEE Internet of Things Journal*, vol. 1, no. 1, 2014, pp. 3–9.

[Ste93] R. Stengel. "Toward Intelligent Flight Control." *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 6, 1993, pp. 1699–1717.

[SV13] A. Schäfer and J. Vagedes. "How Accurate Is Pulse Rate Variability as an Estimate of Heart Rate Variability?" *International Journal of Cardiology*, vol. 166, no. 1, 2013, pp. 15–29.

[Syl+17] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval. "Design Framework for Reliable Multiple Autonomic Loops in Smart Environments." *Proceedings of the 2017 International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2017, pp. 131–142.

[SZS15] D. S. Sousa Nunes, P. Zhang, and J. Sa Silva. "A Survey on Human-in-the-Loop Applications Towards an Internet of All." *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, 2015, pp. 944–965.

[TA05] C. Townsend and S. Arms. "Wireless Sensor Networks." *Sensor Technology Handbook*. Elsevier, 2005, pp. 575–589.

[Tam+13] G. Tamura, N. M. Villegas, H. A. Müller, L. Duchien, and L. Seinturier. "Improving Context-Awareness in Self-Adaptation Using the DYNAMICO Reference Model." *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2013, pp. 153–162.

[Tas96] Task Force of the European Society of Cardiology the North American Society of Pacing Electrophysiology. "Heart Rate Variability : Standards of Measurement, Physiological Interpretation, and Clinical Use." *Circulation*, vol. 93, no. 5, 1996, pp. 1043–1065.

[TC16] K. Thramboulidis and F. Christoulakis. "UML4IoT—A UML-Based Approach to Exploit IoT in Cyber-Physical Manufacturing Systems." *Computers in Industry*, vol. 82, 2016, pp. 259–272.

[Tei+11] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas. "Service Oriented Middleware for the Internet of Things: A Perspective." *Proceedings of the 4th European Conference ServiceWave*. Springer, 2011, pp. 220–229.

[Ter+17] M. Terdjimi, L. Médini, M. Mrissa, and M. Maleshkova. "Multi-Purpose Adaptation in the Web of Things." *Proceedings of the 10th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT)*. Springer, 2017, pp. 213–226.

[TGD14] X. Tang, P.-E. Gaillardon, and G. De Micheli. "A High-Performance Low-Power Near-*Vt* RRAM-Based FPGA." *Proceedings of the 2014 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2014, pp. 207–214.

[Thi+17] H. Thieblemont, F. Haghighat, R. Ooka, and A. Moreau. "Predictive Control Strategies Based on Weather Forecast in Buildings with Energy Storage System: A Review of the State-of-the Art." *Energy and Buildings*, vol. 153, 2017, pp. 485–500.

[Tuo+17] J. Tuominen, E. Lehtonen, M. J. Tadi, J. Koskinen, M. Pankaala, and T. Koivisto. "A Miniaturized Low Power Biomedical Sensor Node for Clinical Research and Long Term Monitoring of Cardiovascular Signals." *Proceedings of the 2017 International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2017, pp. 1–4.

[TVL17] D. M. Tung, N. Van Toan, and J.-G. Lee. "Exploring the Current Consumption of an Intel Edison Module for IoT Applications." *Proceedings of the 2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. IEEE, 2017, pp. 1–6.

[UDD01] UDDI. *UDDI Executive White Paper*. 2001. URL: `http://www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf` (visited on 06/11/2018).

[vAll18] P. van Allen. *NETLabTK*. 2018. URL: `http://www.netlabtoolkit.org/` (visited on 06/19/2018).

[Van+01] W. M. P. Van Der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. "Proclets: A Framework for Lightweight Interacting Workflow Processes." *International Journal of Cooperative Information Systems*, vol. 10, no. 4, 2001, pp. 443–481.

[Vie+03] M. Vieira, C. Coelho, D. da Silva, and J. da Mata. "Survey on Wireless Sensor Network Devices." *Proceedings of the 9th Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2003, pp. 537–544.

[Vil+11] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. "A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems." *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2011, pp. 80–89.

[Vil+13] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas. "DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems." *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 265–293.

[Vin+10] H. Vincent, V. Issarny, N. Georgantas, E. Francesquini, A. Goldman, and F. Kon. "CHOReOS: Scaling Choreographies for the Internet of the Future." *Posters and Demos Track of the 11th International Middleware Conference*. ACM, 2010, pp. 1–3.

[Vou+14] O. Voutyras, P. Bourelos, D. Kyriazis, and T. Varvarigou. "An Architecture Supporting Knowledge Flow in Social Internet of Things Systems." *Proceedings of the 310th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2014, pp. 100–105.

[Wan+10] E. K. Wang, Y. Ye, X. Xu, S. M. Yiu, L. C. K. Hui, and K. P. Chow. "Security Issues and Challenges for Cyber Physical System." *Proceedings of the 6th International Conference on Green Computing and Communications (GreenCom) & 3rd International Conference on Cyber, Physical and Social Computing (CPSCom)*. IEEE, 2010, pp. 733–738.

[Wei+96] M. Weiser, B. Welch, A. Demers, and S. Shenker. "Scheduling for Reduced CPU Energy." *Mobile Computing*. Springer, 1996, pp. 449–471.

[WK07] F. Wotawa and W. Krenn. "Knowledge Extraction from C-Code." Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES). IEEE, 2007, pp. 49–60.

[WMA10] D. Weyns, S. Malek, and J. Andersson. "FORMS: A Formal Reference Model for Self-Adaptation." *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*. ACM, 2010, pp. 205–214.

[Wol03] W. H. Wolf. "A Decade of Hardware/Software Codesign." *Computer*, vol. 36, no. 4, 2003, pp. 38–43.

[Wol94] W. H. Wolf. “Hardware-Software Co-Design of Embedded Systems.” *Proceedings of the IEEE*, vol. 82, no. 7, 1994, pp. 967–989.

[WR97] J. L. Wynekoop and N. L. Russo. “Studying System Development Methodologies: An Examination of Research Methods.” *Information Systems Journal*, vol. 7, no. 1, 1997, pp. 47–65.

[WW83] B. B. Winter and J. G. Webster. “Driven-Right-Leg Circuit Design.” *IEEE Transactions on Biomedical Engineering*, vol. BME-30, no. 1, 1983, pp. 62–66.

[Wyl18] Wyliodrin. *Wyliodrin: Your Next IoT*. 2018. URL: `https://www.wyliodrin.com/` (visited on 06/19/2018).

[Yil+12] Yilin Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, Heejo Lee, A. Perrig, and B. Sinopoli. “Cyber-Physical Security of a Smart Grid Infrastructure.” *Proceedings of the IEEE*, vol. 100, no. 1, 2012, pp. 195–209.

[Zan+14] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. “Internet of Things for Smart Cities.” *IEEE Internet of Things Journal*, vol. 1, no. 1, 2014, pp. 22–32.

[Zha+13] M. Zhao, G. Privat, É. Rutten, and H. Alla. “Discrete Control for the Internet of Things and Smart Environments.” *Proceedings of the 8th International Workshop on Feedback Computing*. USENIX, 2013, pp. 1–6.

[Zha+14] M. Zhao, G. Privat, E. Rutten, and H. Alla. “Discrete Control for Smart Environments Through a Generic Finite-State-Models-Based Infrastructure.” *Proceedings of the 5th International Joint Conference of Ambient Intelligence (AmI)*. Ed. by E. Aarts. Springer, 2014, pp. 174–190.

[Zhu+17] X. Zhu, Y. Badr, J. Pacheco, and S. Hariri. “Autonomic Identity Framework for the Internet of Things.” *Proceedings of the 2017 International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2017, pp. 69–79.

[Zim80] H. Zimmermann. “OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection.” *IEEE Transactions on Communications*, vol. 28, no. 4, 1980, pp. 425–432.

[ZKG04] O. Zimmermann, P. Krogdahl, and C. Gee. *Elements of Service-Oriented Analysis and Design: An Interdisciplinary Modeling Approach for SOA Projects*. IBM, 2004, pp. 1–18.

[ZRF94] L. Zhao, S. Reisman, and T. Findley. “Respiration Derived from the Electrocardiogram During Heart Rate Variability Studies.” *Proceedings of 16th International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 1994, pp. 123–124.

[ZW06] L. Zhang and Z. Wang. “Integration of RFID into Wireless Sensor Networks: Architectures, Opportunities and Challenging Problems.” *Proceedings of the 5th International Conference on Grid and Cooperative Computing Workshops (GCC)*. IEEE, 2006, pp. 463–469.

Glossary

ADC Analog-to-Digital Converter . . . 5, 53, 54, 64, 68, 77, 84
AFE Analog Front-End . . . 5, 64, 77, 80, 84–86, 93
AMQP Advanced Message Queuing Protocol . . . 123
API Application Programming Interface . . . 50, 124, 128
ASIC Application-Specific Integrated Circuit . . . 15, 117

BAN Bersonal Area Network . . . 6
BLE Bluetooth Low Energy . . . 4, 6, 10, 50, 64, 74, 77, 84, 86, 88–90, 95, 147, 154, 160, 161
BPEL Business Process Execution Language . . . 40
BPEL4WS Business Process Execution Language for Web Services . . . 17
BPMN Business Process Model and Notation . . . 17, 20, 21

CASE Computer-Aided Software Engineering . . . 20
CBSE Component-Based Software Engineering . . . 22, 23, 46
CEP Complex Event Processing . . . 117, 118, 122, 128–130, 144
CoAP Constrained Application Protocol . . . 7, 19, 20, 154
COM Component Object Model . . . 22
CORBA Common Object Request Broker Architecture . . . 22
CPS Cyber-Physical Systems . . . 14, 15, 17, 23–28, 46, 120

DAC Digital-to-Analog Converter . . . 53, 55
DCS Discrete Controller Synthesis . . . 35, 44, 45, 109, 111, 119, 123, 147
DDS Data Distribution Service . . . 124, 125, 130–133, 135, 136, 144, 148, 166
DFB Digital Filter Block . . . 64, 76, 77, 79–82, 84, 86, 147
DIY Do-It-Yourself . . . 4, 34, 35, 52
DMA Direct Memory Access . . . 53, 59, 64, 65, 67, 81, 82, 146
DP Domain Participant . . . 124
DPWS Devices Profile for Web Services . . . 19
DSL Domain-Specific Language . . . 19, 144
DSP Digital Signal Processor . . . 5, 32–34, 79

ECA Event-Condition-Action . . . 45, 113
ECG electrocardiogram . . . 62–64, 66, 74, 77, 79–81, 84, 85, 87, 88, 90–93, 95, 146
EMF Eclipse Modeling Framework . . . 21
ESB Enterprise Service Bus . . . 18, 19

FFT Fast Fourier Transform . . . 83
FPGA Field-Programmable Gate Array . . . 31–34
FSM Finite-State Machine . . . 41

GALS Globally Asynchronous Locally Synchronous . . . 116, 117, 119
GDS Global Data Space . . . 124
GUI Graphical User Interface . . . 123, 126, 130, 131, 136, 137, 139, 141, 144

HDL Hardware Description Language . . . 15, 31, 33, 44, 68, 82
HR Heart Rate . . . 63–67, 73–77, 79, 82–85, 88, 93–95, 101, 107, 109, 146
HRV Heart Rate Variability . . . 63–65, 73–77, 82–85, 94, 95, 101, 107, 146

HTML Hypertext Markup Language . . . 126
HTTP Hypertext Transfer Protocol . . . 7, 18, 23, 125, 126, 144, 154
HTTPS Hypertext Transfer Protocol Secure . . . 144

IC Integrated Circuit . . . 5, 13–15, 31–35, 47, 51–53, 56, 59, 61, 62, 64, 68, 70, 76, 77, 93, 95, 146
IDE Integrated Development Environment . . . 34, 67, 68, 81
IDL Interface Definition Language . . . 124
IP Internet Protocol . . . 4, 6, 7

JNI Java Native Interface . . . 125
JSON JavaScript Object Notation . . . 18, 59, 130, 141, 144, 149, 166
JSON-LD JSON for Linked Data . . . 149
JVM Java Virtual Machine . . . 41, 125, 166

LTS Labeled Transition System . . . 10, 11, 43–45, 62, 85, 86, 106–109, 111, 119, 123, 130, 147, 148, 150

MARTE Modeling and Analysis of Real-Time and Embedded systems . . . 21
MDA Model Driven Architecture . . . 21
MDE Model Driven Engineering . . . 14, 20–23, 27, 28, 41, 42, 44, 46, 71
MIAC Model Identification Adaptive Control . . . 36
MQTT Message Queuing Telemetry Transport . . . 7, 19, 23, 123–125, 154, 166
MRAC Model Reference Adaptive Control . . . 36

NFC Near Field Communication . . . 4, 74
NFP Non-Functional Property . . . 9, 10, 62, 98, 102–104, 112, 116

OMG Object Management Group . . . 21
OOSEM Object-Oriented Systems Engineering Method . . . 29
OSGi Open Service Gateway initiative . . . 41
OTA Over-the-Air . . . 95, 150
OWL Web Ontology Language . . . 149

PAN Personal Area Network . . . 6
PCB Printed Circuit Board . . . 14, 31, 33, 34, 51, 52, 62, 78, 85, 93, 171
PID Proportional Integral Derivative . . . 36, 37
PoC Proof-of-Concept . . . 10, 121, 123, 137, 139, 141, 142, 144
PPG Photoplethysmography . . . 101, 108, 109, 115, 116, 138
PSD Power Spectral Density . . . 76, 83

QoS Quality of Service 9, 10, 14, 15, 18, 19, 24, 30, 37, 38, 44, 45, 47, 51, 54, 63, 70, 95, 98–101, 104, 105, 109, 116, 122, 124, 130, 134, 135, 137–141, 144, 147

RAM Random Access Memory . . . 4, 19, 50, 56
RDF Resource Description Framework . . . 149
REST Representational State Transfer . . . 18
RLD right-leg drive . . . 77, 78
RTOS Real-Time Operating System . . . 33
RUP SE Rational Unified Process for Systems Engineering . . . 29
RWF Respiration Waveform . . . 63, 65, 75, 77, 84, 85, 93, 94, 101, 146

SCA Service Component Architecture . . . 22
SFTP Secure Shell (SSH) File Transfer Protocol . . . 134
SLA Service Level Agreement . . . 98, 102, 105
SLO Service Level Objective . . . 105
SOA Service-Oriented Architecture . . . 9, 14, 16–20, 27–29, 31, 41, 46, 50, 51, 69, 146
SOAP Simple Object Access Protocol . . . 17–19
SoC System-on-Chip . . . 7, 15, 31, 33, 34, 47, 50, 52, 53, 61, 62, 64, 65, 70, 75–78, 85, 90, 95, 117
SPI Serial Peripheral Interface . . . 64, 65, 77, 85
SPL Synchronous Programming Language . . . 11, 16, 43–45, 106, 107, 117–119, 123, 147, 150

SQL Structured Query Language 112, 130, 144
SSH Secure Shell 134, 143
SysML Systems Modeling Language 21, 29, 46

TCP Transmission Control Protocol 7, 19, 20, 64

UART Universal Asynchronous Receiver Transmitter 65
UDDI Universal Description, Discovery, and Integration 17
UDP User Datagram Protocol 7, 19
UML Unified Modeling Language 20–22, 29, 46
URI Uniform Resource Identifier 7, 154

WSCI Web Service Choreography Interface 18
WSDL Web Services Description Language 17–19, 50

XMI XML Metadata Interchange 21
XML Extensible Markup Language 17, 18, 22, 129
XMPP Extensible Messaging and Presence Protocol 123, 125

Licenses and attributions:
The iconography used in Figure 4.1 provided by Font Awesome 4.7.0, available at `https://fontawesome.com`.

The serif font used in this manuscript is Adobe Utopia:

> Copyright ©1989 Adobe Systems Incorporated
> Utopia ®
> Utopia is a registered trademark of Adobe Systems Incorporated

The sans-serif font used in this manuscript is Fira Sans, distributed under the SIL Open Font License 1.1.

The monospaced font used in this manuscript is `Source Code Pro`, distributed under the SIL Open Font License 1.1.

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : GATOUILLAT
(avec précision du nom de jeune fille, le cas échéant)

DATE de SOUTENANCE : 20 décembre 2018

Prénoms : Arthur Yves Etienne

TITRE :

Towards smart services with reusable and adaptable connected objects: An application to wearable non-invasive biomedical sensors.

NATURE : Doctorat

Numéro d'ordre : 2018LYSEI123

Ecole doctorale : ED 512 – InfoMaths

Spécialité : Informatique

RESUME :

La prolifération des objets communicants fixes et mobiles soulève la question de leur intégration dans les environnements quotidiens, par exemple dans le cadre de la e-santé ou de la domotique. Les principaux défis soulevés relèvent de l'interconnexion et de la gestion de la masse de donnée produite par ces objets intelligents. Notre premier objectif est d'adopter une démarche des couches basses vers les couches hautes pour faciliter l'intégration de ces objets à des services intelligents. Afin de développer celle-ci, il est nécessaire de d'étudier le processus de conception des objets intelligents indépendamment de considérations matérielles et logicielles, au travers de la considération de leur propriétés cyber-physiques. Pour mener à bien la réalisation de services intelligents à partir d'objets connectés, les deux axes de recherche suivant seront développés : la définition d'une méthode de conception orientée service pour les objets connectés intégrant une dimension formelle ainsi de valider le comportement de ceux-ci, l'auto-adaptation intelli- gente dans un contexte évolutif permettant aux objets de raisonner sur eux même au travers d'un langage déclaratif pour spécifier les stratégies d'adaptation. La validation de ces contributions s'effectuera par le biais du développement et de l'expérimentation à grandeur nature d'un service de diagnostic médical continu basé sur la collecte de données médicales en masse par des réseaux non-intrusifs de capteurs biomédicaux portables sur le corps humain.

MOTS-CLÉS : Internet des objets ; Objets intelligents ; Méthode de conception ; Auto-adaptation

Laboratoire (s) de recherche : LIRIS

Directeur de thèse: Youakim BADR

Président de jury :

Composition du jury :
SONG, Ye-Qiong, Professeur des Universités, Université de Lorraine, Rapporteur
RIVENQ, Atika, Professeure des Universités, UPHF, Rapporteure
MALENFANT, Jacques, Professeur des Universités, Sorbonne Université, Examinateur
GUESSOUM, Zahia, Maître de Conférences HDR, Sorbonne Université, Examinatrice
BADR, Youakim, Maître de Conférences HDR, INSA-Lyon, Directeur de thèse
MASSOT, Bertrand, Maître de Conférences, INSA-Lyon, Co-encadrant de thèse