TÉCNICO LISBOA

A Modular Software Architecture to Cope With Automation Hardware Heterogeneity

Pedro Gonçalo Pinto Domingues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Doutor Paulo Jorge Fernandes Carreira

Examination Committee

Chairperson:	Prof. Doutor José Manuel da Costa Alves Marques
Supervisor:	Prof. Doutor Paulo Jorge Fernandes Carreira
Member of the Committee:	Prof. Doutor Renato Jorge Caleira Nunes

June 2015

"A hundred times every day I remind myself that my inner and outer life depend on the labors of other men, living and dead, and that I must exert myself in order to give in the same measure as I have received and am still receiving."

— Albert Einstein

Acknowledgments

First of all I would like to thank the European Commission, namely MIT Portugal, for inviting us to the outstanding project of SmartCampus.

I would like to show my gratitude towards Prof. Dr. Wolfgang Kastner, for the kindness of accepting our invitation in order to cooperate with us for submitting a scientific publication.

I would also like to thank my adviser Prof. Dr. Paulo Carreira for his extreme support during this work, way ahead of the expectations, but above all for changing the way I perceive software development, particularly in Building Automation. Turning this thesis into a growing experience for me as person.

To my friends who have been there all the time during these years.

Finally, to my family, father, mother and my brother, (and my dog) for always being there supporting me, even when I failed with my presence. Every contribution I may make in this world is, thereafter, your contribution. Thank you.

Resumo

Nos últimos anos, a área de Automação de Edifícios tem adquirido cada vez mais atenção graças à sua capacidade para aplicar medidas de eficiência energética. Infelizmente as tecnologias de automação são caracterizadas pela sua elevada heterogeneidade. Não é possível integrar de forma eficaz soluções de fabricantes diferentes. Consequentemente os clientes são forçados a optar por uma única gama de produtos, ficando presos a um determinado vendedor, traduzindo-se em custos de instalação e manutenção tipicamente elevados. Existe também uma escassez de referências bibliográficas actualizadas relativamente ao tópico da Automação de Edifícios. A existência de inconsistências e desacordos de determinadas definições é um problema recorrente na literatura.

Este trabalho começa por sistematizar conceitos base de automação à luz da literatura internacional estabelecida como norma. Aferimos também quais as funcionalidades mínimas que um sistema de Automação de Edifícios deve providenciar. Demonstramos que não existe uma única tecnologia capaz de cumprir todos os requisitos aferidos, obrigando a que os fabricantes desenvolvam extensões personalizadas para cobrir estas carências de funcionalidade, criando assim mais barreiras para a interoperabilidade entre as diferentes tecnologias.

Para resolver esta situação, considerada uma das maiores barreiras ao crescimento da área da Automação Industrial, desenvolvemos uma plataforma capaz de integrar os diferentes produtos do mercado. A nossa solução é validada contra outras soluções semelhantes já existentes, e por fim é colocada num ambiente de produção, no âmbito de um projecto Europeu, onde não só obtém uma performance superior à das soluções já existentes, como apresenta um elevado potencial para a aplicação de técnicas de eficiência energética.

Palavras-chave: Automação de Edifícios, Domótica, Fieldbus, Sistemas, Protocolos, Modelos de Informação, Standards, Integração, Framework, Tecnologicamente-Agnóstico, Abstracção de Hardware, Ontologia

Abstract

Following the widespread adoption of building management systems, driven mostly by building energy efficiency concerns, Building Automation has earned a great deal of attention. However, it is well known that Building Automation technology is characterized by its heterogeneity in that, solutions from different manufacturers cannot be easily integrated with each other, leading to costumer lock-in situations that increase installation and maintenance costs. However, given the popularity of the subject, one startling aspect of Building Automation is the scarcity of authoritative literature references regarding the topic—definition inconsistency and disagreement are recurring issues. This results in communication difficulties between system developers further hindering the efforts to overcome heterogeneity.

This work comprehensively defines basic automation concepts and functionality in the light of established literature standards. We demonstrate that none is able to totally cover the surveyed functionality span, leaving to developers and manufacturers the responsibility of creating custom extensions to fill functionality gaps, thus hindering the interoperability between different technologies.

In order to solve the previously mentioned heterogeneity issue, considered to be one of the greatest problems in the field of Building Automation, this work presents a technology-agnostic framework capable of integrating several automation technologies. Our proposed solution is validated and compared against other solutions, that fall short attempting to solve the same issue, in a production environment within the scope of an European Project where it surpasses existing solutions and, additionally, shows its potential to apply energy efficiency techniques.

Keywords: Building Automation, Domotics, Fieldbus, Systems, Protocols, Information models, Standards, Integration, Framework, Technology-Agnostic, Hardware Abstraction, Ontology.

Contents

List of Tables

List of Figures

Glossary

AC	Afferent Coupling
BACS	Building Automation and Control System(s)
BAS	Building Automation System(s)
BAT	Building Automation Technology(ies)
BA	Building Automation
CCMS	Central Control and Monitoring System(s)
CC	Cyclomatic Complexity
CS	Control Server(s)
DCS	Distributed Control System(s)
DDC	Direct Digital Controller(s)
DOG	Domotic OSGi Gateway
DOI	Depth of Inheritance
EC	Efferent Coupling
GUI	Graphical User Interface(s)
HAS	Home Automation System(s)
HAT	Home Automation Technology(ies)
HA	Home Automation
HMI	Human-machine interface
HVAC	Heating, Ventilation, and Air Conditioning system(s)
ICS	Industrial Control System(s)
IC	Integrated Circuit(s)
LCOM	Lack of Cohesion in Methods
MPC	Methods per Class
MTU	Master Terminal Unit(s) / SCADA Server(s)
NOC	Number of Children
OPC	Object Linking and Embedding for Process Control
OSGi	Open Service Gateway initiative
PID	Proportional, Integral and Derivative

PLC	Programmable Logic Controller(s)
RTU	Remote Terminal Unit(s)
SCADA	Supervisory Control and Data Acquisition
SOA	Service-Oriented Architecture(s)
UA	Unified Architecture
UI	User Interface(s)
URL	Unified Resource Locator
WS	Web Service(s)

Chapter 1

Introduction

Building Automation Systems (BAS) consist of a set of technologies installed in buildings that controls and monitors equipment such as heating, cooling, ventilation, air conditioning, lighting, shading, power systems, life safety, and security systems. BAS aim at automating tasks in technologically-enabled environments coordinating a number of interconnected electrical and mechanical devices in a network. These systems may be found in industrial infrastructures such as factories, in enterprise buildings and malls, or even in the domestic domain.

Building Automation (BA) has been receiving greater attention due to its potential of reducing energy consumption and facilitating building operation, monitoring and maintenance, while improving occupants' satisfaction. These systems achieve such potential by employing a wide range of sensors (e.g., for sensing temperature, CO2 concentration, zone airflow, daylight levels, occupancy levels), which provide information that enables decision making concerning how the building's equipment is controlled, with the goal of reducing the building's expenses while avoiding decreasing its occupants' satisfaction [1]. A building controlled by a BAS is often referred to as an *Intelligent* or *Smart Building*, or a *Smart Home* if the building consists of a domestic residence.

Unfortunately, with the evolution of BAS and technology in general, some related literature references became outdated and no longer offer coherent definitions in this topic. Adding to that, Building Automation is a highly diverse topic that borrows concepts from different disciplines such as computer science, electrical engineering, artificial intelligence, robotics, among others, and for which almost no authoritative text exists. There is no global agreement with respect to concepts and terminology. One example of literature disagreement concerns the application of LonWorks and KNX at the management level of a BAS [2, 3, p. 23]. Another example is the numerous literature references describing Supervisory Control and Data Acquisition (SCADA) systems that are unable to provide a generic enough description of the SCADA systems' most common architectures [4, 5, 6, 7, 8, 9, 10]. Some of these references are outdated, so their accuracy with respect to the SCADA systems of today is thereby questionable [11, Section 3.61 Note 7] [12]. Similarly, the datapoint concept is also provided with contrasting definitions across the literature [13, p. 54] [11, Section 3.61]

With this loss of field-knowledge, manufacturers end up redefining basic concepts repeatedly instead

of building upon already existing references, culminating in a plethora of concurrent definitions, protocols, standards and service implementation methodologies. As a result, the Building Automation market consists of heterogeneous solutions that (i) cannot cover all the expected functionalities expected in a BAS, (ii) are not able to interoperate with other vendors' solutions without additional overhead, locking costumers to the same product line, which becomes a greater issue in case that such a line gets discontinued, (iii) have closed specifications, (iv) are too complex to use by non-specialized personnel, whether they are end-users or system developers, and (v) most solutions only perform satisfactorily in the exact conditions they were tailored for, not performing so well if the working environment changes—lack of flexibility. This is known as the *Building Automation Heterogeneity Problem*, because the interoperability of installations from different vendors constitutes a significant problem for planners, construction companies and users.

The negative impact of this problem is such that, it is even considered by some authors as a *potential barrier for BA technologies around the turn of the millennium* [14, 15, 16]. Several solutions try to circumvent this problem, usually by creating even more protocols and technologies in hope for manufacturers to adhere. Some example of these technologies are OPC UA, oBIX and BACnet/WS, which specify a set of standards that manufacturers have to implement in their solutions in order to interoperate across different technologies. However this did not solve the heterogeneity problem because, once again, they ended up redefining basic concepts and left manufacturers with the freedom to choose which one of the numerous standards they will follow, hence further promoting market heterogeneity. Faced with the aggravation of this problem, open-source communities started developing technology integration frameworks such as DOG and openHAB, which, instead of hoping for manufacturers to adapt to their standards, its the frameworks that try to adapt to already existing technologies. However, these solutions still fall short, for yet redefining existing concepts, lacking of documentation, having an unnecessarily complex design and for failing to completely abstract the underlying technologies.

In the urgency of finding a solution for this problem this thesis proposes a technology-agnostic integration framework which, following the already established literature references, is capable of bridging different automation technologies. Our framework is able to abstract software developers from hardware intricacies, hence facilitating the creation of BA solutions built atop current technologies which take full advantage of the installed resources in any building. We apply an in-depth qualitative and quantitative analysis using well established architecture quality metrics, we concluded that our framework excels in almost every aspect the existing solutions targeting the heterogeneity problem.

To further validate our solution we applied it to an European Project as a case study, encompassing a complex production environment suffering from technology heterogeneity issues, where several applications targeting building energy efficiency had to be developed within a tight delivery schedule. Our framework was able to fully abstract the underlying hardware to software developers, greatly contributing to the project completion within the time schedule. During the project's demonstration phase we were able to access that the applications developed atop our solution were able to significantly decrease energy consumption in the tested areas, exposing the potential of taking full advantage of a building's installations in order to reduce energy consumption.

1.1 Goals and Contributions

The main contribution of this work is the introduction of a technology-agnostic framework that solves the Building Automation Heterogeneity Problem. Further contributions of this thesis are:

1. **Systematization of building automation concepts**
   Since current literature references leave readers with several unclear definitions and terminology, promoting the lack of interoperability, we start by organizing the basic concepts. We describe the most commonly employed control systems in Industrial Automation, narrowing down to the topic of Building Automation. Within that scope we present concise definitions of aspects present in any Building Automation System such as fieldbus and device, based on a wide range of established literature references.

2. **Compilation of the expected functionalities in a BAS**
   We introduce the expected functionality a BAS should be able to provide, based on two well known standards. Later we detail each functionality as a complement to the, often vague, definitions present in the literature.

3. **Overview of BA state of the art market solutions**
   A detailed description is provided concerning the state of the art technologies existing nowadays in the market. This analysis was based on these technologies official specifications, regarding how their information models map the previously expressed functionalities.

4. **Clear evidence that installations are technically forced to promote heterogeneity**
   During this analysis we have gathered clear evidence that, having no single technology providing all the expected functionalities, in order to get all the desired services in a building constructors are forced to install more than one solution. This promotes even further the heterogeneity problem and raises the need for technology integration solutions.

5. **In-depth analysis of technology integration solutions**
   Concerned by the previous revelations, we evaluated two existing integration solutions, performing an architectural analysis in order to infer how easily can these solutions be understood, reused and extended. Additionally we also verify how they facilitate the task of integrating new automation technologies.

6. **Design of a framework capable of integrating a wide range of automation technologies**
   We propose a technology-agnostic integration framework capable of integrating different automation technologies, abstracting software developers from hardware specificities, enabling the creation of BA solutions capable of taking full advantage of the installed resources in any building.

7. **Validation of the proposed framework in a production environment**
   Finally, our proposed solution is validated in a production environment within the scope of the European Project SmartCampus. Moreover, it is also shown that our solution can still be employed in scenarios not strictly belonging to the context of Building Automation.

1.2 Thesis Outline

The remaining of this document is organized as follows:

Chapter 2 provides a systematization of building automation basic concepts which are critical for understanding this work. This section starts with a brief introduction to the basic concepts of control and automation describing Control Systems and Industrial Control Systems, narrowing down to the Building Automation domain, detailing the basic concepts used in this field of study.

Chapter 3 segments the basic functionality that should be present in any BAS, according to several literature references and standards. Functionalities such as Grouping and zoning, Event Notification, among others, are thoroughly detailed throughout this section.

Chapter 4 overviews the most common Building Automation technologies, detailing how they cover the functionalities revealed in Chapter 3 into their design.

Chapter 5 evaluates two existing integration solutions, ascertaining how these solutions help technicians performing the task of integrating automation technologies.

Chapter 6 proposes our framework capable of integrating different automation technologies and highlights the main implementation challenges.

Chapter 7 validates the proposed solution against two case studies.

Chapter 8 summarizes the work, presents the conclusions and points for future directions.

Chapter 2

Background

Literature references often provide unclear definitions of critical concepts in Building Automation. In an attempt to fill the definition gaps this section starts with a brief introduction to the basic concepts of control and automation by describing what is expected from a *Control System*. Then, we focus on the notion of *Industrial Control Systems*, which can be divided into two main categories, namely, *Distributed Control* and *Supervisory Control* Systems (Section 2.3). Finally we narrow down to the *Building Automation* domain (Section 2.6), detailing the basic concepts used in this field of study.

2.1 Control Systems

A *system* consists of a group of related elements—components—that operate together, transforming inputs (stimuli) into outputs (responses) [17]. A system that is a component of a larger system is called a *subsystem*.

A *control system* is a type of system built with the purpose of obtaining some desired outputs, within desired behavioral constraints, given specific inputs [18, 19]. The component of a control system, responsible for its control activity, is called a *controller* (described in Section 2.7.2).

Control systems are mainly used for (i) power amplification, where a low-power signal can control a larger one, such as a crane that amplifies the human force to pick heavy objects, (ii) remote operation, to handle tasks in nefarious environments where no human could survive, just like the handling radioactive materials, (iii) input transformation, where one type of input can be converted to a different type of output, for example rotating a knob translates into varying a light's intensity, and (iv) for disturbance compensation, specially in very complex systems where few or no humans could take into account every disturbance affecting the system [19].

Control systems have to fulfil some basic requirements: (i) the system's operation should be stable, (ii) the system's outputs must adapt to the inputs, (iii) external disturbances should be compensated—for example a system that maintains the same target temperature in a room, no matter how cold or warm the ambient temperature is—, and (iv) the system should compensate the lack of accuracy of its internal model to some extent [18]. A plane is an example of a control system that has to offer stability

to avoid falling, while obeying the pilots' inputs, while withstanding environment changes such as winds and weather conditions, and because creating a mathematical model of a plane is too complex since these systems have a highly dynamic nature, the plane should care for minimizing the error of its actions accordingly, like constantly maintaining an altitude very close to the pilots' instructions.

2.2 Automation and Control

The terms *automation* and *control* are often treated synonymously due to their close relationship. Within the area of control, we distinguish (i) open loop control referring to one or more controllers carrying out a series of actions autonomously sustaining their decisions on an internal state (Figure 2.1.a) [20]; and, (ii) closed loop control referring to scenarios where controllers continuously read the environment and reason about the results of their own actuations in order to maintain some targeted parameters within desired values—known as set-points— (Figure 2.1.b) [11, p. 7-8,20].

Typically an automation systems measures and controls a technical process (e.g., a manufacturing process, a vehicle, a building and its surroundings)(Figure 2.1.c) [2]. An example of automation in the context of Building Automation is a system that actuates on its devices, for instance, according to the time of day, like setting a temperature of a room at night according to some pre-defined schedule, thus instructing controllers to adjust the room's Heating, Ventilation, and Air Conditioning (HVAC) systems output temperature, where each controller will read from a temperature sensor present in a room in order to receive feedback for its actions while reaching the desired temperature set-point (see Section 3.5). Another example is when, during the daylight, the system must maintain the lights' intensity at a certain desired value by having controllers increasing or decreasing the lights' intensity avoiding the room from getting darker or brighter than the desired set-point. This set-point may vary according to some determined schedule, affecting the way controllers orchestrate the lamps.

2.3 Industrial Control Systems

Industrial Control System (ICS) is a term that describes control systems which orchestrate several process-control activities usually regarding the supervision and control of industrial processes.

A process is a set of interrelated tasks that, together, transform given inputs into outputs (refer to Section 2.2) [21, p. 66].

In ICS, controllers operate processes throughout their duration, gathering data regularly in order to make operation decisions in a way known as control loop [8]. Controllers measure data through sensors (devices that convert some physical reality into electric signals) and act upon that data using actuators (devices that interact with their environment). Process controllers can be implemented in several ways, they can be a software running on a server that controls all the processes in the system (thus centralizing the control function) or, they can be embedded directly in the system devices, using controller devices that interact directly with processes.

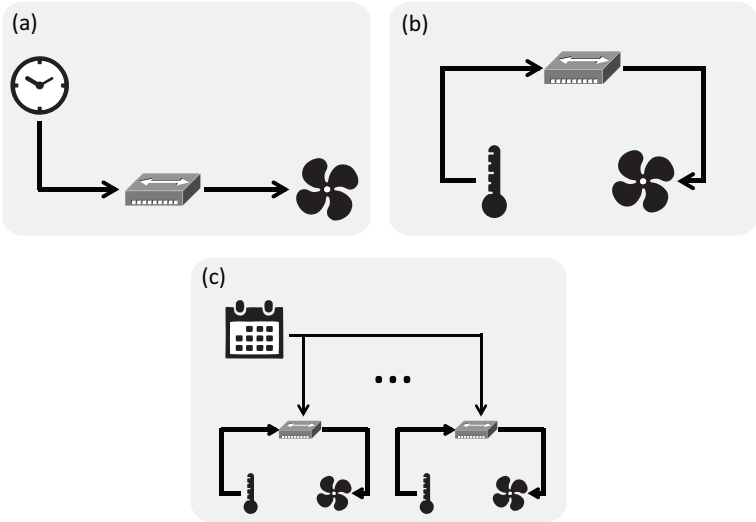

Figure 2.1: Relation between control and automation procedures. (a) The controller varies the way it controls the fan's speed according to the automation system's preferences, in this example, concerning the daytime (open-loop control). (b) A controller adjusts the HVAC output temperature through the feedback received from a temperature sensor, that measures how far the set-point value is from being reached (closed-loop control). (c) Automation system updating the controllers' set-points according to a given schedule.

An ICS can follow numerous network topologies, but for the sake of simplicity we only focus on *Distributed* and *Centralized Control* topologies which are the building blocks of more complex network systems [8].

2.4 Distributed Control Systems

A Distributed Control System (DCS) is a type of ICS that uses a decentralized control approach, delegating its responsibilities on distinct controllers: each one operating field devices (sensors and actuators that control processes) typically over the control network or discrete cabling. As part of their control function, controllers can also communicate with each other distributing the control-loop between several controller modules. For example, a controller attached to a light sensor, can request another controller, that is attached to lamp actuators, to adjust intensity until the sensor measured intensity falls between the desired values.

Controllers are integrated into a layered distributed control architecture, in which they are supervised by the Control Servers (CS) [8]. CS reside at the top of the control architecture and are responsible for orchestrating all the processes in the network by requesting information directly from controllers, located in the layer below, regarding the state of each system's process they control (Figure 2.2). Optionally, a CS can provide some type of Human-Machine Interfaces (HMI) responsible for presenting information regarding the system, and facilitating human intervention when needed. Ideally, in a modular system architecture, HMIs can be decoupled from control servers, thus enabling the creation of remote interfaces (like a web-based or mobile interface [9, 10]).

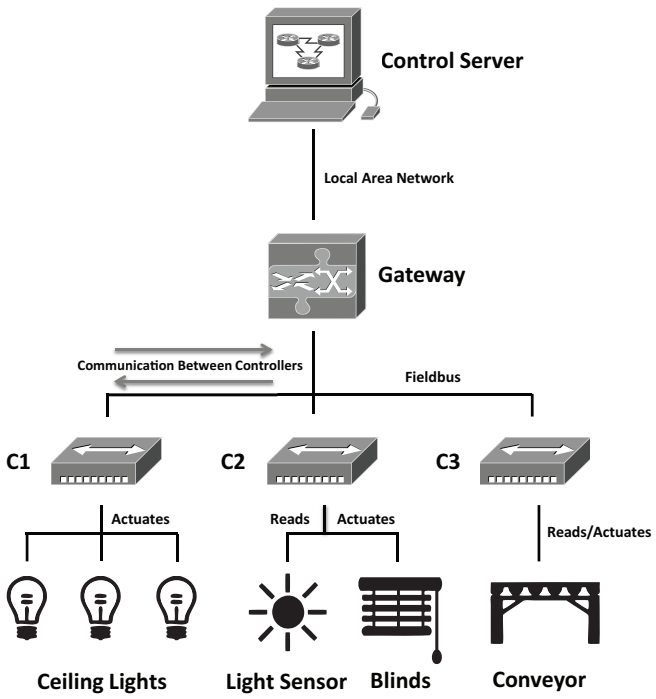

Figure 2.2: Example of DCS Layout. A Control Server supervises three controllers' activities. A Gateway interfaces the LAN to the fieldbus network. C2 communicates with C1 requesting the lamps intensity to be dimmed until C2's sensor reads a certain desired value. C3 performs a distinct control-loop which controls the speed of a conveyor.

Although CS may be directly connected to the process controllers[1], gateways are often required to bridge the different fieldbus protocols, for example a gateway is usually required to bridge between UDP/IP and LonTalk [22]. The CS operates with real-time information and has the capability to react automatically to that information stimuli, using some pre-programmed control logic [10].

Due to the physical proximity of controllers to devices in a DCS, these systems are usually employed in environments where the information is mostly relevant at the instant it arrives, and where the latency between action requests and their arrival to the target controllers must be minimal.

2.5 Supervisory Control and Data Acquisition

Supervisory Control and Data Acquisition (SCADA) systems, also known as Centralized Control and Monitoring Systems (CCMS) [20], consist of an ICS managing several sub-systems, such as DCS or even other SCADA systems, centralizing control and data acquisition. Typically these sub-systems are geographically disperse and usually less complex since they only supervise local processes. SCADA systems can be found in power grids, water, oil and natural gas distribution systems, among others [8].

SCADA systems enable monitoring of remote stations by presenting their current status and some more specific monitoring information, such as alarms and events, often ordered by their priorities, allowing managers to undertake appropriate corrective actions, when needed, by sending commands to the

[1] What differentiates SCADA and DCS: `http://ammas.com/topics/Cars/a204142.html`.

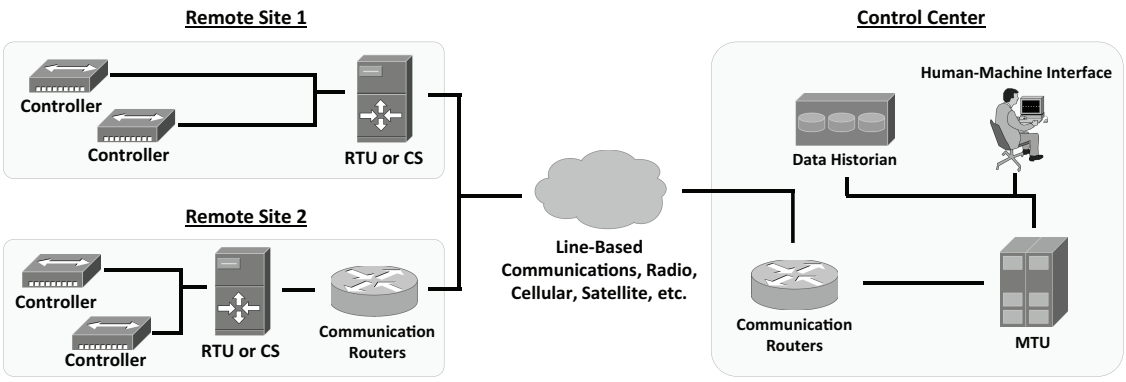

Figure 2.3: General Layout of a SCADA System. Processes are controlled by a DCS located in remote sites. These remote sites have routers or terminals to communicate with the control center. The MTU receives and processes the information, presenting it in the HMI and recording relevant data in the Data Historian server.

remote sites, in order to recover from system's malfunctions. In control centers the personnel responsible for the control and supervision of such environments (the operators) have access to all the system's information, that is displayed through a specialized HMI.

At the heart of a SCADA system is the Master Terminal Unit (MTU), also known as SCADA Server, that is responsible for the data acquisition and for offering services to manage the infrastructure. An MTU also communicates the gathered information to a Data Historian Server responsible for filtering and storing relevant information in order to keep track of the system's activity for diagnostics and future reference [8, 10].

Contrary to a DCS, where a CS directly supervises field devices, a SCADA Server has no direct connection to them, such connection is provided by a Remote Terminal Unit (RTU) installed in remote stations interfacing its remote station's sub-system with the SCADA Server, by sending relevant information about it and receiving commands from the control center. An RTU also controls devices locally, thus it can be seen as an CS with extended communication capabilities. Figure 2.4 depicts a possible relation between SCADA and a DCS architectures. The large scale communication between remote sites and the control center can be performed through radio, cellular networks, satellites or the Internet (see Figure 2.3).

2.6 Building Automation

A BAS is a distributed system oriented to the computerized control and management of building services, also referred as Building Automation and Control System (BACS) [2]. The architecture of this distributed system can be organized into three layers [23]: (i) The lowest layer is known as the Field Layer where the interaction with field devices happens, (ii) the middle layer is the Automation Layer, where measurements are processed, control loops (see Section 2.2) are executed and alarms (described in Section 3.3) are activated, (iii) the top layer is the Management Layer, where activities like system data presentation and archival take place [11, p. 52].

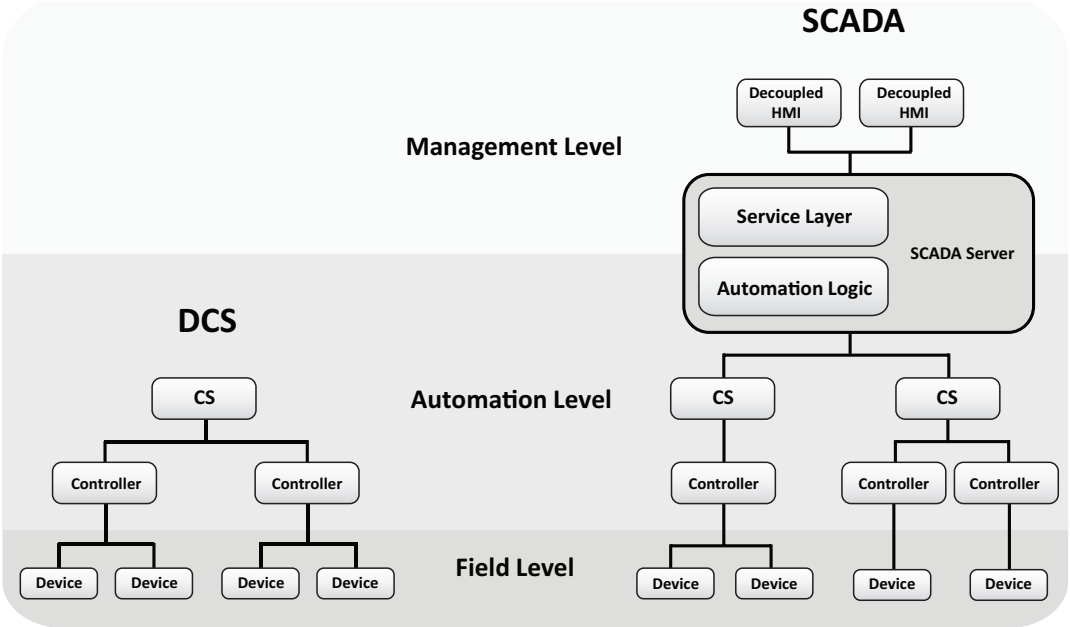

Figure 2.4: Relation between a DCS and a SCADA system. A typical DCS's architecture (on the left), that, is compared with a typical SCADA system's architecture (on the right). The SCADA system manages several DCSs. The SCADA Server (on the right) interfaces the automation systems with the HMI at the management level through a service layer.

Complex BAS often consist of a SCADA system managing several DCS. Such DCS are used to operate confined controlled environments of the building (like rooms, hallways, etc.) while the SCADA system is responsible for the high-level supervision of various sections of the building's infrastructure such as floors or even the whole building.

Modern BAS tend to separate the automation logic from the user interface through service-oriented abstractions, providing flexible access to the BAS from several different platforms and locations [24, 9, 10].

2.7 Building Automation Concepts

2.7.1 Fieldbus

A fieldbus is a digital data bus that allows communication between devices at the field level such as controllers, sensors, and actuators [2, p. 23, Chapter 2] [25, 26]. A fieldbus aims at improving communication quality in comparison with previous analogical communication buses, and at reducing installation costs by cutting down the required wiring, since devices connected through fieldbus only communicate digitally. Devices connected to a fieldbus network are expected to have some computational power, and may even replace several analog devices at once, further contributing to drop installation costs [27].

Many fieldbus systems currently exist on the market with specifications that vary according to requirements of their application [2]. Besides many vendor specific solutions the main standards used today are BACnet [28], KNX [29], LON [30], Modbus [31], ZigBee [32, 33] and EnOcean [34, 35].

2.7.2 Controllers, Sensors and Actuators

The setup of a BAS comprises electric actuators, sensors and hardware modules. Actuators react to signals closing circuits or varying the intensity of electric loads, which are physical devices such as a window blind or a ceiling lamp. Sensors are devices that convert a physical reality into a signal that can be measured. Finally, actuators and sensors are attached to I/O ports of hardware modules that produce electric signals according to digital output commands and create readings from input signals.

Automation hardware is highly heterogeneous and should be abstracted in a way which enables software applications to be as independent as possible from the specificities of the hardware. A common abstraction is to categorize devices into sensors and actuators. Although, some devices fit in both groups due to their sensing and actuating capabilities, but for simplicity they may be perceived as two different virtual devices: one device capable of sensing and another one capable of actuating. Sensors typically reflect a generic class of devices that encompass any device capable of sensing external interactions, besides environment sensors, such as a wall switch. In turn, actuators are often confused with electrical loads or with the physical device they drive, for example, what is commonly understood as a blind actuator is indeed a motor actuator (attached to a blind).

The interaction between devices must be orchestrated through some type of control logic. Such logic lies in components known as *controllers*. In building control systems a controller usually consists of an application-specific hardware with embedded software that continually controls physical actuators (such as lights, blinds, among others) depending on the feedback given by monitored inputs (such as light or occupation sensors) or by receiving commands from the system [11, Section 3.55].

From an application point-of-view controllers expose different types of logic objects that can be read or written. The main types are: memory registers and timers. Depending on the sophistication of the controller, it may be capable of running more than one control program simultaneously, reading and writing I/O ports or communicating with other controllers over the fieldbus network.

Control function logic may have different complexity levels, ranging from simple binary conditions developed using ladder logic or function blocks [36], to mathematical expressions or even more sophisticated algorithms such as Fuzzy Reasoning [37]. Depending on the sophistication of the control functions controllers can be distinguished between *Programmable Logic Controllers* (PLC) or *Direct Digital Controllers* (DDC).

PLCs typically implant simpler more rigid functions that require little or no configuration, while DDCs are more flexible and typically implement functions that require extensive configuration such as scheduling or scenario management.

Control functions can also be implemented in non-embedded software running in a server, thus centralizing the BAS's control functions in its higher level layers, where the control logic can make use of high-level concepts belonging to the application layer. The advantage of a software controller is that it may undertake more complex decisions or explore additional information gathered from the system.

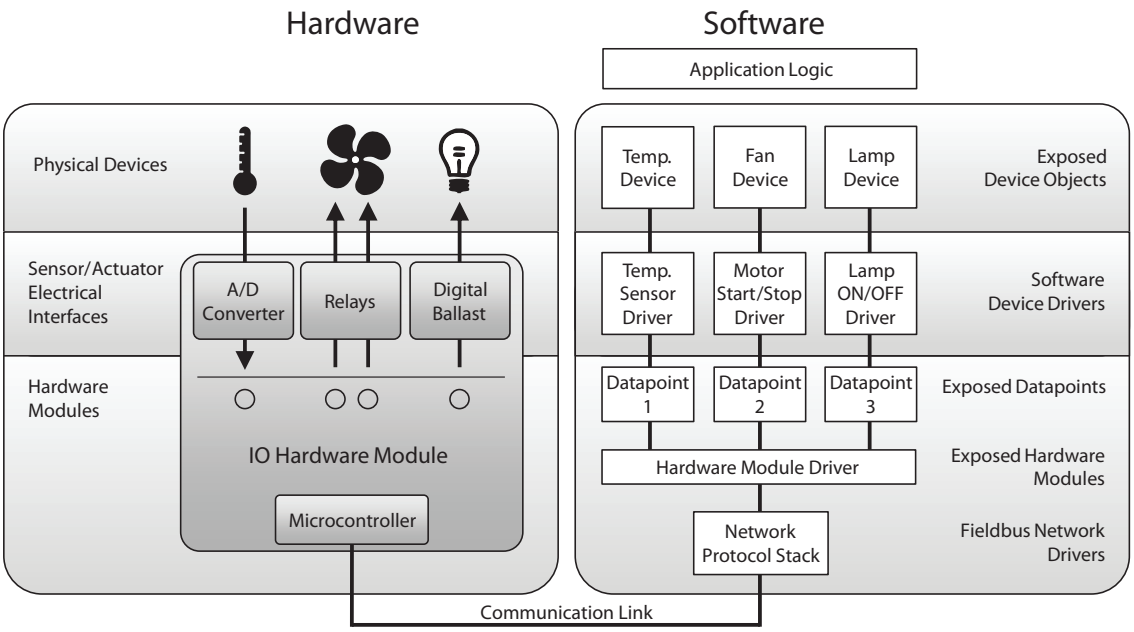

Figure 2.5: Building Automation Hardware and Software stack. Physical devices are connected to an I/O module, responsible for exposing them as addressable entities (on the left). The corresponding software representation of the hardware stack, where the I/O module is operated by a software device driver, exposing each port and each device connected to it (on the right).

2.7.3 Device Model

So far as it concerns a BAS, devices have two interfaces, an *electrical interface*, that defines how to connect the device to the rest of the system, and an *application interface* that enables other devices and software applications to interact with the device through exposed datapoints.

A device driver is a component that controls devices connected to a system, while simultaneously providing a layer of abstraction that simplifies their operation. Each device driver corresponds to a particular type of device, meaning that a system having to support hundreds of different devices, must install hundreds of device drivers. The great diversity of devices available poses a challenge to the creation of a generic device driver that fits every device of the same type, i.e. devices that have compatible interfaces. The interface of one device is said to be compatible with another if some of the properties of one interface exist in the other and share the same data-type. For example, a common device driver controlling every lamp device where all lamps have the same properties and methods, such as on or off. In BAS we can distinguish *hardware device drivers* and *software device drivers*.

Hardware device drivers consist of hardware modules with I/O ports and a micro-controller. The I/O ports are connected to physical devices, which usually possess no intelligence and are directly operated using electric signals, such as a lamp, a thermistor or an air conditioner's fan. The micro-controller is responsible for driving those devices and exposing an interface that can be used to address each I/O port for reading and writing purposes. This hardware module can be connected to a computer, a network or to another hardware module [38, p. 523]. From a software point-of-view, reading from a sensor is

equated as reading from a variable that represents a hardware input port and commanding an actuator is equated with writing a value on a variable representing an output port. So when a device is created we have to specify what are the input and the output ports where the device will be connected.

In contrast, software device drivers consist of application programming interfaces used to enable other applications to interact with devices. These device drivers are used to (i) convert electric signals into values stored in variables or objects that can be read by software applications, and (ii) in contrast, they also convert values into electric signals that drive actuator devices. The first conversion relates to the situation where the applications must read the values from a sensor device, while the second conversion enables software applications to actuate on devices. Such actuations may consist of modifying device's properties or make a device actuate in its environment of influence.

Overall, physical devices are mapped into I/O ports by hardware device drivers running in hardware modules. Hardware modules, in turn, can be connected to other hardware device drivers, providing a higher level of abstraction, or to a software device driver that will act as an hardware abstraction layer, serving other software applications with the capability of operating those devices, without concerning with low-level—hardware related details—such as device communication protocols, number of connections used by a device and baud rates. Figure 2.5 illustrates this situation, where physical devices are connected to I/O ports of a hardware module (acting as a hardware device driver) capable of providing an interface that abstracts these connections. Such interface is then used to connect the hardware module (and thus connecting the physical devices) to a network. On the other hand, software applications require an abstraction layer, provided by software device drivers, to operate those devices. The network device driver will implement the network's protocol stack, abstracting upper layers from communication specificities and exposing the hardware module's presence in the network. This is followed by the hardware module's device driver that will operate the module directly, providing a simpler interface to the upper layer, exposing the module's ports and mechanisms simplifying the tasks of reading and writing to those ports. Each port will have a device driver associated that will expose the device connected to that port and the corresponding operation mechanisms. Finally, high-level abstractions of these devices are created by defining objects that represent those devices and can be manipulated by software applications.

2.7.4 Datapoints

Datapoints, endpoints, tags or points are presented with different definitions across the literature [13, p. 54] [11, Section 3.61], however they all describe datapoints as an addressable point of interaction between the control system and its domain objects [3, 39]. Datapoints can be physical or virtual. A physical datapoint is directly related to a device connected to the system, such as devices I/O ports (where each port can be mapped to a datapoint) [32, p. 10] [40, p. 96]. In contrast, a virtual datapoint acts as a way to address virtual objects such as services provided by devices, for example, a temperature sensor exposing a datapoint that when read returns the average temperature measured in the last hour, or a datapoint addressing a configuration parameter of a given device, where writing to that datapoint

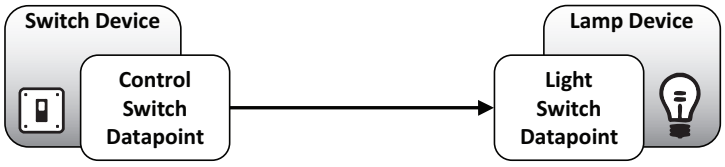

Figure 2.6: Visual representation of two datapoints bound together, where the switch device's datapoint notifies the lamp device's datapoint of status updates.

affects the associated configuration parameter [41] [42, Section 2.7.2]. Virtual datapoints can also be used to address stub devices for testing purposes.

Every datapoint has meta-data associated which describes a set of rules for the interaction with that datapoint, where typically we can find the access type, the data-type, installed location, influence zone and a value update rate for reading and writing operations.

Datapoints offer one of three types of access: read, write, or both. Readable datapoints are read-only and usually relate to sensor devices. Writable datapoints are write-only and relate to actuator devices—writing to datapoints equates to updating the system's state. Datapoints that offer both access types may be used to write updates into a device, such as turning it on, and to read the device's internal state from it, for example, to probe the device on/off status.

Datapoints that provide a read access type should ensure a regular value update rate so that every client application knows how often it should poll that datapoint for value updates in a given period of time, this is known as smallest sampling interval. It would be pointless to read a datapoint faster than its update rate because we would be reading the same value repeatedly. On the other hand, datapoints that support writing operations may provide a maximum rate at which writings can be performed. For instance if a lamp device and its associated writable datapoint are changed in a too frequent fashion, hardware damage may occur. In addition, datapoints may expose only their new value when a significant change of value occurred.

In addition, datapoints have a *data-type* associated, which tells client applications how the information is structured when they read from a datapoint and how it must be structured when writting to that datapoint. Moreover, data-types can have semantic information associated with them, usually represented by a *unit*, which in turn describes what does that structured information represent for a given domain context. For instance, a datapoint's value can represent a binary quantity, which is unitless, or a temperature in Celsius, a percentage, an applied force in Newton, among others.

In some systems, datapoints having the same data-type can be linked together in a process known as *binding*. This means that when a datapoint is updated, linked datapoints are notified and may perform an action, such as obtaining the same value. A practical example of binding is a switch device that has its datapoint—representing the switch status—bound to a lamp device's datapoint, when the switch is pressed the lamp device turns on or off accordingly (Figure 2.6).

Non virtual datapoints always belong to devices which are installed in a physical location of a building. Knowing the installed location of a datapoint is important, specially if that datapoint belongs to a sensor

device. Conversely, datapoints also have a zone of influence, i.e. the space affected by their actuations, which may not be the same as the installed location. For example, an HVAC system usually occupies one room in the building and affects several other rooms, while a lamp device affects the same room as it is installed in.

2.7.5 Commands

Commands are operations that can be executed on devices that, if successful, cause devices to change their internal state or to actuate in their environment of influence. For example, setting the intensity of a lamp is the result of executing the command “Dim to level”.

Commands have parameters specifying what operation should be done, and attributes specifying how it should be done. The command that dims a lamp to a given level requires specifying the target value for the intensity parameter. Commands can be sent to devices individually or in group if all devices in a group can accept that command (Section 3.2).

2.7.6 System Stakeholders

In a BAS, we can find several stakeholders, i.e. agents which affect and are affected by the system. Commonly these stakeholders are the *system engineer*, the *building operator*, the *installation technician*, the *maintenance technician*, and the *occupant*. Frequently we can find one agent assuming more than one role simultaneously, for example the building operator can also be the maintenance technician, and also an occupant of the building.

The system engineer, maintenance and installation technicians undertake the project specification, maintenance and installation activities respectively, such as planning the control strategy, repairing devices, installing new functionality or commissioning the system [43, Section 5.4].

System commissioning refers to the application of a set of engineering techniques—systematic application of scientific and technical knowledge—by the installation technicians, to plan, design, install, calibrate, configure, inspect and test every operational component of the system [44, Section 3.4] [43, Section 5.4.6]. Usually the commissioning process is definitive and configurations do not change once the system is functioning, but sometimes some steps have to be later reiterated due to updates in the system, for example during the installation of new features or devices in the system [11, Section 3.42, 3.47].

While these tasks are generally executed by certified authorities, some manufacturers already provide the building operators with commissioning and management tools to ease the commissioning of these systems. One example is the ETS tool used to commission KNX projects [45].

A building operator, also known as building manager, is the person who supervises and manages the system. More specifically, building operators are responsible for system status supervision—such as monitoring the system for events and alarms—and adjusting system’s parameters according to the building occupants’ needs [43, Section 5.1.1.2] and for calling the technicians when necessary.

Every BAS has an HMI providing the building operator(s) with supervision and controlling capabilities.

Typically, HMIs expose the functionalities offered at the Management level such as management of scenarios, scheduling services, alarm notifications, among others. Although the operator's interaction with the system isn't essentially assigned to the Management level [43, Section 5.1.1], since some tasks require direct interaction with lower levels, providing functionality not covered by the Management level software. For example if a device needs maintenance or requires rebooting, the operator must interact physically with the device by pressing a reset button, such interaction would lie at the Field level.

Finally, occupants can also directly impact the way the BAS actuates. For example, rooms can have occupancy sensors to decide whether the lights should be on or off. Some systems provide local control panel interfaces so that occupants can send commands to the devices in that room [43, Section 5.3.5.3]. Another approach for letting occupants interact with the system is through *comfort voting* where occupants can tell the system how comfortable they are with the room's current settings, hence indirectly controlling the system. Studies have shown that this technique can improve a building's performance [46, 47]. Additionally occupants may also inform the building operator for irregular situations, such as some sort of device malfunction.

Chapter 3

Building Automation Functionalities

In the vast field of Building Automation we, as application developers, are often unaware of what is expected from existing solutions. In an effort to enumerate the basic requirements of a BAS, this section analyzes the functionality that, based on several literature references and standards, is believed to be essential to be honored in modern systems [43, Section 5.1.1].

3.1 Functionality Overview

Table 3.1 groups the functionalities described in two international standards of reference in the following categories: Grouping and zoning, Event Notification, Alarm Notification, Historical Data Access, Scheduling, and Scenarios. These functionalities are described in the following of this section.

3.2 Grouping and zoning

A device group is a logical identification of a set of devices. There are two fundamental types of groups: device collections and command groups. Device collections consist of a set of devices used to organize or structure large installations. For example, “all devices in hallways” could perhaps consist of all luminaries and occupancy sensors of hallways. Command groups are collections of devices with compatible interfaces that are intended for simplifying commanding multiple devices at once. Interfaces are said to be compatible if they recognize the same commands or expose the same type of datapoints. Consider two interfaces, one for controlling a lamp and other for an HVAC system. If they both accept the commands on and off, they are said to have a degree of compatibility with each other, where the on command will lit the lamp and turn the HVAC on and the off command will turn both devices off.

Device groups can be defined by hardware or by software. A hardware device group is a group created by the use of a hardware device module that abstracts several devices into just one device. For example, an air conditioner capable of measuring temperature and ventilate the air can be an abstraction of two independent devices, one being a thermistor and the other a fan. Another way to conceive a hardware device group is by using network logic: several devices can belong to the same sub-network,

	ISO 16484-3	EN 15232
Grouping/Zoning	Individual zone control is listed as a required functionality. (Section 5.1.1.4)	The concept of zones is used to define room or zone specific control activities and setpoint preferences. (p. 27, 32, 46)
Event Notification	Event handling and notification are described as an operator function provided by the HMI. (Section 5.1.1.2, 5.3.5.17)	Refers to the notification of changes in the system's status changes for several purposes such as controlling indoor occupation.†
Alarm Notification	Alarm notification is described as an operator function provided by the HMI. (Section 5.1.1.2, 5.3.5.17)	Alarm notificiation are essential to detect malfunction situations.†
Historical Data Access	This standard presents data archiving and the means for retrieving that data as management functionalities of a BAS. (Section 5.3.2.9)	States that data collection and logging are features offered by building management systems. (p. 10)
Scheduling	Provides a section describing time scheduling features. (Section 5.3.5.15)	Specifies the use of schedules to control some attributes such as the air flow in a room (Section 7.5.1)
Scenarios	There's no direct reference to the "scenario" terminology, although, energy management services are seen as a requirement that according to EN 15232 benefits from this concept. (Section 5.1.1.3)	The standard defines several system operating modes that are used to adapt the operation of the building to occupants' needs. Each operation mode can be modelled as a scenario.†

Table 3.1: BAS functionalities addressed in ISO 16484-3 [43] and EN 15232 [48]. † This aspect is referred throughout the document EN 15232.

thus forming a group. In contrast, software device groups are managed by software applications' data structures such as lists, where each list represents a device group, containing devices belonging to that group. Software device groups have a major advantage over hardware device groups, they can be dynamically arranged in order to add or remove members without requiring any effort modifying the system's structure whereas hardware device groups would require hardware modifications or network restructuring.

Groups of devices gather devices that are supposed to be commanded together with a given purpose. The group will thus behave as a virtual device whose properties can be assigned; assigning a property of a group will perform the corresponding assignment on each device that is installed in the group. A group can be "All lamps in corridors" that are to be commanded together in some situation. Groups are closely related with the concept of zoning.

Building automation is intrinsically related to the idea of controlling spaces, since most actuations are confined to specific spaces. We can say that spaces are divided into several sub-spaces, known as *zones*. A zone can be a floor, a room (or just part of it) or an area within another zone. Zones may be related according to *containment* and *adjacency*, providing information about the building (for example to identify and distinguish halls from rooms in a building). Furthermore, a zones is characterized by meta-data such as the zone name, space usage profile and location within the parent zone as well as its boundary polygon that defines the zone's shape and borders.

Arranging spaces into zones helps commissioning the BAS and eases the task of the user in understanding, navigating and recognizing the controlled area in a user interface software. Zone information is also useful for querying spatial aspects of devices, such as where the devices are installed and their

influenced zones.

3.3 Events and Alarms

An event consists of any occurrence that modifies a system's state (a door that opens or a lamp that turns off). The user of the BAS may choose to be notified of certain events, usually conditions that have been verified, such as a room's temperature that changed 2 degrees Celsius since the last measurement, a specific door has just opened, a new person entered a room, and so on [11, Section 3.74].

Alarms, on the other hand, are exceptional events generated by the system's operating conditions. Typically, alarms correspond to exceptional conditions originating in a specific device or group of devices. Alarm events capture a malfunction in some device, lack of a resource essential for the execution of a process or a condition on the overall system's state. The information pertaining to an alarm indicates the apparent source of the problem and may also indicate when the originating alarming condition ceases. Alarms have severity conditions associated to the degree of disruption of the related object's normal operation. In contrast with regular events, alarms also have an acknowledge process: either they require manual acknowledgement or may be transient, meaning that they are cleared whenever the condition that initiated them is no longer verified [11, Section 3.10].

3.4 Historical Data Access

Logs are essential to understand the activities of complex systems, particularly in the case of applications with little user interaction in BAS. Data logging is the process of recording commands sent to devices, devices' state transitions and events (such as alarms), in order to understand system activity and for diagnostic purposes. Logs may contain information about events, processes, alarms and user interactions [43, Section 5.3.2.10.1]. Log records can be split in two categories: *temporary* and *permanent*. Temporary records are relevant in a short period of time after their occurrence, and after that period they can be removed. Permanent records must be kept for the entire life-cycle of the system. It is important to note that the preservation of permanent records is very demanding in terms of storage. For that reason, it is important to intelligently select what information should be stored permanently, the sampling rate and possibly the usage of data compression techniques.

Data analysis is a very useful functionality built atop of data logging systems, enabling the extraction of relevant information such as energy consumption and performance forecasts [49]).

3.5 Schedules

BAS are often required to execute certain tasks according to given schedules. This is achieved by providing a scheduling service in order to associate task execution—commands sent to devices—with some moment in time. Schedules are uniquely identified by the combination of their date, time and the

tasks being performed. They can be defined according to fine-grained units of time such as days, hours, minutes or even seconds. As time passes, scheduled commands are executed.

Scheduled events can be *one-time events* or *repeatable events*. For convenience, the system should support the creation of several schedules; for example, two schedules used to manage the building's illumination in the winter and summer. Schedules are embedded in hardware controllers, but in larger facilities they are often configured in the main server that runs the BAS software [2].

3.6 Scenarios

A scenario describes the desired status of a device or a group of devices associated to some context (such as the time of day or occupancy rate), relative to one or several zones.

Scenarios can be static or dynamic. A *static scenario* doesn't change the state of devices once it's set, meaning that it does not prescribe actuation whenever environment conditions change. A static "studying scenario" can be defined as having the lights on over the table. The "TV scenario", in turn, would switch off all luminaries so that the occupant could watch TV comfortably, and provide a dimmed ambient light near the corridor. Thus, activating a scenario amounts to setting the devices to the respective pre-sets parameters defined in the scenario.

In contrast, a *dynamic scenario* describes how the system reacts according to changes in the environment. A dynamic "studying scenario" specifies that lamps have to be turned on if there is not enough light. Implementing dynamic scenarios may get complex when they vary according to occupant's preferences. This is specially true when considering the preferences of more than one occupant in a zone or taking into account multiple factors simultaneously such as energy consumption peak times, noise, temperature or humidity. Currently, this is a problem without an ideal solution [50].

Another important aspect of scenarios is the time dimension. Device parameters can be set in sequence having certain delays, i.e. when a scenario is activated the effect on the devices' status may not be immediate. Consider a "morning" scenario which specifies two actions: (i) in the 30 minutes before the first person is expected to enter the building, the heating system, which was previously turned off to save energy, should turn on in order to heat the building according to a given set-point. (ii) Then, 30 minutes later, the building's main entry door gets unlocked so that people can enter while, hopefully, the building's temperature will be at the desired set-point.

Chapter 4

Building Automation Technologies

Information models represent, characterize and relate concepts of a given domain by abiding to a common representation of information, which different agents can use to exchange information with each other. In this section we will map the previously established functionalities to state of the art technologies sold today in the market, abstracting from low-level details regarding physical, electrical and network specificities such as devices architectures or protocol data. Instead we will focus on how the information models address fundamental BAS concepts such as grouping, notifying, scheduling, commanding, among others. Although many standards enable implementing these functionalities through model extensions, software plug-ins, extra hardware modules, and so on, we will only consider functionalities natively provided by the standards' specifications, which formally define how such functionalities should work and get implemented. Otherwise manufacturers are left with the responsibility of implementing these functionalities, having freewill for promoting closed proprietary solutions, thus hampering interoperability with other market solutions, resulting in costumer lock-in.

4.1 KNX

KNX, formerly known as European Installation Bus, is a building control communication system that distributes control across devices through *functional blocks*. A functional block consists of a group of datapoints and a behavioural specification about the device, for example a "binary push button" functional block representing the functionality of an on/off switch [51]. Each functional block can be associated with one device. Although a device must implement at least one functional block, it may have multiple functional blocks. The KNX specification already defines some standard functional blocks and datapoints.

Datapoints represent devices' inputs and outputs, and application parameters such as an internal variable storing the minimum light intensity that a lamp controller can provide. Datapoints have an address and data-types associated [52]. Typically, devices communicate by writing to other devices' datapoints via group communication. Datapoints can be bound to other datapoints (thus joining a group),

in order to notify other devices from updates in the system. When a datapoint value changes, the new value is propagated to all datapoints bound to it enabling modules to receive notifications of datapoint updates from other modules.

The KNX specification defines three distinct ways of binding datapoints: *free binding*, *structured binding* and *tagged binding* with increasing levels of semantics. In free binding datapoints having the same type can be linked freely. Structured binding follows a certain pattern defined by the information model for linking datapoints based on devices' functional blocks. For example, a push-button must have its output datapoints linked to the input datapoints of the device it will control. Finally, tagged binding proposes that part of the datapoint's address should contain some information in order to highlight some aspects like the group the datapoint belongs to. This can be used to select datapoints of a given zone, where parts of those datapoints' addresses represent their location, thus tagged binding can be used to group datapoints or devices [53]. Scene control is also supported, having three distinct approaches: (i) Setting the scene conditions via a commissioning tool, (ii) Setting the scene conditions of the connected actuators and storing this scene as a scene number in the connected actuators, and (iii) by using a datapoint type called scene number [54, Section 4.19]. KNX natively supports device grouping through the Group object type, which can be used to implement zoning functionalities.

KNX devices are commissioned by the Engineering Tool Software (ETS)[1], that provides the user with a higher level of abstraction with respect to the system's configuration complexity. KNX has some limitations: its specification doesn't natively refer to historical data access features, event and alarm notification, task scheduling and scenario management features, giving each vendor free will to implement such functionalities at higher layers without following a base guideline.

4.2 LonWorks

LonWorks is a networking platform developed by Echelon Corporation [55] [42, Chapters 2 and 3]. In LonWorks a network device is called a *node*. Nodes have a unique address and may implement multiple functional profiles. Functional profiles describe in detail the application layer interface of a device, i.e. its expected functionality, which includes its behavioural configurations and network variables. Network variables are datapoints exposed by a device to other devices in the network used to exchange information. Every network variable has a data-type associated that defines units, scaling and structure of the data it contains. Variables can be bound (if they share the same data-type). Network variables usually follow well-defined rules defined in LonWorks Standard Network Variables Types (SNVT) specification [56], guaranteeing interoperability between LonWorks devices. For example, LonWorks specifies variable types responsible for carrying alarm notifications.

An example of functional profiles in action can be a device acting like a switch, implementing the Switch functional profile [57] that exposes an output network variable used to turn on or off other devices, for example, a lamp device. Then, that lamp device implements a Lamp Actuator functional profile [58], exposing an input network variable used to receive input values controlling the lamp's status. By binding

[1] KNX Software Tools – ETS5: `http://www.knx.org/knx-en/software/overview/index.php`

these variables, the switch device will be able to command the lamp device.

Functional profiles can also be a drawback since there is no way that the specification can foresee every need. Therefore it turns out that many manufacturers implement their own proprietary extensions, thus hampering interoperability. An example of a lack of functionality is the absence of network variables tailored for historical data access. However, similarly to other fieldbus standards, LonWorks can have programmable devices which can support such features not covered in the specification, unfortunately, at the cost of interoperability loss [59, p. 5-5,5-6].

LonWorks supports the creation of logical virtual networks within the physical network structure's domains and subnets which can be used to implement zoning. Domains are used to separate large, independent groups of devices in a network, for example, separating lighting systems from HVAC systems. Subnets are physical or logical groups of devices within a domain. For example, a subnet can be a group of all devices of a room. Domains and subnets provide LonWorks with the capabilities of zoning. Inside subnets nodes can be addressed individually or by broadcasting. Another way to associate devices in a way that is independent of domain-subnet-node addressing is by creating a group. Groups are collection of devices, where each member can communicate with others using the group's identification. Groups can be also used for broadcasting messages.

4.3 ZigBee

ZigBee is a standard that enables devices to communicate wirelessly reducing wiring costs and the aesthetical impact of the system's installation. ZigBee's information model is mainly defined across three layers that manage device objects, endpoints and bindings between those endpoints, known as the Application Support Sublayer, ZigBee Device Object and Application Framework layers [32, 40].

The Application Support Sublayer is responsible for binding endpoints, message forwarding between bound devices, and device groups management. Device groups are supported by group addressing where writing to a given address may result in writing to several devices.

The ZigBee Device Object layer is responsible for overall device management, specifically it is responsible for defining (i) the operating mode of the device—telling if a device coordinates network communications or acts as an end device—, (ii) device discovery and determination of which application services the device provides (each application service corresponds to an endpoint of a device), and (iii) for handling binding requests from other devices or coordinators.

Finally the Application Framework is where each device's applications lie. An application object tells what services the device can offer, for example an application object can be a light bulb, a light switch, an LED, or an I/O line. Each ZigBee device can have up to 240 applications, thus 240 endpoints are reserved for this purpose in each device. To communicate with other devices applications make use of specific protocol data units called *clusters* that consist of predefined message structures consisting of commands and attributes, resembling an object in object-oriented Programming context [60, p. 238-239], that both communicating devices are aware of, thus providing a common mean of communication between devices. For example, an on/off cluster defines how to turn something on or off. This cluster is

generic enough for working with light switches, pumps, garage door openers and any other device that can be turned on or off. ZigBee's cluster library provides clusters for managing device groups, scenarios, and alarm notifications [61].

4.4 BACnet

BACnet is a specification developed by ASHRAE [62, 63, 28] that defines a standard way of organizing a BAS system concerning a communication protocol for devices to communicate in a fieldbus network (BACnet IP and BACnet MS/TP) and service API to control the system. Overall, BACnet's defines an information model that organizes the system devices using a standard collection of objects [23]. Objects represent application services along their inputs and outputs, such as devices, calendars and schedules, commands, control loops, among others [64]. In BACnet a device is represented by a *device object*. Device objects define device properties like device model name, device vendor, device status and the list of other BACnet objects associated to the device, for example, a device with two analog inputs will have two instances of analog input objects associated [2, p. 243]. BACnet's model also accommodates proprietary object types for manufacturers to register functionalities not covered by standard objects, at the peril of risking interoperability between devices [65, p. 10].

BACnet supports grouping through a specific *group* object that consists of a list of objects belonging to a desired group and a list of the selected properties of each of those objects. Groups can be used to create logical groups such as zones.

The BACnet's *calendar* object consists of a list of relevant dates (for example public holidays or special events). On the other hand, *schedule* objects associate functions to specific dates, time or date intervals. Functions allow turning on a building's lights after 18:00 every week of January. Schedules can be periodic, if they repeat every week or they can be a one time event [66] [67, p. 41].

BACnet commands are used to write to several attributes of a group of objects simultaneously when invoked. These objects can be devices, calendars, inputs, outputs. BACnet commands can be represented as objects containing a list of actions that can be executed, where each action consists of a list of attributes of other objects to be changed along with their new values. For example, a command object can define a set of attributes to modify when a certain room is unoccupied and another different set of attributes when it is occupied, hence defining two possible scenarios for that room, one called "occupied" and the other called "unoccupied".

Another interesting BACnet feature is the *notification class object* used for distributing event notifications. These objects may require the recipient's acknowledgement and a list of recipients. The subscription of a recipient is maintained by a list of recipient device objects, associated with the notification object.

BACnet neither provides concept of inheritance nor aggregation. Any model extensions must be made based on the creation of new standard objects. This lack of advanced modelling mechanisms difficults data representation, such as sub-typing [3].

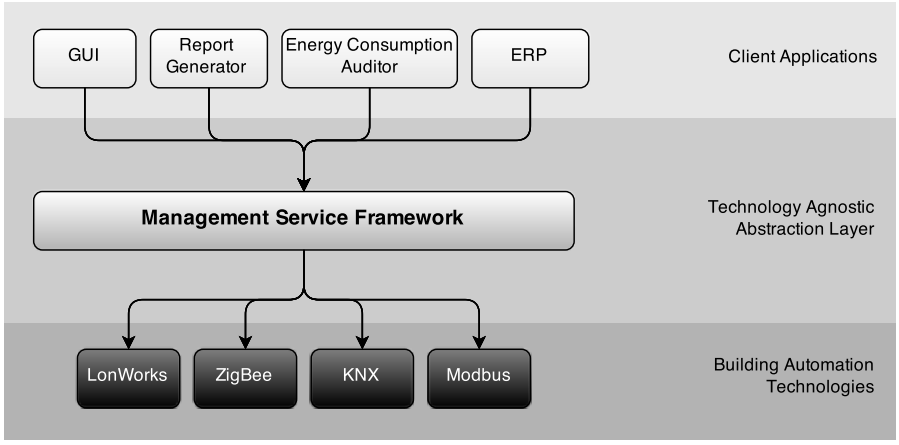

Figure 4.1: Typical topology of a management service framework which abstracts multipurpose client applications from the underlying Building Automation technologies.

4.5 Management Service Frameworks

Service-Oriented Architectures (SOA) or Service Frameworks describe architectural principles and patterns aiming at reducing dependencies between systems to ensure interoperability, sustainability and autonomy, through the abstraction of service [68].

Services are an abstraction of functionality available as a remote procedure call. Service provider applications provide services to consumer applications. One key aspect of most SOAs is that service providers and consumers are loosely coupled. Service providers publish to a service broker server descriptions about their services, specifying each service capabilities and invocation requirements. This service broker server manages a list of available service descriptions published by service providers, that can be requested by consumers. This action is usually known as *service discovery*. Service consumers must adapt to a specific interface in order to establish communication with service providers.

SOAs are most suited to large heterogeneous systems, where each component of the system is implemented by different developers, with different purposes and with different development approaches (such as the programming languages used). SOAs are a frequently used in Building Automation to integrate different technologies, such as devices from different vendors or devices communicating using different protocols. Ultimately a SOA acts as a communication gateway between system devices and client applications.

Services can be reused by other applications that share the same interface guidelines as the service provider. Consider a BAS based on a SOA, where a service framework is responsible for encapsulating the logic of Building Automation and simultaneously for offering its services to enable any external application to interact with the system. These external applications may be graphical user interfaces for the automation system, thus enabling the development of multiple interfaces such as one interface for a desktop computer, other for smartphones, etc. These interfaces are loosely coupled to the system. This eases the task of maintaining the system, because changes in the automation software's internal logic will not affect any of its interfaces, and interfaces can be created or modified without affecting the au-

tomation software. Figure 4.1 illustrates the typical topology found in management service frameworks, which abstract multipurpose client applications from the underlying automation technologies.

In the following sections we study the most commonly used service frameworks in industry to tackle the heterogeneity issue.

4.5.1 BACnet Web Services Layer

The BACnet specification also covers a high-level SOA protocol, concerning a client-server communication protocol oriented to the needs of Building Automation systems, known as BACnet/WS. This protocol provides a WS interface that can be used to communicate with multiple fieldbus networks [69, 70, 62], i.e. it acts technology-agnostic and could, in principle, be deployed to any Building Automation technology.

BACnet/WS services are used for reading and writing data into the servers which can be directly connected to a fieldbus network. These servers are organized by an information model consisting of hierarchically arranged nodes, where nodes are used to map BAS domains. Nodes can only have one parent node, but can have an infinite number of children nodes. Nodes are described by their attributes, like node type, display name, value, units, etc. and these attributes can have nested attributes. Some node attributes are optional others are always required (like the value attribute, for example). BACnet/WS provides two types of nodes: Standard nodes and Reference nodes. Standard nodes contribute to the server's hierarchic model by naming part of the path to each leaf node (where leaf nodes are nodes with no children), i.e. the path to a leaf node is the composition of all standard nodes belonging to that leaf's path. Reference nodes are, as their name suggests, reference to other nodes. These nodes are used to allow other nodes to appear in different places in the same hierarchic model. Each BACnet/WS server instance can only have one hierarchic model, meaning that every concept must be a child of the root node [71, 72].

BACnet/WS defines different types of service, where the most frequently used are: the *options services*, the *read services* and the *write services*. The options services are used to modify the server's behaviour. For example, changing the precision of retrieved values, or the server's default localization (important for localized attributes like multilingual names and the server's time-zone). Read services provide several ways to read values from nodes' attributes, which can go from reading particular attributes, to read an array of attributes with just one request. In contrast, write services provide a way to write values into those attributes, individually or in group.

4.5.2 OPC UA

Object Linking and Embedding for Process Control (OPC) is an industry standard for defining a standardized mechanism to access automation hardware such as sensors, actuators and controllers as well as associated services such as data logging and event notification subscriptions [73]. OPC is based on the idea that each vendor provides OPC-compatible drivers for their network devices[2]. These drivers

[2]OPC introduces a layer of abstraction between data-sources and data-consumers technologies: `http://www.automation.com/library/articles-white-papers/building-automation/introduction-to-opc-for-building-automation`

abstract OPC systems from dealing with the intrinsic specifications used by each vendor's devices. This way every device can communicate through OPC's uniform data representation [23].

Overall, OPC specifies the interaction between OPC servers and OPC clients following a SOA. OPC servers may run on regular computers or in specialized devices [74, p. 2] using application protocols such as SOAP and HTTP, or binary TCP [52].

At its core, OPC conceptual model rests on two generic concepts: *nodes* and *references*. Every OPC object is a node, and every node may have references to other nodes [75] [73, p. 22]. Each node has unique identification and a class type telling the purpose of the node. Nodes may represent objects, types, variables or even methods. Each node has a *display name* which is a human readable localized text describing the object.

References capture relations between nodes. References are associated with a *Reference Type* object that defines their semantics. The most common types of references are inheritance and composition. References between two nodes can be asymmetric implying different roles for each node, such as an inheritance reference, symmetric implying similar roles for both nodes. Reference types may also have references between them, thus supporting sub-typing trough the use of sub-type reference types [73, Fig. 2.5].

OPC defines a meta-model consisting of 8 standard classes of nodes, where the most relevant are: *objects*, *variables* and *methods* [76, p. 82]. Objects are a kind of node used to structure the system's domain, known as address space. They do not contain any information other than the attributes inherited from nodes. Object values and behaviours are exposed using variables and affected by invoking methods. Moreover methods and variables are connected to an object by the use of references. There are two types of objects: *simple* and *complex*. Simple objects are used to define some semantics, usually to organize other nodes; they have no variables nor methods. Complex objects expose some structure of nodes beneath them composed by variables and methods, that are present on each instance of these objects.

For example the OPC specification defines a specific relation type called "HasTypeDefinition" that models the relationship of an object instance to its corresponding object type. Consider, for example, an object instance called "Motor1" that is related to the object type "MotorType", meaning that "Motor1" is an instance of "MotorType", where "MotorType" defines a complex object, consisting of methods (turn the Motor on and off) and internal variables (representing the motor's current speed and temperature). Every object node is related to a node called *object type*, that defines the object's type [76, p. 83]. OPC has some pre-defined object types and new ones can be created.

Objects can also be marked as event sources that can trigger events given certain conditions. Clients can subscribe to this events so they get notifications every time they occur through the subscription services (see Section 4.5.2).

Similarly to objects, variables can also be simple and complex. Simple variable types only define the semantics of a given property, whereas complex variables expose some structure of more variables beneath it. All variables have associated a data-type and a value attribute. The data-type indicates the type of the value hold by a variable. OPC specification defines almost forty standard data-types [77, p.

60], and new ones can be added. Variables can be marked as *historizable*, meaning that the variable's values will be persisted into a log with a defined sampling rate, that can go from the fastest the system can provide, to only logging the value when it suffers a modification. Clients can subscribe to variable's value changes, so that they receive notifications about a variable's progress (see Section 4.5.2).

Although OPC is flexible enough to model different domains other than Building Automation [78], it is also complex, therefore its specification may be hard to understand, extend and support. Due to this complexity, there is a lack of updated free-software and development tools [3, 79]. Moreover, many tools are unable to implement the entire specification[3].

The OPC UA specification establishes a SOA describing how OPC servers interact with OPC clients by means of services [80, 81].

OPC UA defines an extensive list of service sets [77], where the more relevant are the: (i) discovery services, (ii) node management services, (iii) view manipulation services, (iv) querying and attribute services, (v) method services, (vi) monitored items and subscription services.

The discovery service is used by OPC clients to find available OPC servers and to provide their endpoints' descriptions. A service endpoint is an entity to which service messages can be addressed (an example of an endpoint can be an application that listens to a certain port in the server). The endpoint description conveys the information required to address a service endpoint by clients. OPC discovery services can be invoked over the network, where the client may request information concerning connected servers to a Discovery Server where each server registers its services and respective endpoints' descriptions [73, p. 131-134].

Node management services define ways to manage nodes and relations from a server address space. Having the correct permissions clients may modify an OPC server's node-model without interrupting the system, thus reducing maintenance costs. This is useful when new aspects of a BAS must be considered. For example, a BAS that was confined to manage only one building but now there's the need to manage more buildings with that system, thus forcing the creation of new concepts in the system's model.

Views services enable navigating the server's model, filtering the unnecessary information. Current OPC documentation doesn't provide official services to add new views.

Querying services are used by clients to find information in models. Queries return a list of nodes and attribute values, based on filtering criteria executed over the whole address-space of a server [73, p. 131,187]. Clients can also query historical data regarding node attributes.

Method services are used to invoke methods from queried objects of the model. Methods can receive and return several parameters. Some methods can have invocation conditions that, when not fulfilled, may abort the invocation request. This protects the system against repeated invocations made in a short period of time, that could otherwise disrupt its normal operation. Requests must wait until the invoked methods finish their execution. Therefore method calls may vary in response time.

Monitored items service enable clients to subscribe to data changes and events of objects and variables. A variable or object subscribed by clients is called a monitored item. Subscriptions have associ-

[3]Free OPC servers, `http://www.opcconnect.com/freesrv.php`

ated a sampling rate. Item monitorization can be disabled or enabled by the client at any moment.

Each subscription has a publishing interval that defines the cyclic rate (independent from monitored items' sampling rate) at which the subscription will check for new notifications to send. These notifications describe monitored items' value changes or events. Events are triggers associated with conditions. These conditions are associated with objects that typically represent devices. These objects are treated as event notifiers, to which a client can subscribe to. Other nodes can relate to event notifier objects and by consequence, act as event notifiers of that object themselves. This is very useful for dividing complex systems, when they are very prone to generate events, into small sub-systems, each one having condition events associated; then the client just needs to subscribe to the root node of that system. Each time any sub-system triggers an event, the client will be notified and can easily detect the source of that event [82, p. 50-53].

4.5.3 oBIX

oBIX is a platform independent model designed to promote device interoperability through web-services [13]. oBIX implements a SOA with three services used to read and write data, and to perform procedure calls (invoking operations, in oBIX terminology). The read service is supported by every object in oBIX, write is only supported by writeable objects and invoke is only supported by oBIX operations. The inputs and outputs of each service request are specific to the target object or operation.

The oBIX information model consists of typed objects, described by attributes. These objects are identified by their name and a Unified Resource Locator (URL) used to describe the object. oBIX models can be extended through object composition and inheritance.

The oBIX specification describes a mapping into XML, where each object corresponds to exactly one XML element and attributes are represented as that element's XML attribute. The oBIX specification also defines default objects, that correspond to the XML's primitive element types such as string, real and boolean. Some primitive elements have additional attributes such as minimum and maximum values, units and precision. There is no provision for devices, zones, groups or controllers in oBIX's specification, these concepts must be manually defined. oBIX defines history records, events, and alarms services.

The history record service consists of an interface that when implemented by any oBIX object, turns this object into a historical (traceable) object whose attributes will be stored. This interface exposes properties such as the maximum number of history records to maintain for an object, the timestamps of the first and last records, among others. To manage events, oBIX provides a *watch* object, that implements a cache for storing events where clients can register several object attributes whose changes are to be tracked by the watch object every time they happen. Alarms are objects capable of generating events when some conditions are met. Usually these conditions consist of some object's attribute whose value just went out of its defined bounds (for example an object with an attribute called "temperature" whose bounds are 0 and 70, in which its current value is less than 0 or greater than 70). Alarms have to be associated to watch objects in order to keep a track of recent alarms and send them to the client each time it polls the server. oBIX specifies services to query, watch, and acknowledge alarms.

	Building Automation Technologies				Service Frameworks		
	BACnet	KNX	LonWorks	ZigBee	BACnet/WS	OPC UA	oBIX
Grouping/Zoning	●	●	●	●	●	◐	●
Event Notification	●	○	◐	○	●	●	●
Alarm Notification	●	○	◐	●	●	●	●
Historical Data Access	◐	○	○	○	○	●	●
Scheduling	●	○	●	○	○	○	○
Scenarios	●	●	●	●	○	○	○

Table 4.1: Comparison of functional aspects of the studied technologies. Legend: ○ *low* or *no* support ◐ *medium* or *partial* support ● *high* or *full* support.

4.6 Discussion

The previous sections analyze several technologies in order to comprehend how their information models address the BAS concepts and features introduced in this article. Tables 4.1, and B.1 summarize the analysis made concerning these technologies organized in terms of functional aspects covered by their official specifications.

Some technologies are better suited for specific BAS levels than others, for example, OPC UA, oBIX and BACnet/WS are tailored for the management level, since they provide a SOA whose services aim at managing BAS. LonWorks, KNX, ZigBee and BACnet communication protocols are more suited for control operations. Although these particular protocols' functionalities can be extended by the use of programmable devices, such as adding task scheduling and alarm notification capabilities [59, p. 5-5,5-6], it leaves those features implementation's responsibility in manufacturers hands which usually end up impairing interoperability between devices from different vendors by developing closed proprietary implementations.

As BACnet, KNX, ZigBee and LonWorks target the control level, i.e. they are directed to network communications, end up being quite rigid in the sense their specifications do not provide specific ways to develop new concepts, namely high-level concepts such as scenarios or scheduling, over these protocols. In contrast, OPC, BACnet/WS and oBIX are designed to be relatively extensible and capable of representing complex domains with several relations and definitions, likely to have more applications than just in the BAS domain [78]. Although some authors describe BACnet/WS's model as not extensible due to its simplicity [72]. Another problem arising from these specifications is that they end up being too generic and too complex to model simple domains such as the Home Automation context, which do not require all the overhead that, for example, OPC provides.

LonWorks, BACnet, ZigBee and KNX have, in practice, low interoperability, because manufacturers' devices can only interoperate if they implement their functionalities according to the same guidelines, which is not always possible due to standards' limitations. One example is the implementation of the proprietary TAC's network variables over the LonTalk protocol [83], which consists of a proprietary custom extension to the original LonWorks specification. On the other hand, OPC, BACnet/WS and oBIX were designed to target interoperability between systems by defining a common information model and a service abstraction layer, but which still misses some basic functionalities pointed out in Table 3.1.

In the end, as pointed out by our analysis results, there is no single technology which can be used to

Figure 4.2: Coverage of functional layers by distinct BAS and service framework technology specifications (adapted from [3]).

fully satisfy the functional aspects of all levels in a BAS without complementing their specifications with posterior, custom-made, feature developments, practiced by device manufacturers and developers [3] as we can conclude from Figure 4.2. Apparently, BACnet is the most complete solution, since it covers all three functionality layers.

A direct consequence of the shortcomings of most technologies is that, to get all the desired functionality in a complex infrastructure, more than one technology is required, and adding to this, the integration of such technologies can easily become a very complex undertaking [23, 52, 84]. Moreover, most solutions only provide their specifications' documents leaving their implementations in the developers hands. Although, some fieldbus standards already provide low-level libraries that implement their protocol stacks such as OpenLDV for LonTalk [85], and Falcon[4] or Calimero[5] for KNX. At the management level, only oBIX[6] and BACnet/WS[7] have up-to-date and freely available implementations of their specifications. During this study, no freely available and complete implementations of OPC UA's specification were found.

[4]KNX Falcon, `http://www.knx.org/knx-en/software/falcon/about/index.php`
[5]Calimero KNXnet/IP, `http://calimero.sourceforge.net/`
[6]oBIX Tools, `https://code.google.com/p/c-obix-tools/`
[7]BACnet Server, `http://bacnetserver.sourceforge.net/`

Chapter 5

Technology Integration Frameworks

Technology Integration Frameworks are software solutions that aim at solving the previously analyzed heterogeneity issue inherent to the automation industry. In contrast with Management Service Frameworks, where manufacturers have the responsibility of making their devices compatible with a given framework, or having network gateways that provide the necessary interfaces to bridge incompatible devices, Technology Integration Frameworks try to adapt themselves into incorporating already existing devices. The advantage of using an integration framework is that, in principle, every device can be accommodated, given the necessary driver implementations.

Most of these solutions target more problems than just heterogeneity, such as building modeling or ontology definitions, but for the sake of simplicity we will give especial attention to how their architecture corroborates with our concerns. During this analysis we will evaluate several architectural quality aspects, through the use of known metrics, which will tell us how easily can such architecture be understood, reused and extended, and we will also verify how complex is the task of integrating new automation technologies using these solutions. For this analysis we have considered the two most widely used open-source platforms in industry: *Domotic OSGi Gateway* and *openHAB*.

5.1 Domotic OSGi Gateway

The Domotic OSGi Gateway (DOG)[1] is a solution that exposes different automation networks as a single, technology neutral, Home Automation system [86].

DOG architecture is built upon the OSGi framework (described in Section A) and is organized in 4 layers, each dealing with different tasks and goals, ranging from low-level interconnection issues to high-level modeling and interfacing (Figure 5.1). The version analyzed was 3.1.0.

Layer 1 includes the DOG common library and the OSGi bundles necessary to control and manage interactions between the OSGi platform and the other DOG bundles. At this level, system events related to run-time configurations, errors or failures, are generated and forwarded to the entire DOG platform.

[1] DOG Official Website: `http://dog-gateway.github.io/`

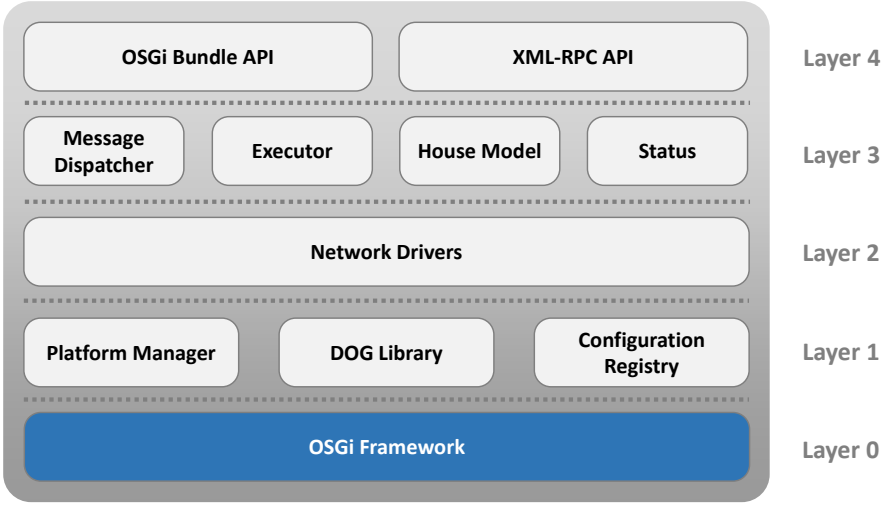

Figure 5.1: DOG high-level architecture proposed by its specification organized into layers (adapted from the official specification).

Layer 2 encompasses the DOG bundles that provide an interface to the various automation networks to which DOG can be connected. Each network technology is managed by a dedicated driver, similar to device drivers in operating systems, which abstract network-specific protocols into a common, high-level representation that allows to uniformly drive different devices. Layer 3 provides the routing infrastructure for messages traveling across network drivers and directed DOG bundles. Layer 3 also hosts the core intelligence of DOG, based on an abstract formal model of the building environment, that is implemented in the House Model bundle. Finally, Layer 4 hosts the DOG bundles offering access to external applications, either by means of an API bundle for OSGi applications, or by an XML-RPC endpoint[2] for applications based on other technologies. Throughout the rest of this section we will focus on Layer 2, and consequently on Layer 1, which is the layer targeting the heterogeneity issue that we are trying to solve in our work.

While analyzing the project source code we became aware that dependencies between components are not as evenly distributed as the DOG specification tries to transmit, i.e. it is not clear which components belong to each functional layer as shown in Figure 5.2, making it difficult to replace a layer implementation without performing deep modifications on the other dependent layers. Adding to this, DOG does not present any mechanism of testing the functionality of each layer individually nor any component at all. So any small change to DOG's architecture imposes a colossal risk of disrupting the correct operation of the overall solution.

Further in the analysis, we were unable to find any references showing us that DOG is minimally aligned with the concepts defined in previous sections of this article, which are based in known industry standards. More specifically, we could not find any representation of a datapoint, a network address and a gateway. This means that such concepts must be later defined by the person who needs to enrich DOG framework with a new network driver. Since there are no guidelines for fabricating drivers in DOG,

[2]However, a REST API has been released which, apparently, wasn't part of the initial specification: `http://dog-gateway.github.io/build-on-it.html`

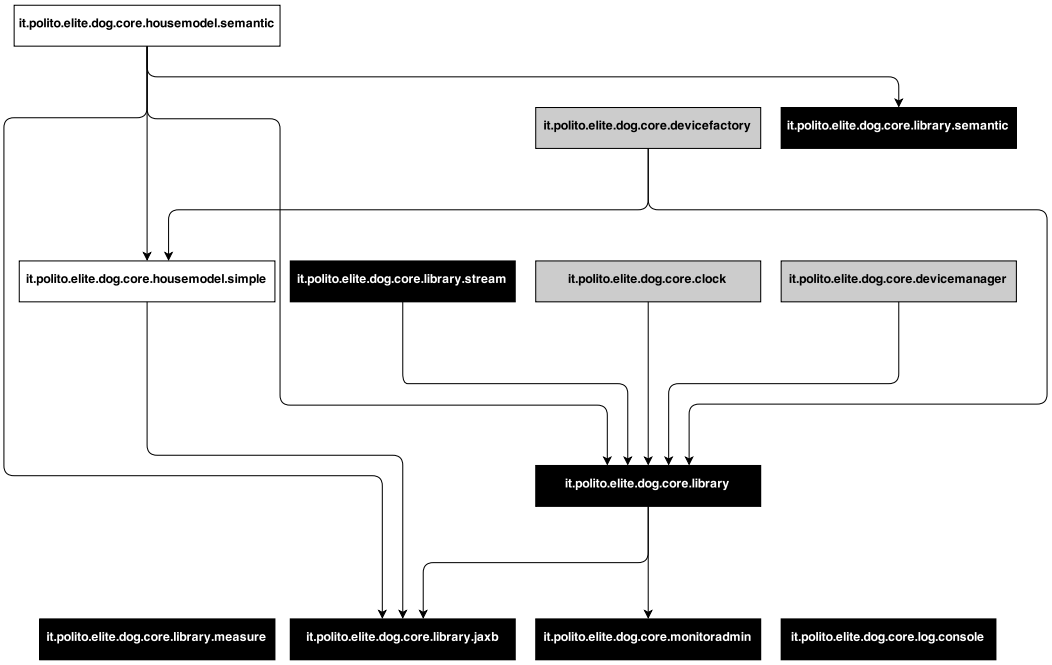

Figure 5.2: DOG core components diagram with dependencies. The black squares represent Layer 1 components, the grey squares represent the Layer 2 components and the white squares represent the Layer 3 components, according to our interpretation of Figure 5.1. The black arrows depict dependencies from one component to another.

each driver will have a distinct structure and a certain degree of subjectivity to the developer.

Contrarily to what is expected in a solution which is advertised as being *technologically neutral*, we have found that it contains core devices that should be generic and technology-agnostic (such as *DogSwitchOnOff* represents a generic ON/OFF switch) but are in fact attached to vendor-specific technologies and terminologies. Examples of some of these devices are *DogEchelonIlon100Gateway*, *DogZigBeeGateway*, *DogTexasInstrumentsGateway* and *DogZWaveGateway*, which could be abstracted in one single device called *DogGateway*, which does not exist. Since there are several classes from where a new driver can inherit, like we can conclude from Figure 5.4, developers are left with too many choices to make relating the implementation of drivers, which is also a source of concern [87].

DOG's device model is defined as shown in Figure 5.3. One major design issue detected by looking at the class diagram is that domain-specific definitions, such as *HousePlants*, *ElectricalSystem* and *Controllable*, are directly encoded into the model. Meaning that if the user desires to use this solution in a different context, for example in a mall, he will have to define new concepts (for example *MallPlants*) and remove unused ones, followed by full code compilation. Moreover, using class inheritance as a mean of categorizing domain-specific concepts, increases the depth of inheritance, which is an issue already covered in Section 7.1. On the other hand we have abstract concepts, such as the *DeviceDescriptor*, which rely on JAXB classes that are implementation specific. Namely, these JAXB classes are responsible for representing Java objects in XML so that Layer 4 can serve these representations in its exposed XML-RPC API. Hence, the task of representing domain objects into XML should be fully covered in Layer

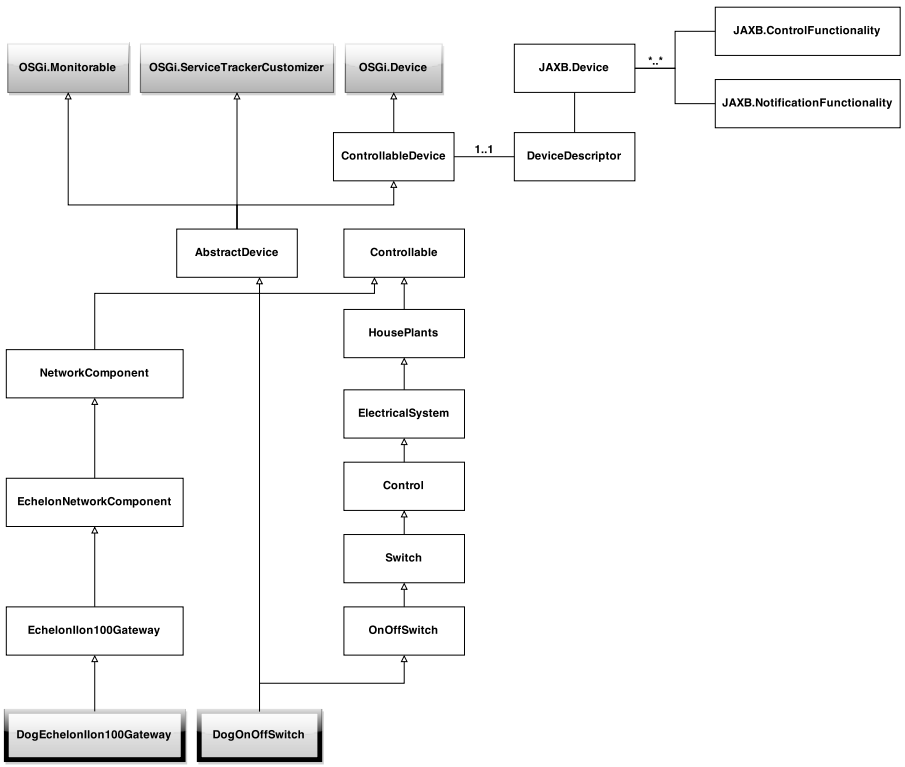

Figure 5.3: DOG's device class diagram. Classes from third-party libraries are in grey, DOG abstractions and interfaces are in white, and in bold we have two examples of core devices defined in DOG.

4, and avoid making lower layer's architectural aspects depend on upper layers responsibilities [88, p. 93].

As a consequence of the complexity behind the device model, extended device drivers such as the KNX ON/OFF device exemplified in Figure 5.4 end up being too complex, and the developer implementing the device must assume that domain-specific classes like the ones mentioned earlier exist. This means that if the running instance of DOG does not contain concepts such as *HousePlants*, this KNX device driver is rendered useless. Adding to this, device driver developers are kept the responsibility of categorizing their driver correctly, resulting in a subjective selection of categories for each device drivers. For this specific KNX ON/OFF driver it was categorized as being a *SimpleLamp*, a *Lamp*, a *MainsPowerOutlet* and a *Buzzer*. To aggravate things even further, due to the class-based categorization system, this KNX ON/OFF driver has at least 5 direct dependencies, which is a bad practice as explained in Section 7.1. Finally it is not clear why some device drivers inherit from classes other than the *AbstractDevice*, which suggests that there are multiple ways to implement new drivers, resulting in an anti-pattern [89].

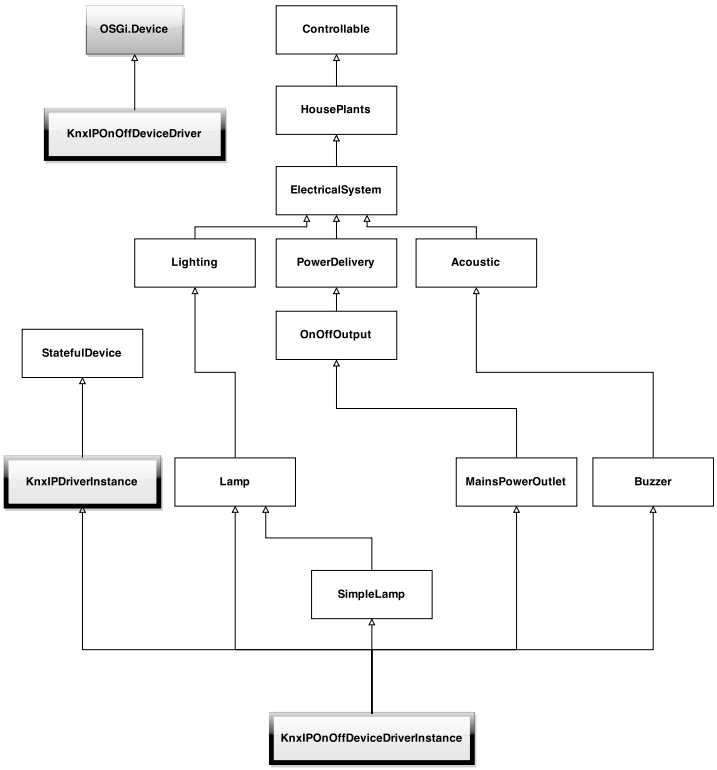

Figure 5.4: Official ON/OFF KNX driver implementation for DOG Framework. Classes from third-party libraries are in grey, DOG abstractions and interfaces are in white, and in bold we have the ON/OFF driver definitions.

REST Endpoint	Description
/devices	Represents domotic devices handled by DOG and ”controllable” by applications using this API.
/devices/{device-id}	Represents a single domotic device handled by DOG, identified by a unique device-id (currently encoded in the id attribute for the XML response to the GET /devices request), and ”controllable” by applications using this API.
/devices/{device-id}/location	Update the location of a single domotic device handled by DOG, identified by a unique device-id.
/devices/{device-id}/description	Update the description (i.e., the long name) of a single domotic device handled by DOG, identified by a unique device-id.
/devices/status	Represents the status of devices registered in the DOG gateway runtime, i.e., defined in the DOG configuration and successfully registered within the gateway runtime.
/devices/{device-id}/status	Represents the status of the device identified by the given device-id, registered in the DOG gateway runtime, i.e., defined in the DOG configuration and successfully registered within the gateway runtime.
/devices/{device-id}/commands/{command-name}	Represents a command, identified by a command-name, to be sent to the device identified by the given device-id. Commands are idempotent: the same command always results in the same behavior of the selected device. If the command brings the device in same state in which the device is, no differences will be appreciable.
/dog/configuration	Unsupported, to be implemented in future.
/environment	Represents the environment (i.e., the building) configured in DOG.
/environment/flats	Represents all the flats present in the environment (i.e., the building).
/environment/flats/{flat-id}	Represents a specific flat present in the environment (i.e., the building).
/environment/flats/{flat-id}/rooms	Represents all the rooms present in a given flat.
/environment/flats/{flat-id}/rooms/{room-id}	Represents a specific room present in a given flat in the environment (i.e., the building).
/rules/	Represents the rules registered in DOG. By using this resource, it is possible to get all the existing rules or add a new rule.
/rules/{rule-id}	Represents a single rule registered in DOG. By using this resource, it is possible to update or delete an existing rule.

Table 5.1: Official specification of the REST endpoints exposed by DOG to external applications. The words between brackets represent the input parameters of each endpoint.

5.2 OpenHAB

Similarly to DOG, openHAB[3] is presented as a vendor and technology-agnostic open source Home Automation software. The openHAB concept is divided in several modules, as depicted in Figure 5.5, where the core functionalities, such as network integration, and their direct dependencies can be found in the *openHAB Core* and *openHAB Base Library*.

The core concepts of openHAB are *items* and *bindings*. An item is a representation of a network datapoint that is usually associated with a binding. Items do not worry whether they carry information concerning a given data-type nor the source of that information. Items can provide means to get readings from physical sensors or from a resulting calculation from a mathematical expression, or from a table in a database, among others. The item abstraction detaches the upper layers from any reference to device specific characteristics, such as vendor-specific data-types. Conversely, bindings are the drivers which integrate the different solutions into openHAB's architecture. At the time of this writing openHAB is shipped with 95 bindings.

In contrast with DOG's architecture, openHAB is completely technology independent and does not blend domain-specific semantics, such as house plants, with the network concepts (see Figures 5.6 and 5.7). Moreover it provides a more clear interface from where device drivers can arise, whereas in DOG it is not clear whether new devices should inherit from classes such as *StatefulDevice* or *AbstractDevice*. However, the binding implementation task provides too much free-will to the developer, in the sense that each binding must re-implement certain functionalities which could be offered by the core architecture, increasing code re-usability. Among these functionalities are message schedulers, connection status listeners and driver configuration utilities.

Throughout our analysis we have noticed that each binding can be divided in two drivers, one transport driver and one gateway driver. The transport driver takes care of abstracting the gateway driver from the physical means of protocol message transportation, for example a TCP/IP or RS232 transport driver. The gateway driver would be responsible for exchanging protocol messages with the gateway connected to the network. In the current state of affairs, each time a developer needs to implement a new binding, he must implement both the transport driver and the gateway driver. For example, the TCP/IP transport driver from the KNX binding cannot be used by other bindings to communicate through TCP/IP. This leads to code repetition, paving the way for recurrent programming errors [90].

Another aspect hampering its future efforts in developing on openHAB is the remarkably extensive task of setting up the development environment to the needs of this project, so that one can develop additional features for openhab[4]. Moreover, the openHAB project promotes the integrated development environment (IDE) Eclipse, as it is the only IDE where openHAB can be developed on without the developer having to resolve all its third-party dependencies manually. So if somehow the developer is unable to use Eclipse, developing new features on openHAB renders itself an unpractical undertaking.

As in DOG's REST API, openHAB offers more services than just simple datapoint listing, read and writing by letting bindings extend the API. Moreover, openHAB's API can be extended, which can in-

[3]OpenHAB Official Website: `http://www.openhab.org/`
[4]Official setup instructions for the openHAB IDE: `https://github.com/openhab/openhab/wiki/IDE-Setup`

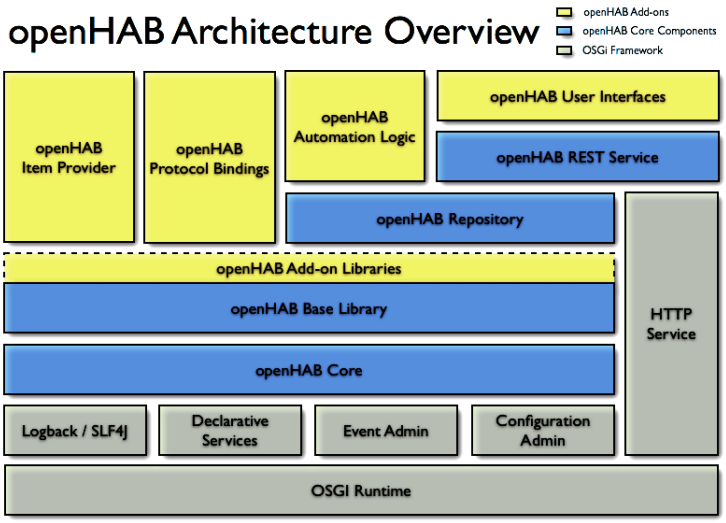

Figure 5.5: OpenHAB high-level architecture displaying add-ons, core components and the OSGi framework (adapted from the official specification).

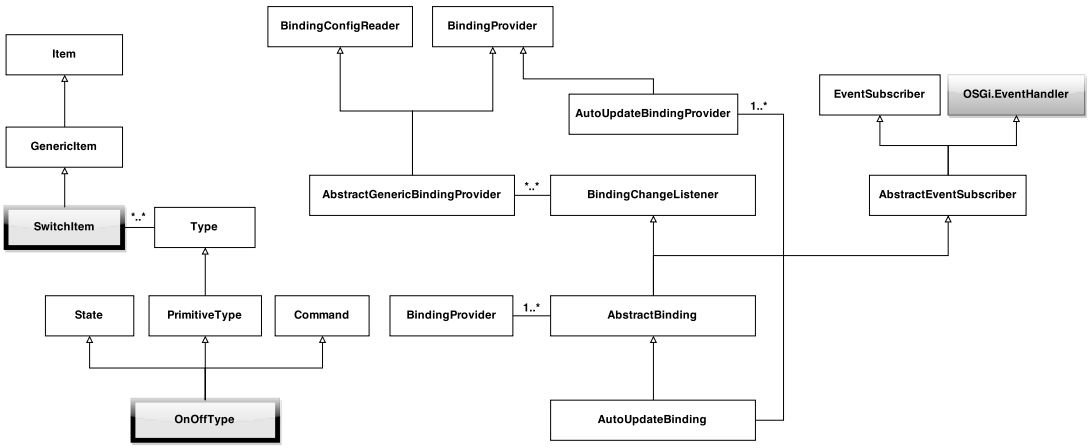

Figure 5.6: OpenHAB bindings class diagram. Classes from third-party libraries are in grey, openHAB concepts are in white, and in bold we have two examples of particular item and data-type definitions.

crease complexity and push the heterogeneity issues upwards to client applications that may have to deal with an heterogeneous REST API. Also, XML is the chosen data load format, lowering the openHAB server performance when tremendous amounts of information is requested, for example, when asking all the readings from a meter in the last month [91]. OpenHAB's REST endpoints are summarized in Table 5.2.

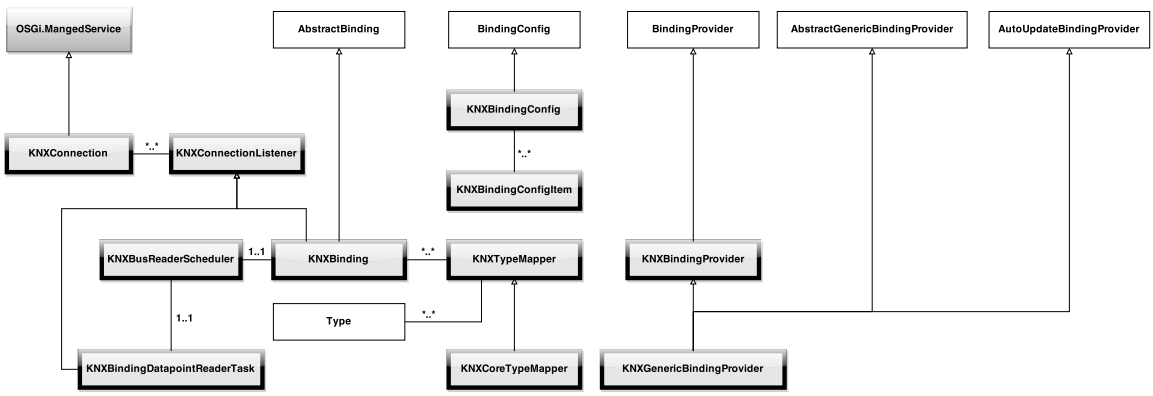

Figure 5.7: Official KNX binding implementation for openHAB Framework. Classes from third-party libraries are in grey, openHAB abstractions and interfaces are in white, and in bold we have the KNX binding definitions.

REST Endpoint	Description
/rest	Lists the available top-level endpoints exposed by extensions. The default endpoints are explained in the following rows.
/rest/items	Returns a collection of all declared items (datapoints).
/rest/items/{item-id}	Access a single item description. An HTTP POST can be used to send a command.
/rest/items/{item-id}/state	Access a single item's current value.
/rest/sitemaps	Returns a list of zones.
/rest/sitemaps/{zone-id}	Returns the list of items inside a single zone, known as a sitemap. A zone not only contains the static information that the user has configured in the system, but it also holds derived data like icons and labels and this group's contents, known as a page.
/rest/sitemaps/{zone-id}/{page-id}	Returns the derived data for this group.

Table 5.2: Specification of the REST endpoints exposed by openHAB to external applications. The words between brackets represent the input parameters of each endpoint.

5.3 Discussion

In this section we analyzed two of the most prominent solutions that tackle the heterogeneity problem inherent to the automation industry. More than simply integrating fieldbus technologies, these solutions try to provide higher layers of abstraction by providing domotic related semantics to the networked devices. Although these solutions work remarkably well in the Home Automation context our analysis uncovered a number of issues, in particular:

1. **Lack of documentation**

   At the time of this writing there is lack of documentation pertaining the driver development for both frameworks. From a practical point of view, we cannot tell, by analyzing different drivers, which is the correct way of implementing one driver since developers are left with an extensive list of choices they can make. This results in heterogeneous driver architectures, hampering components re-usability.

2. **Terminology discrepancy**

   We have established in Chapter 2 that there already exists some terminology discrepancy in the

field of Building Automation. However the analyzed solutions provide us with even more terminology such as *bindings*, *sitemaps*, *pages* and *items*, which could be simply named *devices*, *zones*, *groups* and *datapoints* respectively. Not following the standard terminologies is, by its own nature, an integration issue.

3. **Context specific semantics**
   Both analyzed technologies introduce the usage of context specific semantics, such as *HousePlants* or *flats* indicating that these solutions where specifically designed for Home Automation. Where the complexity is much lower than in large infrastructures.

4. **Code re-usability**
   OSGi is the core of both DOG and openHAB frameworks, which is natively built on the philosophy of code re-utilization. However both frameworks could explore this feature even further by providing a clear separation of a driver's responsibilities, such as separating drivers into gateway drivers and transport drivers. This benefits new driver developers that do not need to, for example, re-implement the transport layer for each developed driver, promoting re-usability.

5. **Complex architectures**
   Both solutions target more objectives than just network integration, such as providing high-level Home Automation semantics, we would need to undertake the task of modifying and recompiling the core architectures in case we want to apply them to different usage contexts. However this task is complex to accomplish in large architectures, specially in the absence of unitary tests validating that our modifications are not interfering with the existing functionality, as in the case of DOG.

In sum, both analyzed solutions fall short when their architectures need to be extended with a new network driver or the user tries to apply them to domains typically more complex than Home Automation, such as industrial or building automation, were developers must perform core modifications to the architecture or the underlying information model. For these reasons we believe that there is a strong urge for a solution that (i) is capable of being easily applied to a broader range of scenarios without performing critical modifications, (ii) encourages component re-usability, consequently (iii) having simpler architecture, with a reduced number of domain objects and third-party dependencies, thus easier to extend and maintain, and (iv) guides developers throughout the driver implementation process.

Chapter 6

Solution Proposal

During the comprehensive analysis of BAS technologies and integration frameworks performed in previous Chapters, we were faced with the fact that buildings are often pushed to acquire solutions from different manufacturers, typically resorting to different network technologies—promoting heterogeneity. In addition, it is shown that there simply are no technology integration frameworks capable of satisfying infrastructures of considerable size without core modifications. This results in having a building manager coordinating several different technologies from distinct, non-centralized control points, i.e. the hardware integration is performed manually. Moreover, this hinders the efforts of applying automated energy efficient behaviors into the building due to the fact that devices cannot interoperate correctly.

Motivated by these issues, in this Chapter we introduce an OSGi based, technology-agnostic, integration framework, presenting itself as a hardware abstraction layer, capable of integrating different automation technologies. The aim of this framework is to abstract BA application developers from hardware intricacies, enabling the creation of software solutions which are able to take full advantage of the infrastructure's installed resources.

6.1 Domain Model

Our domain model consists of an ontology revolving around the concept of *datapoint* as depicted in Figure 6.1. A datapoint holds information about a network resource, which can support read or write operations. Additionally we can read cached values from a datapoint. Every datapoint is linked to one *datapoint provider*, which can be a *gateway* or a *device*. This means that, for example, writing to a datapoint will result in a write request command sent to the gateway providing that datapoint.

A datapoint object holds metadata concerning the access type that the datapoint provides, the datatype that can be read or written, its sampling rate, the associated hysteresis, the connection status (offline, online or unknown), the cache size, a human-readable description, and a flag indicating if this datapoint is virtual. These concepts are aligned with the international standards overviewed in previous

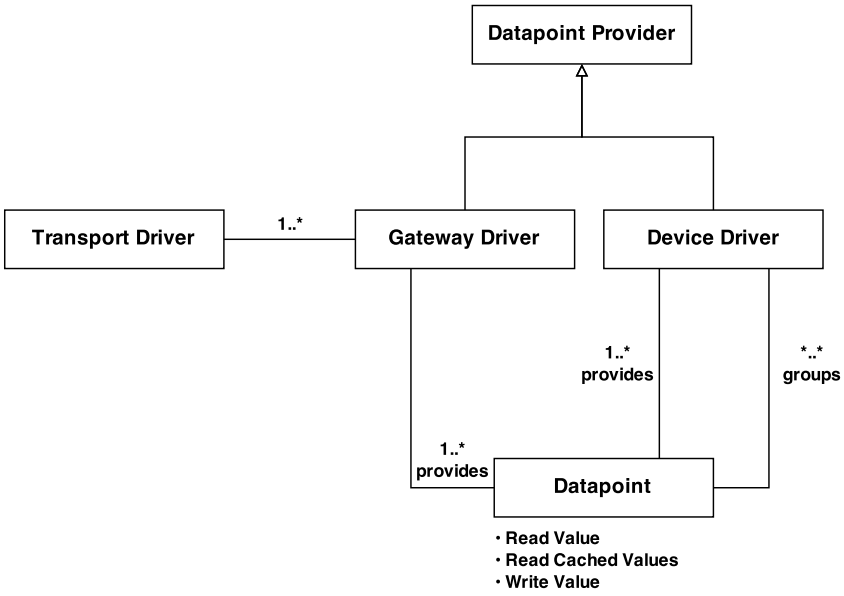

Figure 6.1: Overview on the proposed domain model. Multiple datapoints can be provided by datapoint providers, gateway and device drivers. A gateway driver makes use of one transport driver, which can be re-used by other gateway drivers. A device driver groups datapoints from other datapoint providers such as gateway drivers.

sections.

On the other hand we have gateway objects, which have three main responsibilities: (i) overall, to communicate with the network gateway using the proper communication protocol, (ii) to discover and expose the datapoints provided by that given gateway, (iii) to maintain the status of each datapoint, and (iv) fully abstract device drivers from protocol specificities.

The gateway object holds the knowledge concerning the protocol messages to exchange with the network gateway. However it uses a transport driver in order to establish a connection link. This is useful to cover situations where multiple gateways use the same protocol messages, differing just in the connection link they exchange their messages on. For example, Cooper Controls[1] provide multiple gateways, all sharing protocol similarities, differentiating only in the way they can be connected, some use USB connections, others use RS232 or even TCP/IP. This means that the same gateway driver can be re-used with several different gateways sharing the same protocol messages.

Finally, we have device drivers, consisting of a group of datapoints and control functionality to drive those datapoints, exposing these capabilities through another group of datapoints. For example in Figure 6.2, a RGB lamp device could be associated with three datapoints, representing the blue, red and green colors, and then exposing a single datapoint providing a simpler color selection interface for upper layer applications. Device drivers do not need to know any communication protocol, just the data-type of the datapoints they will group. So the RGB lamp device driver could, for instance, expect three datapoints having an *integer* data-type ranging from 0 to 255. Such level of abstraction enables the same device driver to control any RGB lamp device exposing three datapoints ranging from 0 to 255,

[1] Cooper Controls iLight Solution: `http://www.ilight.co.uk/`

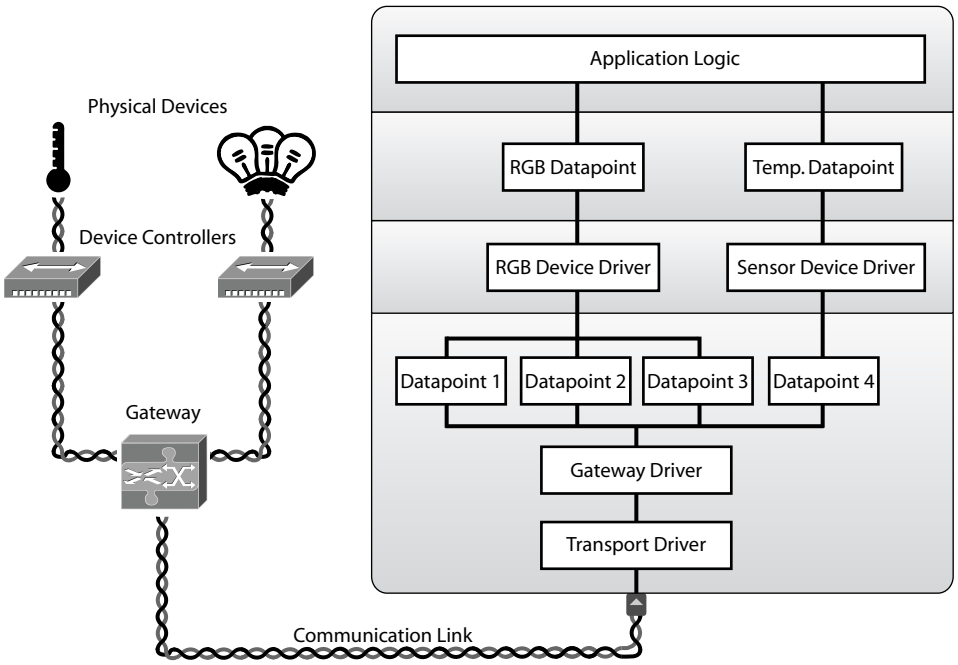

Figure 6.2: Illustration of how the proposed solution will abstract the hardware stack from high-level applications. In this particular example, an RGB lamp and temperature sensor devices are abstracted. Notice the resemblance with Figure 2.5.

disregarding if that lamp is connected to a KNX, ZigBee or LonWorks network.

With this domain model we can abstract every high-level application from protocol and network specificities without compromising our solution to any specific realm of application such as Home Automation and Building Automation. In contrast with the solutions analyzed in Chapter 5, all our domain concepts are based on the terminology defined in well known international standards. Moreover, we intentionally avoided modeling high-level concepts such as Zones, Scenarios, Schedules, and others, because by only relying on networking concepts our framework can be used in a plethora of control applications such as robotics and vehicular automation. By not following the *one-size-fits-all* approach used in previously analyzed frameworks we are avoiding users and developers from dealing with concepts which are unnecessary and eventually will not fit their needs. It is left to higher-level applications the responsibility of defining their domain-specific concepts.

6.2 Architecture

The architecture of our solution is divided in four functional layers: *Transport*, *Gateways*, *Drivers* and *Application* as depicted in Figure 6.3.

The Transport Layer is the lowest layer in the network chain. This layer contains all transport drivers installed which serve as a bridge responsible for sending the byte messages coming from the upper layer to the output stream. The output stream can be a RS232 or an USB port.

The Gateways Layer contains all the installed gateways for each supported protocol. Datapoint

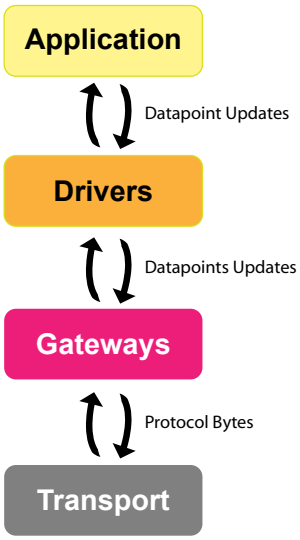

Figure 6.3: Proposed solution's functional layers.

operations from the above layer are converted into specific gateway protocol message bytes. Bytes received from the network gateway are parsed into appropriated datapoint readings.

The Drivers Layer exposes simplified interfaces through datapoints to the Application Layer, creating an abstraction between the Application and the inherent fieldbus protocols. Device Drivers are used to group one or more generic datapoints exposed by gateways.

Finally, the Application Layer is where the application control logic features lie. These applications make use of the datapoints exposed by device drivers to interact with the system. Examples of applications are user interfaces, a service providers (such as Web Services) or controller applications (to manage buildings or robots). This layer is abstracted from every device's protocols and specificities.

Moreover, one of our main goals enable developers to further extend our solution. Developers can add new capabilities to the system such as new device, gateway and transport drivers, datapoint registries, datapoint data-types, and datapoint status trackers. Additionally new functionalities can be built atop of the solution. For example, in the case studies covered in Section 7.2, our solution is provided with two additions: a configuration loader and a REST API. The former stores in disk all configurations concerning existing gateways and devices, and loads them every time the system boots up, the latter enables the access to all exposed datapoints for external applications.

Our solution, depicted in Figure 6.4, is separated in six functional layers: The first layer encompasses the transport drivers, which are then connected to the upper layer's gateway drivers at execution time. The third layer consists of an API exposing all datapoints provided by gateways. The fourth layer holds the device drivers which orchestrate the datapoints exposed by gateway drivers and provide a simpler control interface to the upper layer which exposes all device datapoints. The sixth layer abstracts third-party applications from low-level domain concepts such as gateways and transport drivers, while exposing the datapoints provided by device drivers, disregarding the underlying automation technologies. Atop of our solution, we have the third-party applications, which consume our Java or REST API,

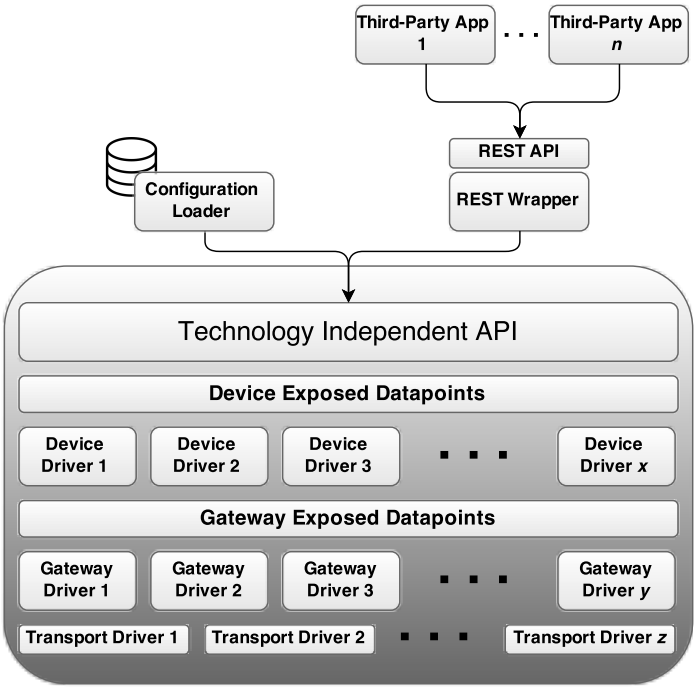

Figure 6.4: Proposed solution architecture consisting of six functional layers.

depending on whether they are local or remote applications respectively, which enrich our solution with further, domain-specific, capabilities such as energy or facility management features.

6.2.1 Transport Drivers

A Transport Driver possesses several responsibilities which go from reading from and writing to the network, to maintain a connection available when possible (opening and closing each connection properly if necessary), report error situations, provide means for configure the connection, and notify the arrival of new messages. The concept of Transport Driver is defined in the interface class *ITransportDriver*, whose methods are highlighted in Table 6.2.

Fortunately, for developers implementing new transport drivers, most of these responsibilities are already considered in the class *AbstractTransportDriver*, leaving to developers only need of implementing the methods which directly relate to the communication link, such as read and write operations. Abstract classes promote implementation simplicity and driver similarity, i.e., all Transport Drivers are implemented the same way, like Figure 6.5 illustrates. For example, there is no *read* method exposed by the Transport Driver since network messages are received through notifications, i.e. read and write interactions with Transport Drivers are asynchronous, hence avoiding any application from actively hanging while waiting for a response. The underlying *AbstractTransportDriver* implementation already provides mechanisms informing message listeners of new message arrivals, homogenizing the way incoming messages are handled in every Transport Driver. Table 6.1 summarizes all capabilities natively present in every Transport Driver—implemented by the abstract class—, which the developer does not need to implement in new drivers, however they can be optionally extended.

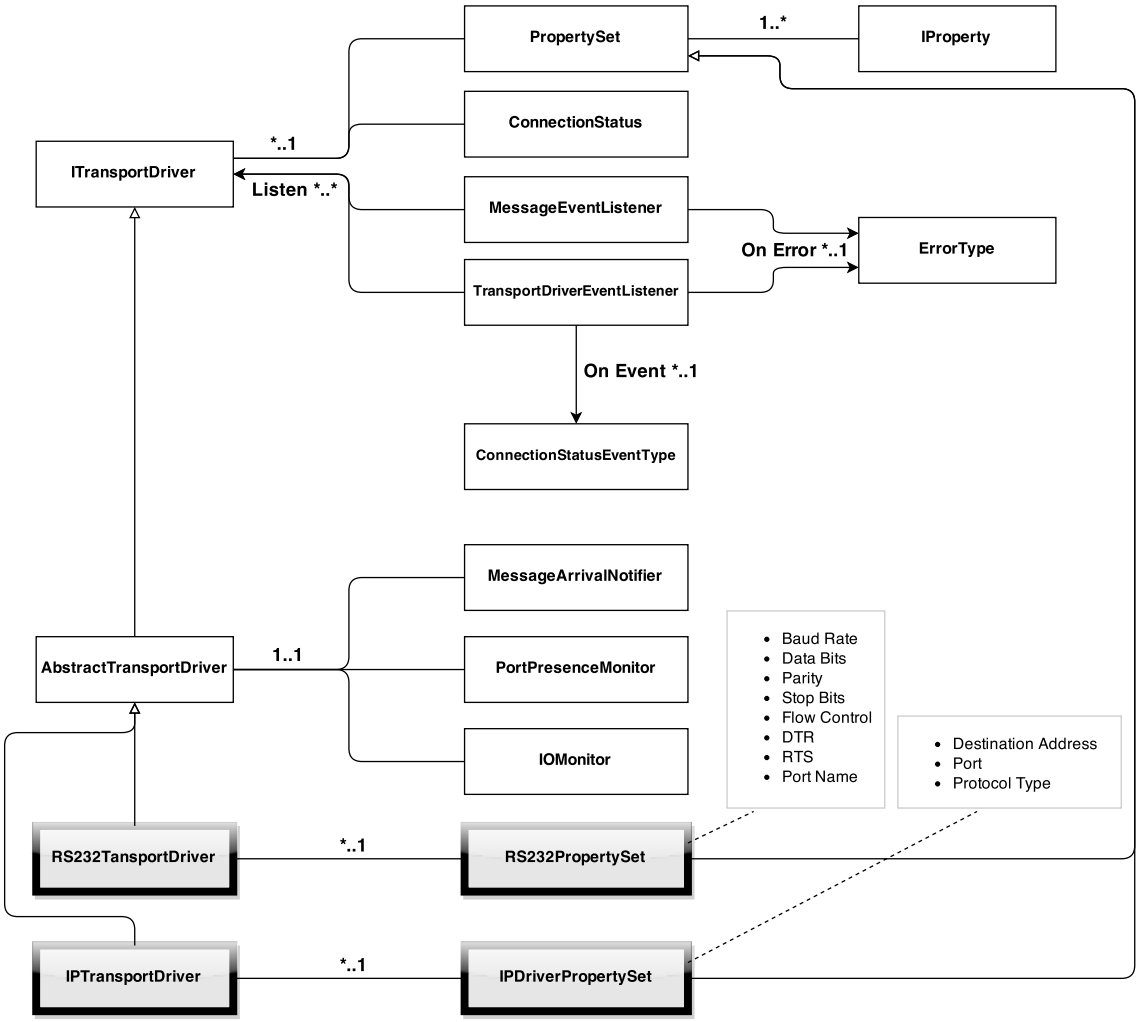

Figure 6.5: Transport Driver class diagram. Black arrows depict dependencies from one component to another. Abstractions and interfaces are in white, in bold we have example implementations of RS232 and IP transport drivers, related to the list of each driver's properties.

Transport Drivers are then used by Gateway objects, which register listeners to receive notifications on new messages or connection problems, open and close the driver when needed, and write the desired messages to the network. Listeners are very important because they provide upper layers with information that may trigger certain control mechanisms. For example if the connection is lost, the driver might need to be restarted. On the other hand listeners can be used by loggers to log the system's activity.

Capability	DOG	openHAB	Our Solution	Description
Listener Notification	○	○	●	Maintains and notifies a list of listeners of incoming network messages and events such as connection errors.
Communication Link Monitor	○	○	●	Regularly monitors the connection status.
Sending Message Scheduler	○	○	●	Manages the queue of messages to send to the network, avoiding problems such as concurrent writing.
Configuration Addressing	○	●	●	Automatically informs the driver about the configurations selected by the user.
Resource Managing	○	○	●	Manages the resources used by each driver, detaining those resources when the driver opens and releasing them at closing.
Status Logging	○	○	●	Automatically logs the driver activity.
Timeout Handling	○	○	●	Detects and notifies about situations where any given operation exceeded an acceptable amount of time.

Table 6.1: Capabilities natively present in any Transport Driver, not requiring developer implementation. We use a ● to indicate if each feature is also natively supported by the solutions analyzed in Chapter 5 or a ○ otherwise.

Name	**Input Parameters**	**Output Parameters**	**Description**
open	None.	*True* if connection was opened, *False* if an error occurred.	Opens a connection.
close	None.	*True* if connection was closed, *False* if an error occurred.	Closes the connection.
getCurrentConnectionStatus	None.	A connection status object.	Gets the current connection status.
getCurrentPortName	None.	The port name.	Gets the currently connected port name.
write	The message to send in bytes.	None.	Writes a message to the network.
getPropertySet	None.	A property set containing the properties.	Gets the properties used to configure this driver.
isDriverOpened	None.	*True* if connection is opened, *False* otherwise.	Checks if the driver is opened.
addMessageListener	The message listener.	None.	Register a message event listener to receive notifications about the arrival of new messages.
addTransportDriverEventListener	The event listener.	None.	Register an event listener to receive notifications about the connection status.
removeMessageListener	The message listener.	None.	Removes the message listener.
removeTransportDriverEventListener	The event listener.	None.	Removes the event listener.

Table 6.2: Summary of the methods defined in the interface class ITransportDriver. The only methods that developers need to implement for each new Transport Driver are presented in bold.

6.2.2 Gateways

Conceptually, in our model, a Gateway is essentially a datapoint provider, which stores a list of known datapoints, tracks their connection status, notifies registered listeners about any occurring event such as connection error and sends and receives messages using any given Transport Driver (Figure 6.6). Any Gateway can have its Transport Driver swapped at runtime, meaning that no Gateway Driver is attached to a specific Transport Driver, increasing flexibility when dealing with gateways who support different communication links such as RS232, USB or TCP/IP. However some Gateway Drivers are designed for gateways having only one type of communication link. For example a Gateway Driver for Lifx[2] would only work with an IP Transport Driver, since Lifx only relies on TCP/IP. The concept of a Gateway Driver is defined in the interface class *IGateway*, whose methods are highlighted in Table 6.4.

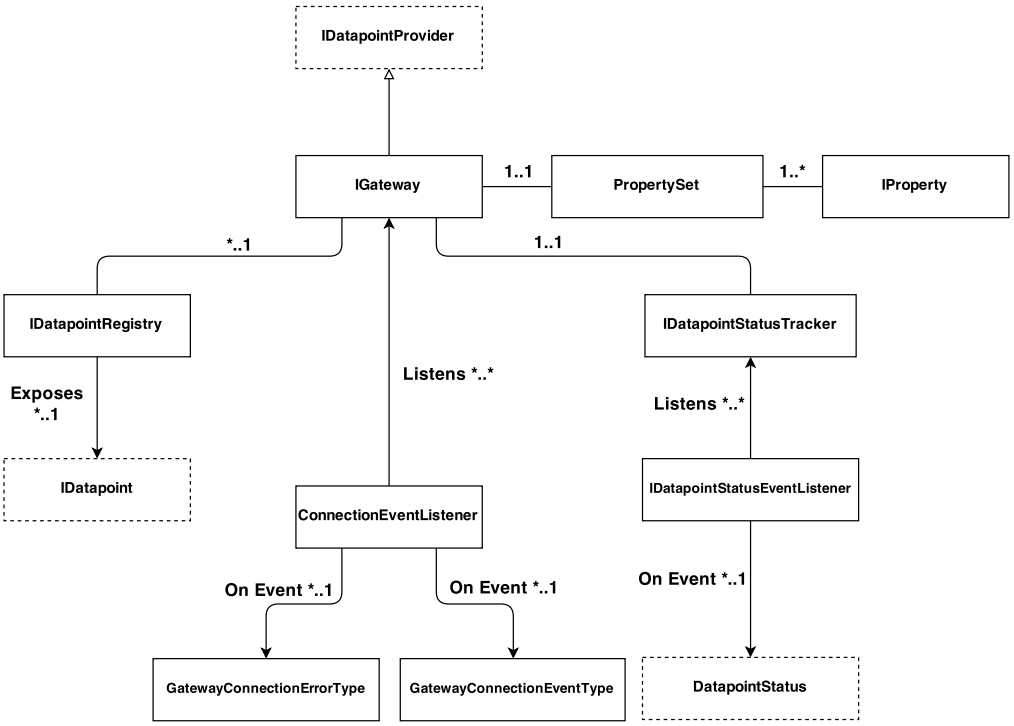

Figure 6.6: Gateway Driver class diagram. Black arrows depict dependencies from one component to another. Dashed boxes represent the usage of classes defined in another diagram.

The datapoint list for each gateway is managed by one or more Datapoint Registries, which can be extended with alternative implementations. Depending on the registry implementation, it may fill itself with datapoints stored in a text file on disk, or can ask the gateway to use a *datapoint discovery service*, which makes sense for protocols supporting these discovery services such as KNX.

A Datapoint Status Tracker is responsible for monitoring and updating the communication status of each datapoint in the Gateway's registries. Similarly to the Datapoint Registry, it can be extended with several implementation, which go deriving each datapoint status studying the network activity or by polling each datapoint regularly to check the connection.

As Transport Drivers, Gateways also provide notification capabilities, enabling listeners to detect

[2]Lifx Project Website: `http://lifx.co/`

Capability	DOG	openHAB	Our Solution	Description
Listener Notification	○	○	●	Maintains and notifies a list of listeners of incoming network messages and events such as connection errors.
Datapoint Status Monitor	○	○	●	Regularly monitors the connection status of each datapoint.
Datapoint Listing	○	○	●	Lists and manages all datapoints exposed by each gateway.
Request Scheduler	○	○	●	Manages the queue of requests made to the gateway, avoiding problems such as concurrent writing.
Configuration Addressing	○	●	●	Automatically informs the driver about the configurations selected by the user.
Resource Managing	○	○	●	Manages the resources used by each driver, detaining those resources when the driver opens and releasing them at closing.
Status Logging	○	○	●	Automatically logs the driver activity.
Timeout Handling	○	○	●	Detects and notifies about situations where any given operation exceeded an acceptable amount of time.
Transport Driver Integration	○	○	●	Automatically integrates the Gateway Driver with any desired Transport Driver.

Table 6.3: Capabilities natively present in any Gateway Driver, not requiring developer implementation. We use a ● to indicate if each feature is also natively supported by the solutions analyzed in Chapter 5 or a ○ otherwise.

connection problems or network events such as read and write request messages sent to the network. Developers wishing to implement their own gateway drivers have clear implementation guidelines. Every gateway driver must implement four classes as shown in Figure 6.7. So any other gateway driver would have essentially the same structure, making it easier to re-use and understand different drivers from different developers than in the previously studied solutions. Moreover, by changing the Transport Driver of the EG2[3] implementation from an TCP/IP Transport Driver to a RS232 Transport Driver, we can control a different gateway called SI2[4], since both use the same communication protocol.

The Abstract Gateway implementation uses a Request Scheduler, which can also be extended, to sort the received read or write requests and dispatch them by a given order. One basic scheduler would be a basic first-in-first-out, but we might also implement a different scheduler that would, for example, give higher priority to write requests over read requests.

By following these implementation guidelines, every Gateway Driver developed will be compatible with any extension of Datapoint Registry, Datapoint Status Tracker and Request Scheduler. Unless developers deliberately make these extensions depend on any given technology specificities. For example, Datapoint Registries who rely on a datapoint discovery service, will render ineffective if used with Gateways not supporting this feature. Table 6.3 summarizes the native capabilities of any Gateway Driver.

[3]EG2 iLight Gateway Datasheet: `http://www.ilight.co.uk/ican-datasheets/iLight--EG2-and-EG2NA--datasheet--rev3--1014--EATON.pdf`

[4]SI2 iLight Gateway Datasheet: `http://www.ilight.co.uk/manuals/SI-2%20installation%20guide%2073-887-00%20IM8406%20Iss.01%28c%29.pdf`

Name	Input Parameters	Output Parameters	Description
open	None.	*True* if connection was opened, *False* if an error occurred.	Opens a connection.
close	None.	*True* if connection was closed, *False* if an error occurred.	Closes the connection.
getDatapoints	None.	A list with all the exposed datapoints.	Lists all the datapoints exposed by this gateway.
getCurrentPortName	None.	The port name.	Gets the currently connected port name.
getPropertySet	None.	A property set containing the properties.	Gets the properties used to configure this driver.
isOpened	None.	*True* if connection is opened, *False* otherwise.	Checks if the driver is opened.
getDatapointStatus	The datapoint to check.	The status of the datapoint (online, offline, unknown).	Gets the datapoint status.
submit	The request and a callback object.	None.	Schedules a new read or write request in this gateway, notifying the result (success or failure) to the callback object. This method uses the implementation of the class *AbstractRequestHandler* to fulfill the request.
addConnectionEventListener	The connection listener.	None.	Register a connection event listener to receive event notifications.
addDatapointStatusListener	The status listener.	None.	Register an event listener to receive notifications about the connection status.
removeConnectionEventListener	The connection listener.	None.	Removes the connection listener.
removeDatapointStatusListener	The status listener.	None.	Removes the status listener.

Table 6.4: Summary of the methods defined in the interface class IGateway. The only methods that developers need to implement for each new Gateway Driver are presented in bold.

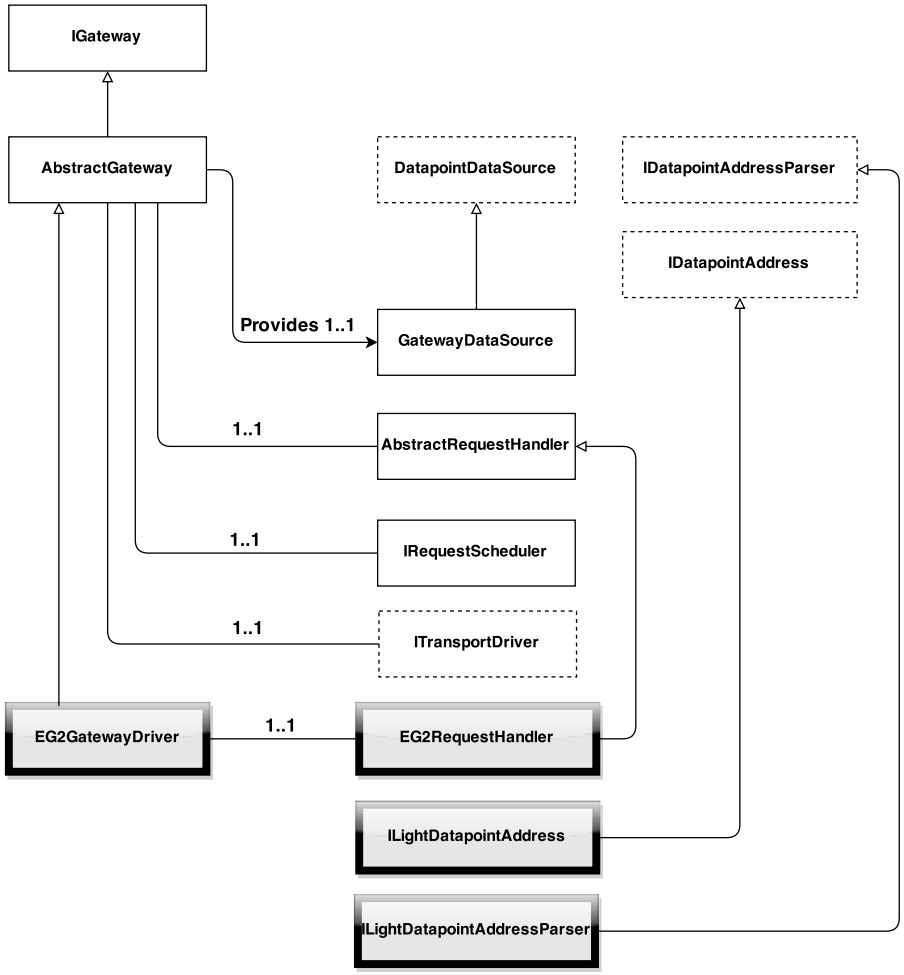

Figure 6.7: Class diagram for the Abstract Gateway Driver. Black arrows depict dependencies from one component to another. Abstractions and interfaces are in white, in bold we have an example implementations of an iLight gateway called EG2. The dashed boxes represent the usage of classes defined in another diagram.

6.2.3 Datapoints

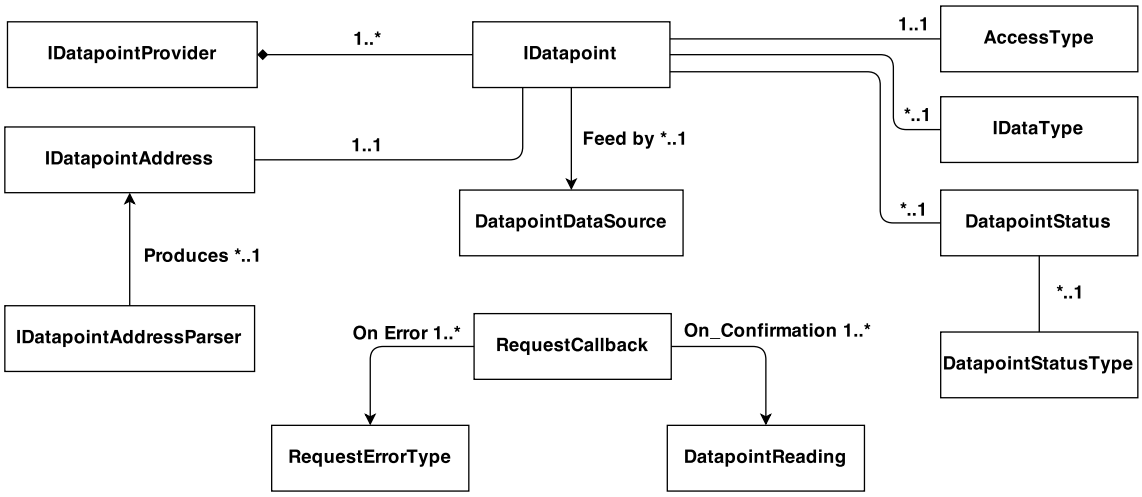

Figure 6.8: Datapoint class diagram. Black arrows depict dependencies from one component to another.

Datapoint objects define the core concept of our framework. They are the interface exposed to the Application Layer, through Device Drivers or Gateways. In their essence, datapoints provide the means to read and write values from network devices. This concept holds as the main idea behind our proposed solution's architecture. Every automation technology has one common concept: the datapoint. It is transversal to every automation technology the existence of addressable data-sources which can be read or written. So in order to create a common abstraction that fits with any technology we adhere to this concept and translate every technology's means of interaction to datapoint interactions. The full class diagram of a datapoint object is depicted in Figure 6.8.

Every datapoint is associated with a datapoint address object, a datapoint provider, a data-source, an access-type, a status and a data-type. Datapoint addresses contain the unique address of a datapoint, and can be generated by a datapoint address parser object, for example, in the situation of parsing the address from a configuration file. The datapoint provider object is responsible for mediating the read and write operations through a *Datapoint Data Source* object. When we read or write a value to a datapoint object, the provider of that datapoint, which can be a Gateway Driver, will receive the request and dispatch it. The access type object holds the information concerning the type of interactions possible with a given datapoint. The status object reflects the system knowledge about a datapoint—online, offline or unknown.

Finally the datapoint's data-type object is used to inform any application trying to use the datapoint about the compatible ways of representing information. For example boolean values should be represented as "*true*" or "*false*", in contrast a temperature value could be similar to "25.1°C". Our data-types library is defined upon the well known JScience library[5], which defines 46 SI units, 84 non-SI units and means for converting between compatible dimensions.

The information model translation from automation technologies is usually straightforward. For exam-

[5]JScience Library Website: `http://jscience.org/`

ple, KNX devices, which have addresses, are directly mappable into the datapoints concept. The same goes for other protocols such as iLight, X10 and LonTalk. A different type of translations are the ones who do not relate directly to a network address, but to a system procedure. For example, in iLight we have specific protocol messages which enable us to activate any given scenario in a certain area. We can translate this to a virtual datapoint object, where writing to that datapoint equates to the activation of that particular scenario in that area. Table 6.5 summarizes the operations supported by a datapoint object.

Name	**Input Parameters**	**Output Parameters**	**Description**
getCurrentDatapointValue	A callback object	None.	Requests the latest reading, notifying the result (success or failure) to the callback object.
getCachedReadings	The amount of readinds to collect and a callback object.	None.	Requests a batch of readings, notifying the result (success or failure) to the callback object.
getDatapointDataType	None.	The data-type of the datapoint.	Gets the data-type for this datapoint, informing us how to interpret the values read and how written values should be structured.
getAccessType	None.	The access type (read, write or read and write).	Gets the access type for this datapoint.
getDatapointAddress	None.	An object holding the address of this datapoint.	Gets the unique address of this datapoint.
getDatapointConnectionStatus	None.	An object containing the datapoint status (online, offline, unknown).	A datapoint status indicates if this datapoint is online and can be reached or, in contrast, it can be offline. In some situations datapoint status checking is not supported by the gateway so the datapoint status may remain unknown until a read or write operation is able to confirm its status.
getDescription	None.	The datapoint's description.	The human-readable description of this datapoint.
getReadCacheSize	None.	The size of this datapoint cache.	Informs how many readings can this datapoint cache store.
getHysteresis	None.	The hysteresis.	The minimum amount of time in milliseconds that the client must wait until the datapoint can be written again.
getParentProvider	None.	The Datapoint Provider object that exposes this datapoint.	Gets the parent provider.
isVirtual	None.	*True* if this datapoint is virtual, *False* otherwise.	Checks if this datapoint is virtual.
setCurrentDatapointValue	An object holding a new value according to this datapoint's data-type and a callback object.	None.	Sets the current datapoint value. This method also validates the given value against this datapoint's data-type.

Table 6.5: Summary of the methods defined in the interface class IDatapoint.

In contrast with Transport and Gateway Drivers, developers do not need to extend nor implement any

aspect of the datapoint definition, since Gateway Drivers already undertake the address mapping from the underlying vendor-specific technologies to datapoint objects. Having a single datapoint representation enables the binding of datapoints from different technologies.

6.2.4 Device Drivers

Device objects arise from the need for simplifying some device control aspects, abstracting the Application Layer from low-level and technology specific details.

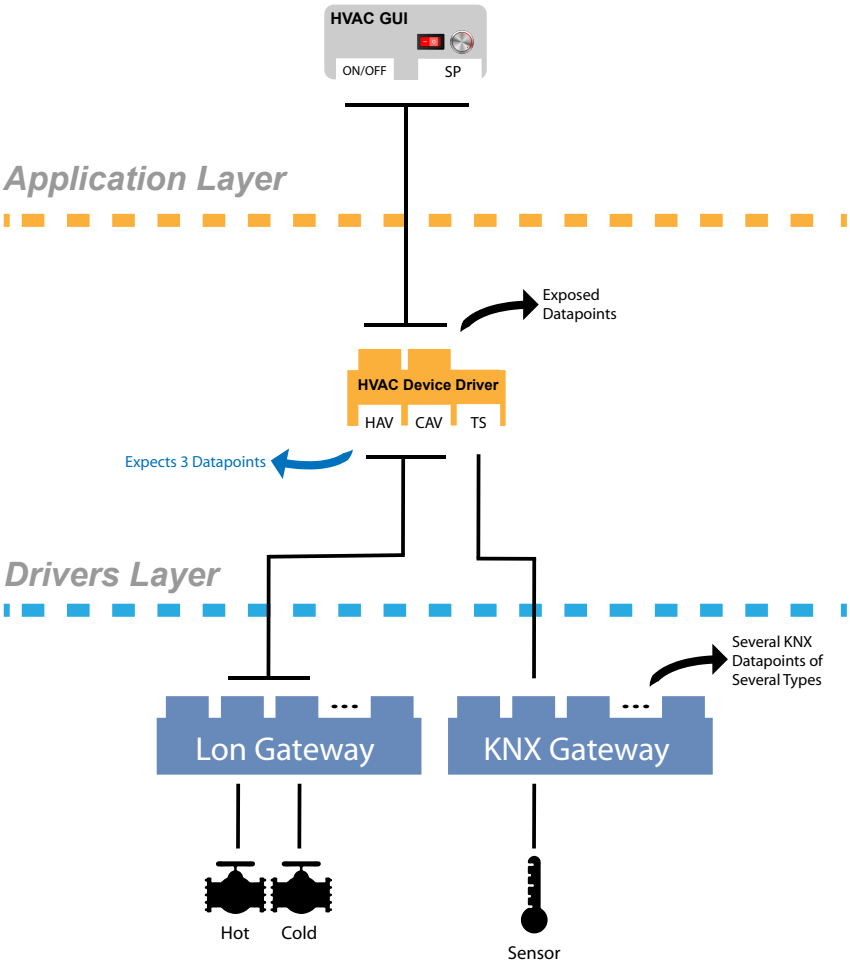

Figure 6.9: Two air valves, one hot air valve (HAV) and one cold air valve (CAV), which control a room's temperature are connected to a Lon network as parte of the HVAC system. Meanwhile a KNX temperature sensor measures the temperature in that room. Using a Device Driver we can abstract the task of controlling of the air valves in order to reach a desired temperature setpoint (SP). The Device Driver acknowledges a desired SP, reads from the KNX temperature sensor and actuates on the Lon air valves accordingly.

Device objects are a group of datapoints which, controlled in a specific way, can abstract upper layers from control specificities of certain devices. In the end Device objects expose datapoints composing a simplified interface to manage any particular device. For example, consider a complex HVAC system where there is no datapoint to define a given temperature setpoint. Its temperature can be defined by

actuating on the hot and cold air valves, which are exposed as datapoints. Because controlling this HVAC system in order to obtain the desired setpoint temperature is difficult, a Device object is used to group the necessary actuation datapoints, and to expose a simpler interface such as an ON/OFF and a temperature setpoint datapoints. The control loop responsible for translating a temperature setpoint to actuations in the air valves would be contained within the Device object implementation. Figure 6.9 illustrates the advantages presented by Device objects for the abstraction and integration of different network datapoints.

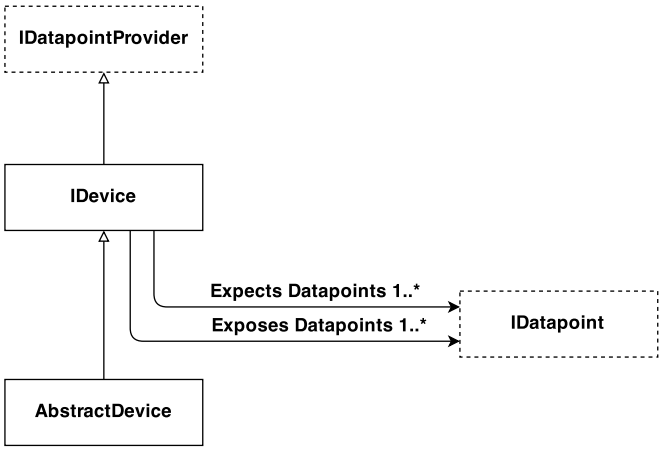

Figure 6.10: Device class diagram. Black arrows depict dependencies from one component to another. Dashed boxes represent the usage of classes defined in another diagram.

Figure 6.10 depicts the class diagram for a Device and Table 6.6 lists its methods. Essentially a Device object contains information concerning the datapoints data-types it expects, alongside with their read and write capabilities. Similarly, it also describes the characteristics of the datapoints they expose.

Name	Input Parameters	Output Parameters	Description
getDatapoints	None.	A list with all the exposed datapoints.	Lists all the datapoints exposed by this device.
isEnabled	None.	*True* if the device driver is running, *False* otherwise.	Checks if the driver is operating, having all its expected datapoints fulfilled.
bindExpectedDatapoint	The position of the datapoint to bind and the datapoint object	None.	Binds a new datapoint to this device driver.
unbindExpectedDatapoint	The position of the datapoint to bind and the datapoint object	None.	Unbinds a new datapoint to this device driver.
getExposedInterfaceDefinition	None.	A list with the datapoint definitions.	Lists the data-types and capabilities of the datapoints exposed by this device driver.
getExpectedInterfaceDefinition	None.	A list with the datapoint definitions.	Lists the data-types and capabilities of the datapoints expected by this device driver.

Table 6.6: Summary of the methods defined in the interface class IDevice. In bold we have the only methods that developers need to implement for each new Device Driver.

6.3 Implementation Challenges

6.3.1 Finding a Common Information Model

During the conception and implementation of our solution we have faced several issues while integrating different technologies. One of the major issues was finding a common information model ontology that would fit any technology. According to several authors, to address semantic interoperability we should fulfill the need for a single shared ontology, bridging different expressions of the same knowledge in different representation languages [92, 93, 94]. From Tables 3.1, 4.1 and B.1 we have concluded that different technologies represent their domain knowledge differently, usually through features which are not considered in other technologies—such as automatic datapoint discovery, control loops definition, action sequences activation, among others.

It is impossible to create a static information model capable of gathering all the existing functionality and additional functionalities that advent technologies may bring. Moreover, we also cannot provide a unique API for interacting with all network devices the same way. For example, a network protocol specification providing control loop features must have a different API in contrast with the ones which do not offer those features. Additionally, such features may be solely sustained by the devices themselves, meaning that each vendor's devices may have different means for programming their behavior (such as defining control loops) hence provide the system with a new set of features, i.e. a new API—which is the most common scenario found in the Building Automation context. Thus, even for each protocol we cannot guarantee that a single API might fulfill every need. In the worst case scenario, we end up having one API for each unique device type, which in no way provides interoperability nor simplifies external interaction with the system from third-party applications.

In order to keep both our information model and API simple and functional, we boiled down the number of features to just three. These three features are available in every network technology, no matter its usage context. These features are listing, reading and writing to network addresses. This way a unified information model can be created and used among different automation technologies, since every operation can be expressed in terms of reading and writing to network addresses—known as datapoints in Building Automation. This enables easy exchange of information between datapoints from different technologies and vendors, enabling previously impossible undertakings such as *technology-agnostic datapoint binding*. As a result, we can provide a single and static API, which directly maps our datapoint-centric ontology.

With this conception we are able to read and write to any device within any given network address and modulate tasks such as activating a scenario, a control loop or even an alarm as a write operation to a certain address. Gateways may also provide virtual network addresses to enable changing its configurations. However, in some situations translating a feature into the simple language of reading and writing datapoints may not be straightforward.

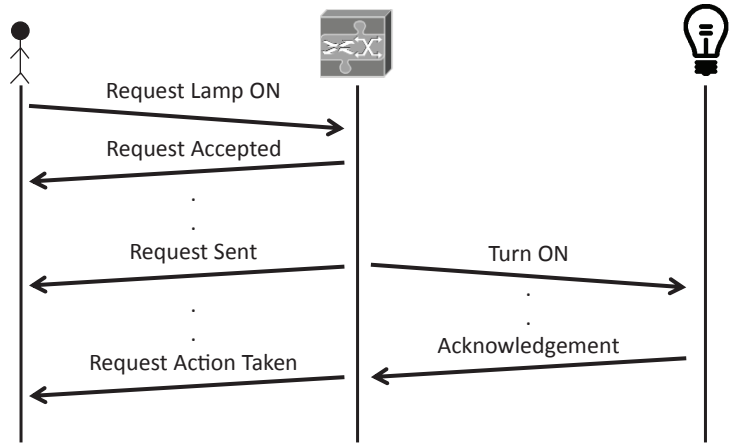

Figure 6.11: Sequence diagram illustrating how a client is asynchronously notified of its request's progress. On the left we have the client application performing the request to our framework, in the middle, which sends the proper turn ON command to the lamp device on the right. The client application is never blocked, being free to perform other tasks in the time periods depicted by the dots in vertical.

6.3.2 Behavior Gaps

There are several tasks in Building Automation which falsely seem very simple, but a closer look quickly shows how complex they are. Consider a simple lamp connected to the fieldbus network. The user sends an ON command to that lamp, which usually takes some seconds to result in any physical actuation, if everything goes well. However the following questions start arising: (i) How long will the user have to wait to receive an acknowledge about his command? (ii) Should the user be blocked during the time between the command request and the acknowledgement or should he be able to carry on with other tasks, getting notified by the system that the command was successful later in time? (iii) What if the lamp cannot acknowledge that it was turned ON? (iv) What happens if two or more users try to concurrently address the same lamp with different commands, knowing that the lamp may have an hysteresis time associated? (v) How long, after a command is sent, can one validate the lamp's final status? (vi) What happens in an error situation such as a connection lost or a device malfunction, how should the user get notified of such problems? There are no standards nor bibliographic references posing as guidelines for solving these issues, thus we are facing several behavior gaps.

Given all these questions around a simple lamp device, it is easy to predict the complexity growth for more elaborated devices. However, some of these problems arise due to the tendency of synchronizing every interaction with the system. Some authors even state that synchronized API calls are harmful to client applications [95].

Our solution to this problem was to create a request system where client applications perform read and write requests to our API, those requests are submitted and applications are later notified of their fulfillment. So every API call which involves interaction with any physical device will no longer block the client application. No requests are left unanswered, in an error situation the client is notified of the error, even if that means that the device did not respond within an acceptable time—meaning that the request timed out. Figure 6.11 depicts how a writing request to turn a lamp ON is handled.

Thanks to this asynchronicity, we are able to schedule each request, through the use of the Request Scheduler objects depicted in Figure 6.7, since we no longer need to respond to it immediately because the client is not blocked waiting for the answer. This helps us to serialize and prioritize concurrent requests. For example if we receive two simultaneous requests, one for reading from one datapoint and another to write to that same datapoint, maybe we should give priority to the write request. Otherwise the read request will be outdated as soon as the following write request is handled, rendering the previous read request useless since it no longer represents that datapoint's current state.

Other problem that we are able to solve resorting to request schedulers is network flooding. If several clients send equivalent requests—like several repeated commands turning a lamp ON—, the scheduler can detect that situation and handle just one request, ignoring the repeated ones. This reduces the amount of messages traveling in the fieldbus network, improving communication speeds and avoiding buffer overflow situations in devices with low memory which results in message loss.

6.3.3 Reducing Driver Implementation Time

A different, but important issue at hand was the simplification of the driver implementation task. From the analyzed solutions in previous sections we learned that driver manufacturing is the most repetitive task in the context of integration frameworks. For example, openHAB currently has 95 driver implementations. In order to make our framework easy to work with, we tried to automate most of the driver's tasks (see Tables 6.1 and 6.3). Every driver shares a considerable list of responsibilities, which are independent of the underlying technology, such as connection status listener, event notification and logging, incoming and outgoing messages management, error handling, among others. So for every extensible feature of our platform, we provide abstract classes which already provide an implementation of these features, freeing the developer from repetitive and prone to error tasks. In the end developers solely need to implement the communication protocol inherent to the technology they are integrating. Furthermore, we have fully separated the transport layer from the gateway protocol layer, as shown in previous sections. This increases code re-usability and ensures a minimum level of functionality for all drivers, while simultaneously decreasing the amount of time spent and errors produced during the implementation of any driver.

As a consequence of re-usability and the small margin for extrapolation during the implementation process, drivers will share the same project structure, meaning that knowing how to implement one driver, equates to knowing how to interpret other developers' drivers. This improves team communication, which is one of the key aspects of collaborative software development.

6.3.4 Interfacing with External Remote Applications

Another challenge that we had to face was enabling external applications, usually running in remote servers, to control the underlying automation infrastructure. For this we have developed a REST API, which specification is described in Table 6.7. Our goal was to create a simple API which enabled the services of listing, reading and writing to datapoints, directly mapping with our unified information model.

REST Endpoint	Input Parameters	Output Parameters	Description
/datapoints	None.	A list with all the exposed datapoint addresses.	Gets the addresses of all datapoints of devices controlled by this service.
/metadata/{address}	The address of the datapoint.	A list of metadata concerning that datapoint.	Returns the metadata of a given datapoint.
/datapoints/{address}/{numberOfCachedReadings}	The address of the datapoint and the number of cached readings to read (or *0* to get all cached readings)	A list of readings paired with their respective timestamp.	Gets a given number of cached readings of a datapoint counting from the most recent reading.
/datapoints/{address}	The address of the datapoint.	A reading from this datapoint with its associated timestamp.	Gets the latest available reading of a datapoint.
/datapoints/{address}/{value}	The address of the datapoint and desired value to write.	The operation status, indicating the write request was successfully scheduled in the system.	Writes a value to a datapoint.
/requests/{request id}	The id of the request.	The request status, indicating if the request operation was completed or is still pending.	Indicates if a request operation was completed, otherwise, mentioning the reason why the request was aborted or is still pending.

Table 6.7: REST endpoints of the services exposed to external applications. The words between brackets represent the input parameters of each endpoint.

6.3.5 Centralized and Decentralized Approaches

A relevant question that arised with the creation of the hardware abstraction layer was *"Where should the building commissioning configuration lie?"*. Typically systems have their configuration stored in hardware devices, such as timers, schedules and scenarios. Hence the only control given to the end-user is typically the possibility of activating those scenarios or the manual interaction with some devices through physical interface panels.

With the presence of our solution we are exposed to a paradigm shift, where commissioning information does not need to be stored in the devices themselves, but in the single server supervising the entire system. This leads to the centralization of functionality, having all its advantages and disadvantages. Consider the situation where the end-user can program in our solution the desired scenarios, timers and schedules. In this situation the user is able to modify the system at any time, without the need to recommission network devices each time a adjustment is desired, simply because it is all handled in software. However, this would mean that our framework has full responsibility over maintaining these configurations working, meaning that if the server running our application runs into a problem, the system may come to an halt and be left in an unknown state. In other words, if our framework gets shutdown, all the configured scenarios, timers and schedules will have no effect. This is the main issue concerning centralized systems [96].

As an attempt to avoid falling in this pitfall our solution supports both centralized and decentralized approach. It is possible to use a gateway driver, which is already present within the core of our framework,

designed to enable the integration of several different instances of our framework running in different servers. This way it is possible to have redundancy, where in a fault situation only part of the overall systems gets affected, since different instances are dealing with different responsibilities. Additionally, redundancy can be enhanced to the point of having one instance taking the place of a faulty instance, further reducing the chances of the system suffering from a fault.

Chapter 7

Validation

This work has resulted in a framework which promises to consolidate software development for Building Automation, while providing the required basis for creating new software solutions, fully abstracted from hardware specificities. This evaluation tries to validate the developed framework according to those principles.

The evaluation process is split in two parts, the first part uses well established software engineer methodologies and metrics to assert the architectural quality of our solution in comparison to the solutions analyzed in Chapter 5. The second part consists of two case studies, one in the scope of the European Project SmartCampus, which provided the means to correctly test our solution through the access to the IST Taguspark's automation network, and several device drivers were implemented in order to fulfill the project requirements. The second case study consists of an example where our framework controls a mobile robot, showing that our solution is flexible enough to be capable of performing in contexts different from BA, in particular, robotics.

7.1 Quantitative Architecture Validation

We are going to validate the existing framework architectures by analyzing quantitative and qualitative aspects. To obtain quantitative measurements concerning the architecture of the analyzed solutions, this validation stage comprises the application of object-oriented metrics. The main aspects that can be measured in a software are functionality, usability, maintainability and portability [97]. According to several literature references, in order to develop a system easier to maintain and to extend, we should be concerned about maximizing the system's *cohesion* and minimizing the system's *coupling* [98, 99, 100].

Coupling measures the degree to which each program module relies on each one of the other modules. The greater the coupling is, the more difficult is to modify or exchange some module without affecting other modules. The coupling factor is used to measure a system's stiffness—a system having highly coupled modules is harder to maintain than a system having low coupled modules [101]. In the context of object-oriented languages we will consider coupling at the class level. *Afferent coupling* and *efferent coupling* are two metrics used to measure the coupling of an architecture, and consequently, its

instability [102, 103]. Afferent coupling (AC) concerns with the number of modules which are external to, and depend on a given module. Efferent coupling (EC) refers to the number of external dependencies of a given module. On the other hand, the instability per module is calculated based on AC and EC (Equation 7.1), where the goal is to reduce instability, meaning reducing EC and maximizing AC. Usually, in object-oriented languages, EC and AC modules equate to packages.

$$Instability = \frac{EC}{AC + EC} \tag{7.1}$$

In contrast, cohesion concerns how well defined the module's responsibilities are. If a module has high cohesion, it means that this module has a well defined responsibility, otherwise, the module does several tasks that do not have much in common, handling multiple responsibilities, which usually indicates poor system organization. Usually, in object-oriented languages, this metric is applied at the class level, where methods are perceived as responsibilities.

Consider P the set composed by pairs of a given class methods that use totally different resources, where resources can be variables or data structures within that class, and Q the set composed by pairs of methods that use at least one common resource. Given these conditions we know that $P \cap Q = \emptyset$ and that the total number of methods in that class is $|P| + |Q|$. So, the *lack of cohesion in methods* ($LCOM$) of a module can be measured according to Equation 7.2.

$$LCOM = max(|P| - |Q|, 0) \tag{7.2}$$

If $|Q| \geq |P|$ then a module has several related tasks and thus high cohesion, otherwise it has low cohesion. So the intention is to minimize the lack of cohesion metric [101]. To evaluate the lack of cohesion for each architecture we will calculate the average lack of cohesion for every module and the respective standard deviation.

To measure a software architecture's complexity the most frequently used metrics are based on *methods per class* (MPC), *depth of inheritance* (DOI), the *number of children/subclasses* (NOC) and *cyclomatic complexity* (CC) [101, 104, 105]. So for measuring the complexity for each architecture we will calculate the average and the standard deviation for each one of these metrics. In sum, an easily maintainable software should have:

1. **A low number of methods per class**
   The number of methods in a class are correlated with class complexity.

2. **Decreased depth of inheritance**
   The deeper the inheritance tree gets, the greater is the design complexity, since more methods and classes are involved, making it more difficult to predict a class's behavior, thus, increasing developers' cognitive load.

3. **High number of subclasses**
   The number of immediate subclasses is directly correlated with code reuse. Moreover, it highlights

Metrics	Goal
Number of Classes	↓
AC	↑
EC	↓
Instability	↓
LCOM	↓
MPC	↓
DOI	↓
NOC	↑
CC	↓

Table 7.1: Summarized architecture quality metrics according to their desirable tendencies in terms of value minimizing and maximizing, where metrics having ↓ should have their values minimized, and metrics having ↑ should have their values maximized.

#	Title	Description
1	Automation Networks Integration	Interconnection of several automation networks.
2	Message Bridging	Translation and forwarding of messages across different networks.
3	High-Level Network Protocol	Technology independent, high-level network protocol allowing homogeneous access to automation networks.
4	Public Access API	Public API allowing external services to easily interact with each network's devices.

Table 7.2: Summary of functional requirements expected in analyzed solutions.

which classes have more children, which end up being the most critical classes in the overall architecture.

4. **A low cyclomatic complexity**
   Which equates to a low number of conditional branches in the code, resulting in a simpler control flow graph describing all the possible behaviors of the software.

In Table 7.1 we summarize the desirable tendencies in terms of value minimizing and maximizing for each considered metric during this analysis, depicting which metrics should be minimized and which metrics should be maximized [101, 104].

Despite quantitative software analysis being the most accurate way to compare architectures, the blind application of these metrics will also raise potential issues concerning fairness: Since the solutions we are going to analyze were tailored to cover much more functionalities than the ones we are expecting, which are detailed in Table 7.2, it is normal to expect an increased complexity in their overall architecture, resulting in more classes, more methods and possibly deeper inheritance trees. In order to minimize the impact of these extra requirements can have in our analysis, we will carefully isolate the architecture components responsible for the requirements of our interest, to which we will apply the quantitative analysis.

Quantitative analysis, however, will only tell us how easy it is to understand, extend, maintain, reuse and adapt the analyzed software implementation. But it will not extract any useful information concerning the structure of their conceptual domain model, which can be assessed through a qualitative analysis. For the qualitative validation stage we consider the set of requirements we wish to see fulfilled (see Table 7.2), and ascertain how these requirements are honored by domain models of the analyzed

Characteristics	Description	Calculation
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions	Counting the number of implemented functionalities from Table 7.2
Usability	The capability of the software product to be understood, learned, used and attractive to developers, when used under specified conditions	Calculated through complexity metrics of an existing automation network driver implementation, for each analyzed framework
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the environment and in the requirements and functional specifications	Calculated through the analyzed framework architecture's complexity metrics
Portability	The capability of the software product to be transferred from one environment to another. The environment may include organizational, hardware or software environment	Analyze each framework's public API exposing the network services to client applications

Table 7.3: Software characteristics highlighted in ISO 9126-1 Quality Model [97].

frameworks. We conclude the validation of each solution by calculating the set of characteristics supporting the quality of a software as defined by ISO 9126-1 [97], which are summarized in Table 7.3.

In order to validate the functionality of the analyzed solutions, we will use Table 7.2 as a reference and score each functionality as implemented (1), partially implemented (0.5) and not implemented (0). Since these frameworks target users are driver developers, usability will be evaluated from the perspective of how *simple* is it to develop a new network driver for each framework. Complexity metrics will be used to judge the complexity of implemented drivers used to integrate new technologies.

As explained before, the maintainability of a software product is largely evaluated through the complexity metrics, coupling and cohesion. We will also analyze, for the same desired functionalities, the amount of third-party dependencies each solution requires. The aim is to minimize third-party dependencies, since they increase complexity, take up more resources and may require the resolution of transitive dependencies [106].

For the portability aspect, we analyze the existence of a public API which enables other applications to interact with the network devices. During this evaluation we will not consider the underlying technology implementing each API, instead we will focus on the services it provides, such as reading, writing and listing devices in a network, where each service is worth one score value, totalizing three points if all these services are provided.

In addition to these validation aspects, Chapter 7.2 presents a case study where these analyzed solutions will be tested in a production environment, having to integrate a heterogeneous network of devices, determining the complexity of the task of setting up each framework in the production environment, comparing the obtained results with our proposed solution.

In the previous section we have detailed our domain model and the resulting architecture. We believe that we have achieved a good commitment between model simplicity and provided functionality. Moreover, we avoided blending domain-specific semantics into our model, maintaining an elevated level of compatibility in several situations. To confirm this hypothesis we have tested our solution in two real world scenarios, showing that this framework is not essentially restrained to the Building Automation

field.

7.1.1 Domotic OSGi Gateway

As described in the official documentation, DOG covers all functional requirements expected, as shown in Table 7.4. However, the public device API exposed by DOG, summarized in Chapter 5, offers more services than just the network services in Layer 2, which may present themselves as a cognitive burden if our only desire is to control devices directly without embedding them in DOG's home model concepts, such as location related semantics.

#	Title	Coverage
1	Automation Networks Integration	●
2	Message Bridging	●
3	High-Level Network Protocol	●
4	Access API	●

Table 7.4: Summary of functional requirements covered by DOG. Legend: ◯ *not* covered ◐ *partially* covered ● *fully* covered.

For the architecture quality analysis of DOG network solution we will consider layer 2 components and its direct dependencies only, as depicted in Figure 5.2. These components are ***it.polito.elite.dog.core.clock***, ***it.polito.elite.dog.core.devicemanager***, ***it.polito.elite.dog.core.devicefactory***, ***it.polito.elite.dog.core.library*** and ***it.polito.elite.dog.core.housemodel.simple***. The calculated metrics are summarized in table 7.5

Metrics	Mean	Standard Deviation
Number of Classes	814	-
AC	22.475	68.653
EC	13.575	41.922
Instability	0.502	0.394
LCOM	0.095	0.237
MPC	6.291	7.064
DOI	1.690	1.393
NOC	1.186	7.506
CC	1.369	1.232

Table 7.5: Summarized metrics from DOG's architecture quality analysis.

To access the complexity of implementing a driver for DOG framework we evaluated the stock KNX driver presented in DOG's project website, which we compare to our implementation of the KNX driver. This driver consists of a project containing 8 packages, 15 classes and 2982 lines of code. The architecture metrics of DOG's KNX driver are summarized in Table 7.6. On the other hand, Table 7.7 summarizes the results obtained for DOG, which are compared to our framework in Section 7.1.3 in order to extrapolate the architectural performance of both solutions.

Metrics	Mean	Standard Deviation
Number of Classes	15	-
AC	4.000	4.123
EC	1.625	0.696
Instability	0.522	0.392
LCOM	0.447	0.297
MPC	12.933	7.594
DOI	1.333	0.596
NOC	0.333	1.011
CC	2.168	2.349

Table 7.6: Summarized metrics from DOG KNX driver's architecture quality analysis.

Characteristics	Score
Functionality	4 of 4
Usability	See Table 7.6
Maintainability	See Table 7.5
Portability	3 of 3

Table 7.7: DOG architecture analysis in the light of ISO 9126-1 highlighted characteristics.

7.1.2 OpenHAB

Concerning the architecture quality analysis of openHAB network solution we considered the core components and their direct dependencies only. These components are ***org.openhab.config.core***, ***org.openhab.core***, ***org.openhab.core.library***, ***org.openhab.core.persistence***, ***org.openhab.core.scheduler***, ***org.openhab.core.scriptengine***, ***org.openhab.core.transform***, ***org.openhab.io.net***, ***org.openhab.model.core*** and ***org.openhab.model.item***. The calculated metrics are summarized in Table 7.8. On the other hand, has we can conclude from our analysis in Section 5.2, openHAB covers all expected functional requirements as shown in Table 7.9.

Metrics	Mean	Standard Deviation
Number of Classes	223	-
AC	32.078	211.546
EC	2.1228	6.324
Instability	0.481	1.086
LCOM	0.095	0.799
MPC	3.510	13.623
DOI	1.316	2.883
NOC	0.229	2.457
CC	2.283	8.750

Table 7.8: Summarized metrics from openHAB's architecture quality analysis.

To access the complexity of implementing a driver for the openHAB framework we evaluated the stock KNX driver presented in the project's website. This driver consists of a project containing 7 packages, 15 classes and 1721 lines of code. The architecture metrics of this driver are summarized in Table 7.10 and its class diagram is partially depicted in Figure 5.7. On the other hand, Table 7.11 summarizes the results obtained for DOG, which are compared to our framework in Section 7.1.3 in order to extrapolate

#	Title	Coverage
1	Automation Networks Integration	●
2	Message Bridging	●
3	High-Level Network Protocol	●
4	Access API	●

Table 7.9: Summary of functional requirements covered by openHAB. Legend: ◯ *not* covered ◐ *partially* covered ● *fully* covered.

the architectural performance of both solutions.

Metrics	Mean	Standard Deviation
Number of Classes	15	-
AC	1.429	1.178
EC	1.429	0.728
Instability	0.569	0.285
LCOM	0.101	0.215
MPC	5.000	5.342
DOI	1.467	1.258
NOC	0.267	0.573
CC	4.022	7.159

Table 7.10: Summarized metrics from openHAB KNX driver's architecture quality analysis.

Characteristics	Score
Functionality	4 of 4
Usability	See Table 7.10
Maintainability	See Table 7.8
Portability	3 of 3

Table 7.11: OpenHAB architecture analysis in the light of ISO 9126-1 highlighted characteristics.

7.1.3 Result Comparison

In contrast to DOG and openHAB, our solution provides the most minimalist API, exposing no more services than listing, reading and writing datapoints, as previously explained in Chapter 5, in order to avoid creating unnecessary complexity fruit of exposing redundant services. In Tables 7.12 and 7.13 we present the architecture's quantitative analysis, comparing the results obtained with the previous analyzed solutions; DOG and openHAB. We consider having a *superior performance* in a certain architectural aspect if our mean is closer to the desired tendency of that aspect (for example, we want AC to increase and EC to decrease) and if our obtained standard deviation for that aspect is more contained, i.e., having a lower *coefficient of variation*. Since high standard deviations dilute the usefulness of the mean.

One remarkable result that we were able to achieve was the low number of classes needed to fulfill the requirements of Table 7.2 compared to DOG and openHAB modules which are responsible for obeying these requirements. However, having less classes just by itself has no meaning unless more

Metrics	Mean	Standard Deviation	DOG	openHAB
Number of Classes	91	-	●	●
AC	6.571	9.439	●	●
EC	6.286	3.881	●	●
Instability	0.643	0.283	●	●
LCOM	0.178	0.307	○	●
MPC	6.481	8.257	○	●
DOI	1.231	0.826	●	●
NOC	0.538	0.929	●	●
CC	1.426	1.218	○	●

Table 7.12: Summarized metrics from our solution's architecture quality analysis and comparison with the previous analyzed solutions. We use a ● where our solution is considered to have a *superior performance* at a given aspect and a ○ otherwise, based on the values from Tables 7.5 and 7.8.

Metrics	Mean	Standard Deviation	DOG	openHAB
Number of Classes	6	-	●	●
AC	0	0	●	●
EC	4	0	●	●
Instability	1	0	●	●
LCOM	0.146	0.327	○	○
MPC	4	4.726	●	●
DOI	1.5	0.5	●	●
NOC	0	0	●	●
CC	1.828	2.408	○	●

Table 7.13: Summarized metrics from our KNX driver's architecture quality analysis. We use a ● where our solution is considered to have a *superior performance* at a given aspect and a ○ otherwise, based on the values from Tables 7.6 and 7.10.

information is provided, so another key aspect to note is that the mean number of methods per class (MPC) is essentially the same as the other solutions. The complexity of these methods can be accessed by the LCOM and CC metrics, which is also very close to the analyzed solutions, although not optimal. Overall, this means that our architecture is simpler to understand than of the analyzed solutions given its reduced size and complexity.

On the other hand, we were able to decrease the average amount of inheritance levels (DOI) while maintaining a reasonable number of children (NOC) and instability ratio compared to the other solutions. Moreover, all our metrics are provided with lower standard deviations compared to the other solutions, suggesting that our architecture offers more coherence along its modules. This implies that in sum, any module of our architecture is as easy to maintain as any other module.

As for the KNX driver comparison; our driver is simpler than the ones studied previously, containing fewer classes and fewer methods per class (MPC). The calculated instability for our driver is also remarkably lower and has no deviation, since any driver in our framework has only 4 external dependencies (as depicted in Figure 6.7). However cohesion got negatively affected due to the concentration of contrasting responsibilities in fewer classes—one class is responsible for the tasks of reading, writing and listing datapoints of a gateway.

Concluding, these results express that our solution covers all functional requirements shown in Table 7.14 without compromising the architecture design. From this analysis, which is summarized in

#	Title	Coverage
1	Automation Networks Integration	●
2	Message Bridging	●
3	High-Level Network Protocol	●
4	Access API	●

Table 7.14: Summary of functional requirements covered by our solution. Legend: ○ *not* covered ◐ *partially* covered ● *fully* covered.

Characteristics	Score
Functionality	4 of 4
Usability	See Table 7.13
Maintainability	See Table 7.12
Portability	3 of 3

Table 7.15: Summary of our solution's architecture analysis in the light of ISO 9126-1 highlighted characteristics.

Table 7.15, we can assert that our solution is significantly easier to maintain and to develop on, given the fact that we greatly reduced the number of modules and classes involved in fulfilling all the same requirements, without increasing the code and execution flow complexity.

7.2 Case Studies

7.2.1 SmartCampus Project

The SmartCampus European Project[1] aims at the development of services and applications supported by a data gathering platform that integrates real-time information systems and energy management systems. This integration drives a bi-directional learning process such that both the user and the building learn how to interact with each other in a more energy efficient way.

One of the main challenges of this project was to cope with all the hardware heterogeneity depicted in Appendix C.1 followed by the tight delivery schedules. This meant that we had to homogenize the means by which we communicate with the automation devices and speed up the development process for several applications. These requirements paved the way to validate our solution, since it mainly targets those two problems—hardware heterogeneity and reduced development time.

During this project we had to develop five gateway drivers for interfacing with the KNX, DCOM, iLight, Enoro REST and the Arduino Sensors protocols, and one transport driver for the TCP/IP communication link which was reused in each one of these gateway drivers. The integrated automation network created in the scope of the SmartCampus project consisted of approximately five hundred datapoints.

In order to minimize the IST Taguspark building's energy consumption various technical strategies were adopted. The first one was to bring some control to the users hands through physical panels and graphical user interfaces which enable control of the lights, blinds and HVAC systems in some offices

[1] SmartCampus European Project Website: `http://greensmartcampus.eu`

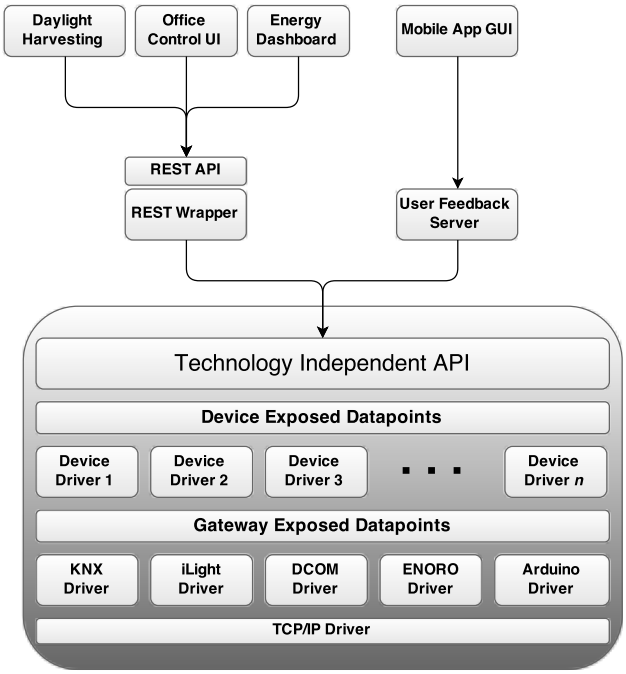

Figure 7.1: Diagram illustrating the application of our solution into the SmartCampus Project. Five gateway drivers were required to integrate five different networks. Although only one TCP/IP transport driver was required for the communication link. All four SmartCampus applications lie atop of the technology independent API, alienated from the underlying technologies. The Configuration Loader application was not depicted in order to simplify this Figure.

(Appendix C.2.b). Then through the creation of smartphone mobile applications we were able to engage students into optimizing energy consumption in classes by letting users to vote their preferences and have the professor to choose the best setpoints according to the votes (Appendix C.2.a). Finally, in order to range a wider audience we have created an energy consumption dashboard to let the building's occupants acknowledge how much is being spent daily, weekly and monthly, and according to several internal and external aspects such as occupation, meteorologic conditions, among others (Appendix C.2.c). Later we developed an application capable of performing daylight harvesting in the library. Using two Arduino light sensors, which were installed in the ceiling, the application was able to measure the light intensity in real-time. According to the variation of the measured sunlight in the library, the system tries to dim the ceiling lights in order to sustain a desired light intensity setpoint but reducing energy consumption. Through daylight harvesting we were able to reduce the library's energy consumption by, approximately, 40%, by simply integrating two sets of devices from different automation technologies and manufacturers (see Figures C.3, C.4 and C.5).

All these applications were developed against the same API, by different developers, abstracted from the building's hardware heterogeneity. Furthermore, it was a project requirement that the Daylight Harvesting, Office Control UI and Energy Dashboard could run in remote servers, so they use the Technology Independent API through our solution's REST Wrapper. Figure 7.1 illustrates where, relative to our solution's architecture, are the developed applications and drivers located.

In order to assess the amount complexity eliminated by our solution we analyzed and compared

the steps required by each framework to fulfill the SmartCampus Project's requirements. This analysis assumes that we are only provided with the core of each solution, stripped from already existing drivers and third-party configuration tools[2]. Tables 7.16 summarizes the steps needed to complete SmartCampus for each of the analyzed solutions, presenting the conclusion that our solution excels in the driver manufacturing and configuration phases (also, keep in mind that it is simpler to implement a driver in our framework as depicted by Tables 6.1 and 6.3). Moreover, it illustrates how the analyzed frameworks assume that the user always needs to use semantics such as device location and graphical interface images and icons, in every domain context. Their models enforce the user to define those semantics even when they make no sense to exist, like in the context of a mobile robot (see Section 7.2.2), which promotes the idea that openHAB and DOG do not target industrial applications—only in the Home Automation context can the user take full advantage of these frameworks.

One aspect observed during this comparison is that both analyzed solutions, DOG and openHAB, push the heterogeneity problem to the configuration phase. During the configuration process the user has to configure each driver differently, sometimes being aware how the driver is implemented internally. In openHAB the user must also manually implement every data-type in the configuration files, expressing the decimal cases and the units. Since each driver or user may implement a data-type differently, two datapoints having the same data-type but with different implementations—like different symbolic representations of the same unit—will not be able to bind. Additionally, in openHAB there is no conversion mechanism between compatible data-types, further hampering interoperability. So in the end these solutions only partially abstract the user and client applications from the underlying technologies, leaving some integration related work to the configuration phase. In contrast our proposed solution has a library of data-types, which are common to every integrated technology. Thus, the only job left to the user whilst the configuration phase is the association of one data-type to its respective datapoint and the attachment of that datapoint to a device driver object.

[2]Since openHAB has a catalog of 95 drivers and DOG has only 5, the comparison would lack fairness. Moreover, these drivers and any other tools are developed and maintained by the open-source community, thus usually cannot be found in DOG and openHAB official releases.

Setup Step	Our Solution	DOG	openHAB
1 Installation			
1.1 Obtain the Solution	✘	✘	✘
1.2 Obtain Desired Drivers	✘	✘	✘
1.3 Compile Source Code		✘	
2 Develop Drivers			
2.1 Learn the Development API	✘	✘	✘
2.2 Develop Re-usable TCP/IP Transport Driver	✘		
2.3 Develop KNX Gateway Driver	✘	✘	✘
2.4 Develop KNX TCP/IP Transport Driver		✘	✘
2.5 Develop DCOM Gateway Driver	✘	✘	✘
2.6 Develop DCOM TCP/IP Transport Driver		✘	✘
2.7 Develop iLight Gateway Driver	✘	✘	✘
2.8 Develop iLight TCP/IP Transport Driver		✘	✘
2.9 Develop Arduino Gateway Driver	✘	✘	✘
2.10 Develop Arduino TCP/IP Transport Driver		✘	✘
2.11 Develop ENORO Gateway Driver	✘	✘	✘
2.12 Develop ENORO TCP/IP Transport Driver		✘	✘
2.13 Develop Specific Device Drivers	✘	✘	✘
3 Configuration			
3.1 Learn How to Individually Configure each Driver		✘	✘
3.2 Create Configuration Files		✘	✘
3.3 Defining House Plant Semantics		✘	✘
3.4 Defining Icons and Other GUI Related Items		✘	✘
3.5 Edit Single Configuration File	✘		
4 Develop Client Applications			
4.1 Learn How to Use the REST API	✘	✘	✘
4.2 Develop the User Feedback Server	✘	✘	✘
4.3 Develop the Office Control UI	✘	✘	✘
4.4 Develop the Energy Dashboard	✘	✘	✘
4.5 Develop the Daylight Harvesting Controller	✘	✘	✘

Table 7.16: Comparison of the setup steps needed for each solution to achieve the requirements of the SmartCampus Project. These steps are sorted chronologically. In this analysis we consider that no solution starts having drivers already implemented.

7.2.2 Mobile Robot

In this Section we ascertain that our solution can be applied to scenarios which are remarkably different from Building Automation, scenarios where none of the analyzed solutions could fit without core modifications. We have set up an experiment where our solution must be able to control a mobile robot.

The mobile robot built for this case study, depicted in Figures 7.2 and 7.3, consists of a chassis holding a sonar, one motor driver attached to two electrical motors and their respective wheels, one Arduino board, one Raspberry Pi and a power source fueling the circuitry. The sonar, interfaced by an Arduino connected through USB, will be used to detect any obstacles within the maximum range of 2 meters. The Raspberry Pi will have our framework embedded which will preform the computation task. Finally using the general-purpose I/O (GPIO) ports of Raspberry Pi it will control the driver integrated circuit (IC) to drive the wheel motors.

In this example we created a simple control loop which polls the sonar for any obstacles within a 30cm range and stops the motors while the obstacle is still present. The application performing the control loop further exposes a minimalist REST API, providing ways to control the two wheels and to read from the sonar to third-party applications (Figure 7.4). In order to apply our solution to this scenario we had to develop two transport drivers, one for the serial communication with Arduino (consisting of a virtual RS232 link) and one for the Raspberry Pi's GPIO interface. These transport drivers are then

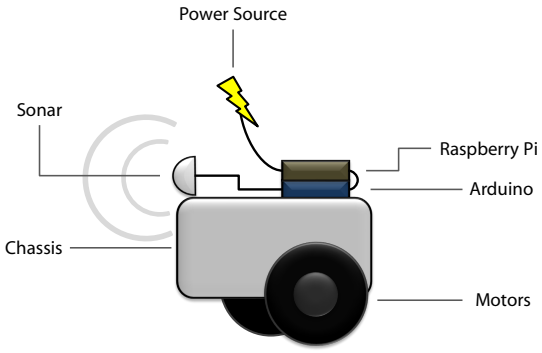

Figure 7.2: Graphical overview over the robot built for this case study.

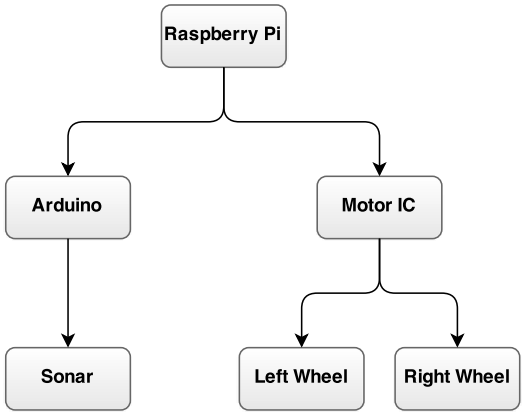

Figure 7.3: Schematic of the system within the robot.

used by the Arduino Gateway Driver which communicates with the Arduino board in order to get sonar readings exposed into one read-only datapoint, and a motor driver IC Gateway Driver which exposes datapoints to control the robot wheels. Then we have a device drivers exposing the sonar and the robot wheels to external applications. Finally, atop of our framework, we can find the application responsible for sustaining the control loop.

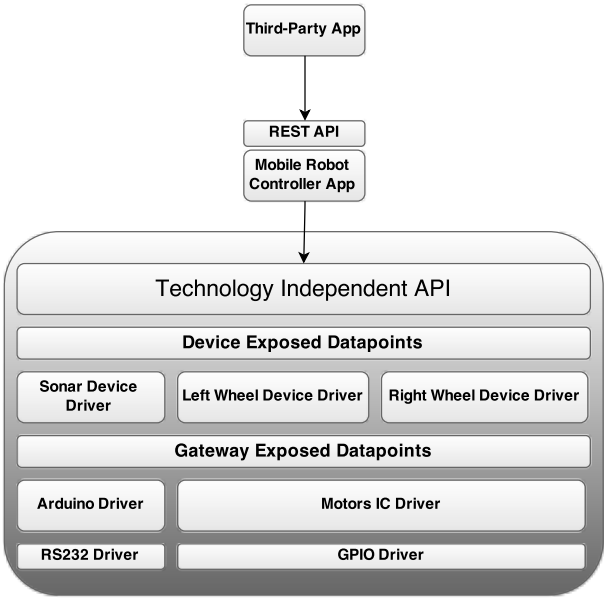

Figure 7.4: Diagram illustrating the application of our solution into the mobile robot scenario. At the bottom we have our framework extended with the necessary gateway and transport drivers. Atop of our framework is the control application, while sustaining a control loop also exposes the orchestrated sensors and actuators to third-party applications through a REST API. The Configuration Loader application was not depicted in order to simplify this Figure.

For some, the application of our framework may be perceived as a complicated solution for a minimalist problems like the one in this case study. However, it is also evident that for more complex robots, such as humanoid robots—usually having hundreds of sensors and actuators—, our framework can

provide numerous benefits. Additionally, in the situation where the robot needs to replace some electrical components, or the robot itself needs to be replaced, the loop control algorithm can still perform as well in the new circumstances, since the technological independent API is independent of any hardware modifications. This might be useful for (i) abstracting the control algorithm from any hardware specific dependencies, thus simplifying the process of development of sophisticated control algorithms, and (ii) promoting a modular software design where the control algorithm is understood as a context-independent module, passive of being re-used in future projects.

Chapter 8

Conclusions

The beginning of this work systematized fundamental aspects of BAS, contributing to a common understanding of fundamental Building Automation concepts aligned with the well known standards ISO 16484-3 [43] and EN 15232 [48]. Using these standards as guidelines this work highlights the scope of the functionality that should be expected from the typical BAS.

We assess the industry standard Building Automation technologies in terms of functional requirements employing an uniform terminology. Moreover, this study also provides a detailed mapping of features between existing technologies according to their official specifications and identifies their functionality gaps. It becomes clear that no single technology fully covers all the standard functionality expected from a BAS, thus requiring posterior, custom made, feature additions—a state of affairs that is among the main causes of heterogeneity in building automation.

After a deep analysis of the current market attempts in solving this issue we arrived at the conclusion that they fall short for several aspects such as not following any known terminology, having complex architectures, inability to fully abstract software developers from the underlying technologies, and even lack of documentation. For this, device heterogeneity is still considered an issue to this date.

To solve this problem this work presents a technology-agnostic integration framework capable of bridging different automation technologies, providing an abstraction layer to developers wishing to create Building Automation software. Our approach is unique in the sense that it (i) provides a smaller architecture capable of performing the integration of different automation technologies, (ii) strongly promotes modules re-usability, hence simplifying the development task, (iii) can be applied to numerous different contexts other than simply home or Building Automation, without core modifications to the architecture, (iv) accelerates the development of third-party applications due to the layer abstracting the underlying technologies specificities, and (v) further simplifies the driver development process by natively injecting every driver with functionalities which are pertinent to every driver, independent of the targeted technology, avoiding developers from having to repeat code, hence reducing the margin for producing mistakes.

To demonstrate that our framework outperforms any existing solutions, there were two validation stages: the first one consisted on a qualitative and quantitative architectural analysis against cur-

rent state of the art solutions, the second one consisted of two significantly complex case studies. The first one was performed within the scope of the European Project SmartCampus—a production environment—, where one of the requirements was orchestrating the heterogeneous environment found in IST Taguspark building, while developing the necessary applications to control the building. The other scenario consisted on commanding a mobile robot employing a control loop querying a sonar for obstacles in the path and actuating accordingly to avoid crashing into the obstacle, confirming the versatility of our solution. Within the scope of the SmartCampus project our solution demonstrated a great potential in the application of energy efficiency techniques and in the development of applications to achieve user behavior transformation and to revitalize old Building Automation hardware installations.

For these reasons we firmly believe that the solution proposed herein will help explore every Building Automation System's full potential, supporting the implementation of new solutions to better serve building occupants, while simultaneously providing benefits such reduced energy consumption. We expect to assist an exponential growth of technology-agnostic BA advanced solutions accompanied with reduced installation costs, by providing a state of the art solution which brings together different technologies. These advanced solutions can help building managers to closely monitor and control the building's activity, while allowing novel control systems, having strongly developed AI—which is only feasible when hardware details are ignored—, able to perform complex decisions that no market solution today is able to execute.

8.1 Future Work

Despite the fact that our proposed solution excels at several aspects, it still needs a few enhancements in order to end-users take full advantage of the minimalist abstraction layer it provides:

- **Driver abundance**
  We need to develop and provide a plethora of transport, gateways and device drivers, in order to cover the most common technologies used nowadays in building and Home Automation. By having an enriched catalog of drivers to choose, developers will have to perform less work—ideally none—to integrate any infrastructure using our solution.

- **Configuration GUI**
  In order to provide a pleasant user-experience we need to feature a GUI to enable users to configure the platform. Currently all configuration is performed by manually editing a single text file. However, taking advantage of the fact that all technologies are abstracted, it is possible to develop a GUI which enables the homogeneous configuration of any integrated technology. The concept of a configuration GUI is already employed in openHAB.

- **Creation of high-level applications**
  We believe in the importance of an application which enables the end-user to customize a set of scenarios, timers and schedule events. Although our framework integrates several automation networks, it does not deal with such high-level details regarding building automation. These concepts

strictly belong to the automation field hence constraining our application solely to automation scenarios. However, using our abstraction layer, which abstracts applications from hardware idiosyncrasies, it is straightforward to handle such high-level notions by software. This will also provide the end-user with ways to modify scenarios, timers and schedules at will, having no need to have a technician to modify the underlying technologies commission configurations—as it happens with technologies such as iLight.

- **Application marketplace**

  One of the main goals of creating a technology-agnostic framework for Building Automation is to uniformize the way applications are developed in this field. Having an increased community, this project may eventual lead to the creation of a market place, similar to mobile application stores, where numerous developers share their applications. These applications will benefit from the hardware abstraction layer our solution provides, being able to perform in any building without having to deal with interoperability issues.

Bibliography

[1] M.R. Brambley, D. Hansen, P. Haves, D.R. Holmberg, S.C. McDonald, K.W. Roth, and P. Torcellini. Advanced sensors and controls for building applications: Market assessment and potential r&d pathways, pnnl-15149. Technical report, Prepared for the U.S. Department of Energy by Pacific Northwest National Laboratory, 2005.

[2] Hermann Merz, Thomas Hansemann, Christof Hübner, James Backer, Viktoriya Moser, and Leena Greefe. *Building Automation: Communication systems with EIB/KNX, LON and BACnet (Signals and Communication Technology)*. Springer, 2009.

[3] W. Granzer and W. Kastner. Information modeling in heterogeneous building automation systems. In *Proceedings of the 9th IEEE International Workshop on Factory Communication Systems*, 2012.

[4] G. Clarke and D. Reynders. *Practical modern SCADA protocols: DNP3, 60870.5 and related systems (IDC Technology)*. Newnes, 2004.

[5] F.F. Wu, K. Moslehi, and A. Bose. Power system control centers: Past, present, and future. *Journal of Proceedings of the IEEE*, 93:1890–1908, 2005.

[6] Steve Mackay, John Park, and Edwin Wright. *Practical Data Communications for Instrumentation and Controls*. Elsevier, 2003.

[7] I.N. Fovino, A. Carcano, and M. Masera. A secure and survivable architecture for SCADA systems. In *Proceedings of the 2nd International Conference on Dependability*, 2009.

[8] Keith A. Stouffer and Joseph A. Falco and Karen A. Scarfone. SP 800-82. Guide to Industrial Control Systems (ICS) Security: Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control Systems (DCS), and other control system configurations such as Programmable Logic Controllers (PLC). Technical report, National Institute of Standards & Technology, 2011.

[9] G. Zecevic. Web based interface to SCADA system. In *Proceedings of the International Conference on Power System Technology*, 1998.

[10] J.P. Lahti, A. Shamsuzzoha, and T. Kankaanpaa. Web-based technologies in power plant automation and SCADA systems: A review and evaluation. In *Proceedings of the IEEE International Conference on Control System, Computing and Engineering*, 2011.

[11] ISO 16484-2. *Building automation and control systems (BACS) - Part 2: Hardware*, 2004.

[12] Gunnar Björkman, Teodor Sommestad, Hadeli Hadeli, Kun Zhu, and Moustafa Chenine. SCADA system architectures. Technical report, European Community: 7th Framework Programme, 2010.

[13] Toby Considine and Brian Frank. OBIX Version 1.1. Technical report, OASIS Open Building Information Exchange TC, 2010.

[14] J. P. Thomesse. Fieldbuses and interoperability. In *Control Engineering Practice, Vol. 7, Iss. 1*, pages 81–94, 1999.

[15] E. Finch. Is ip everywhere the way ahead for building automation? In *Facilities, Vol. 19, Iss. 11/12*, pages 396–403, 2001.

[16] Ulrich Schrittesser Armin Veichtlbauer, Thomas Pfeiffenberger. Generic control architecture for heterogeneous building automation applications. In *SENSORCOMM 2012: The Sixth International Conference on Sensor Technologies and Applications*, pages 148–153, 2012.

[17] Walter Buckley. *Systems Research for Behavioral Science: A Sourcebook*. AldineTransaction, 2008.

[18] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback and Control of Dynamic Systems, Sixth Edition*. Prentice Hall, 2010.

[19] Norman S. Nise. *Control Systems Engineering, Sixth Edition*. Wiley, 2010.

[20] W. Kastner, G. Neugschwandtner, S. Soucek, and H.M. Newmann. Communication systems for building automation and control. *Journal of Proceedings of the IEEE*, 93:1178–1203, 2005.

[21] Electronic Industries Alliance - Government Electronics and Information Technology Association Engineering. *Processes for Engineering a System*, 1999.

[22] Consumer Electronics Association (CEA). *CEA-852 Details, ISO-16484, Edition B*, 2010.

[23] A. Fernbach, W. Granzer, and W. Kastner. Interoperability at the management level of building automation systems: A case study for BACnet and OPC UA. In *Proceedings of the IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA)*, 2011.

[24] Peter Pallinger and László Kovács. Ili: A framework for implementing smart spaces. ERCIM News, 2011.

[25] Jean-Pierre Thomesse. Fieldbus technology in industrial automation. *Journal of Proceedings of the IEEE*, 93:1073 – 1101, 2005.

[26] D. Dietrich and T. Sauter. Evolution potentials for fieldbus systems. In *IEEE International Workshop on Factory Communication Systems*, 2000.

[27] Moore Hawke. Introduction to fieldbus. Technical report, Moore Industries-International Inc., 2006.

[28] ISO 16484-5. *Building automation and control systems (BACS) - Part 5: Data communication protocols*, 2003.

[29] ISO/IEC 14543-3-1. *Information Technology – Home Electronic Systems (HES) Architecture – Part 3-1: Communication layers – Application layer for network based control of HES Class 1*, 2006.

[30] ISO/IEC 14908-1. *Information Technology - Control Network Protocol - Part 1: Protocol stack*, 2012.

[31] MODBUS. *Application Protocol Specification V1.1b3*, 2012.

[32] ZigBee Alliance, Inc. *ZigBee Specification*, 2008, Available at `http://home.deib.polimi.it/cesana/teaching/IoT/papers/ZigBee/ZigBeeSpec.pdf`.

[33] IEEE 802.15.4. *Standard for Local and Metropolitan Area Networks*, 2011.

[34] EnOcean GmbH. *EnOcean Radio Protocol Specification V1.0*, 2012.

[35] ISO/IEC 14543-3-10. *Information Technology - Home Electronic Systems (HES) - Part 3-10: Wireless Short-Packet (WSP)*, 2012.

[36] IEC 61131. *Programmable controllers*, 2003.

[37] J.M. Sousa, R. Babuška, and H.B. Verbruggen. Fuzzy predictive control applied to an air-conditioning systems. *Control Engineering Practice*, 5:1395–1406, 1997.

[38] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006, ISBN = 0596005652.

[39] H. Himmelbauer. SOAP interface for an internet/fieldbus gateway. Master's thesis, Technical University of Vienna, 2006.

[40] Digi International Inc. *Dynamic C - An Introduction to ZigBee*, 2008.

[41] E.U. Warriach, E. Kaldeli, J. Bresser, A. Lazovik, and M. Aiello. Heterogeneous device discovery framework for the smart homes. In *Proceedings of the IEEE GCC Conference and Exhibition*, 2011.

[42] Echelon Corporation. *LONMARK Application Layer Interoperability Guidelines*, 2001.

[43] ISO 16484-3. *Building automation and control systems (BACS) - Part 3: Functions*, 2005.

[44] ISO 16484-1. *Building automation and control systems (BACS) - Part 1: Project specification and implementation*, 2010.

[45] KNX Association. *KNX Commissioning ETS4*, 2013.

[46] A. Wagner, E. Gossauer, C. Moosmann, Th. Gropp, and R. Leonhart. Thermal comfort and workplace occupant satisfaction—results of field studies in german low energy office buildings. *Energy and Buildings*, 39:758–769, 2007.

[47] Richard de Dear and G. S. Brager. Developing an adaptive model of thermal comfort and preference. Technical report, Center for Environmental Design Research, University of California, Berkeley, 1997.

[48] GreenBuilding Programme. prEN 15232: Energy performance of buildings — Impact of Building Automation Control and Building Management. Technical report, EU Directive 2002/91/EC on the Energy Performance of Buildings, 2006.

[49] S. Ramos, J. M. M. Duarte, J. Soares, Z. Vale, and F. J. Duarte. Typical load profiles in the smart grid context; a clustering methods comparison. In *Proceedings of the IEEE Power and Energy Society General Meeting*, 2012.

[50] Hyun Lee, Jae Sung Choi, and R. Elmasri. A conflict resolution architecture for the comfort of occupants in intelligent office. In *Proceedings of the 4th International Conference on Intelligent Environments*, 2008.

[51] Thu Ba. BACnet and KNX. `http://www.emate.com.vn/newsprint.aspx?id=427`, 2011. [Online; accessed 14-July-2014].

[52] Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. KNX and OPC UA. Technical report, Automation Systems Group, Vienna University Of Technology, 2011.

[53] Konnex Association. *KNX System Specifications*, 2009.

[54] KNX Association. *KNX Advanced Course: Interworking*, Home and Building Management Systems.

[55] Echelon Corporation. *LonTalk Protocol Specification*, 1999, Available at `https://support.dce.felk.cvut.cz/pub/hanzalek/_private/ref/LonTalk.PDF`.

[56] Echelon Corporation. *LONMARK SNVT Master List*, 2012, Available at `http://www.lonmark.org/technical_resources/resource_files/snvt.pdf`.

[57] Echelon Corporation. *Functional Profile: Switch*, 1997, Available at `http://www.lonmark.org/technical_resources/guidelines/docs/profiles/3200_10.pdf`.

[58] Echelon Corporation. *Functional Profile: Lamp Actuator*, 1997, Available at `http://www.lonmark.org/technical_resources/guidelines/docs/profiles/3040_10.PDF`.

[59] Echelon Corporation. *Introduction to the LONWORKS System*, 1999, Available at `http://downloads.echelon.com/support/documentation/manuals/general/078-0183-01a.pdf`.

[60] Robert Faludi. *Building Wireless Sensor Networks with ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, 2011.

[61] ZigBee Alliance. *ZigBee Cluster Library Specification*, 2008, Available at `http://www.everyhue.com/wp-content/uploads/2012/12/ZigBee_Cluster_Library.pdf`.

[62] ASHRAE - American Society of Heating, Refrigerating and Air-Conditioning Engineers. *BACnet A Data Communication Protocol for Building Automation and Control Networks (Standard 135-2004 – ANSI Approved)*, 2004.

[63] Carrier Corporation. *BACnet Basics User's Guide*, 2010, Available at `http://dms.hvacpartners.com/docs/1000/Public/04/11-808-417-01.pdf`.

[64] Bill Swan, Alerton Technologies, Inc. The Language of BACnet-Objects, Properties and Services. `http://www.bacnet.org/Bibliography/ES-7-96/ES-7-96.htm`, 1995. [Online; accessed 14-July-2014].

[65] David M. Underwood David M. Schwenk, Stephen J. Briggs and Joseph Bush. Development of an open building automation system specification based on ANSI/ashrae 135-2004 (BACnet communications protocol). Technical report, US Army Corps of Engineers, 2007.

[66] Chipkin Automation Systems. BACnet: The Schedule Object. `http://www.chipkin.com/bacnet-the-schedule-object/`, 2007. [Online; accessed 14-July-2014].

[67] WAGO Kontakttechnik GmbH & Co. KG. Wago-i/o-system 750 BACnet - objects, properties, services, version 1.0.2. Technical report, WAGO, 2010.

[68] F. Jammes and H. Smit. Service-oriented paradigms in industrial automation. *Journal of IEEE Transactions on Industrial Informatics*, 1:62 – 70, 2005.

[69] ASHRAE - American Society of Heating, Refrigerating and Air-Conditioning Engineers. *Proposed Addendum ar to Standard 135-2010, BACnet - A Data Communication Protocol for Building Automation and Control Networks*, 2010.

[70] ASHRAE. *Proposed Addendum ar to Standard 135-2010, BACnet - A Data Communication Protocol for Building Automation and Control Networks*, 2004.

[71] Stefan Szucsich. Web services in building automation with focus on BACnet/ws. Technical report, Vienna University of Technology, 2010.

[72] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner. Web services in building automation: Mapping KNX to BACnet/ws. In *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, 2007.

[73] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer, 2009.

[74] Broadwin Technology, Inc. *OPC Device Driver Guide - WebAccess Configuration Manual*, 2003.

[75] OPC Foundation. *OPC UA Part 1 - Overview and Concepts 1.01 Specifications*, 2009.

[76] OPC Foundation. *OPC UA Part 3 - Address Space Model 1.01 Specification*, 2009.

[77] OPC Foundation. *OPC UA Part 5 - Information Model 1.01 Specifications*, 2009.

[78] A. Maka, R. Cupek, and J. Rosner. OPC UA object oriented model for public transportation system. In *Proceedings of the Fifth UKSim European Symposium on Computer Modeling and Simulation*, 2011.

[79] ICONICS. *OPC-UA Defined and How It Impacts Automation*, 2009.

[80] OPC Foundation. *OPC UA Part 4 - Services 1-2.01 Specifications*, 2009.

[81] Edited by Bogdan M. Wilamowski and J. David Irwin. *The Industrial Electronics Handbook, Industrial Communications Systems*. CRC Press, 2011.

[82] OPC Foundation. *OPC UA Part 9 - Alarms and Conditions 1.00*, 2010.

[83] TAC Vista. *TAC Xenta Network Guide*, 2001.

[84] Matthias Neugschwandtner, Georg Neugschwandtner, and Wolfgang Kastner. Web services in building automation: Mapping KNX to obix. In *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, 2007.

[85] Echelon Corporation. *OpenLDV Programmer's Guide, Release 2.1*, 2005.

[86] Dario Bonino, Emiliano Castellina, Fulvio Corno. DOG: an Ontology-Powered OSGi Domotic Gateway. *20th IEEE International Conference on Tools with Artificial Intelligence*, 2008.

[87] Wang Haoyu and Zhou Haili. Basic Design Principles in Software Engineering. *Fourth International Conference on Computational and Information Sciences*, 2012.

[88] Paul Bakker & Bert Ertman. *Building Modular Cloud Apps with OSGi*. O'REILLY, 2013.

[89] David H. Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In *ECOOP'89*, pages 25–38, 1989.

[90] Kirk Knoernschild. *Java Application Architecture: Modularity Patterns with Examples Using OSGi (Robert C. Martin Series)*. Prentice Hall, 2012.

[91] Boci Lin, Yan Chen, Xu Chen, and Yingying Yu. Comparison between json and xml in applications based on ajax. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 1174–1177, Aug 2012.

[92] J.S. Hughes, D.J. Crichton, and C.A. Mattmann. Ontology-based information model development for science information reuse and integration. In *Information Reuse Integration, 2009. IRI '09. IEEE International Conference on*, pages 79–84, Aug 2009.

[93] P.R.S. Visser and Pepijn R. S. Visser and Dean M. Jones and T. J. M. Bench-capon and M. J. R. Shave. An Analysis of Ontology Mismatches; Heterogeneity Versus Interoperability. *AAAI Technical Report SS-97-06*, pages 164–172, 1997.

[94] Panagiotis Plastiras and Dympna O'Sullivan, Peter Weller. An Ontology-Driven Information Model for Interoperability of Personal and Electronic Health Records. *eTELEMED: The Sixth International Conference on eHealth, Telemedicine, and Social Medicine*, pages 130–133, 2014.

[95] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[96] W.A. Gruver. Distributed intelligence systems: A new paradigm for systems integration. In *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on*, pages nil14–nil15, Aug 2007.

[97] ISO/IEC 9126-1. *Information technology – Software product quality – Part 1: Quality model*, 2000.

[98] Eric J. Braude, Michael E. Bernstein. *Software Engineering: Modern Approaches: An Object-Oriented Perspective*. John Wiley & Sons, 2nd Edition edition, 2010.

[99] Muktamyee Sarker. An overview of Object Oriented Design Metrics. Technical report, Department of Computer Science, Umeå University, Sweden, 2005.

[100] Martin Hitz and Behzad Montazeri. Measuring Coupling and Cohesion In Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

[101] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20(6):476–493*, 1994.

[102] Varun Gupta and Jitender Kumar Chhabra. Package Coupling Measurement in Object-Oriented Software. *Journal of Computer Science and Technology*, 2:273 – 283, 2009.

[103] Shweta Sharma and Dr S. Srinivasan. A review of coupling and cohesion metrics in object oriented environment. *International Journal of Computer Science & Engineering Technology (IJCSET)*, 4:2229 – 3345, 2013.

[104] PerOlof Bengtsson. Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. *First Nordic Workshop on Software Architecture (NOSA'98), Ronneby*, 1998.

[105] McCabe, T.J. A Complexity Measure. *Software Engineering, IEEE Transactions on (Volume:SE-2, Issue: 4)*, pages 308 – 320, 1976.

[106] Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications Co., 2011.

[107] Echelon Corporation. *Introduction to the LonWorks Platform, revision 2*, 2009, Available at `http://downloads.echelon.com/support/documentation/manuals/general/078-0183-01B_Intro_to_LonWorks_Rev_2.pdf`.

Appendix A

OSGi

OSGi (Open Service Gateway initiative) is a specification describing a modular system and a service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments[1]. The most commonly used OSGi implementations are Apache Felix Framework[2] and Eclipse Equinox[3].

OSGi is not necessarily pointing towards the Building Automation domain, but it is the most adopted framework to develop solutions that tackle the problem of Building Automation device heterogeneity. Its success can mainly be ascribed to the following reasons [106]: (i) Reduced complexity, developing with OSGi technology means developing isolated components, known as bundles, which hide their internals from other bundles and communicate through well defined services. (ii) Reuse, the component model makes it very easy to use many third-party components in an application. An increasing number of open source projects provide their bundles ready to be used in different projects. (iii) Dynamic updates, the component model is dynamic in which bundles can be installed, started, stopped, updated, and uninstalled without halting the whole system. (iv) Versioning, in the OSGi environment all bundles are carefully versioned and only bundles that can collaborate are wired into the system run-time. (v) Lightweight, OSGi implementations usually consist of small binary files, enabling it to operate on a large range of devices such as embedded devices, smart-phones and host-servers.

[1] OSGi Alliance Home Page: `http://www.osgi.org`
[2] Apache Felix Framework Home Page: `http://felix.apache.org/site/apache-felix-framework.html`
[3] Eclipse Equinox Home Page: `https://www.eclipse.org/equinox/`

Appendix B

Functionality Coverage

Technologies	Grouping/Zoning	Event Notification	Alarm Notification	Historical Data Access	Scheduling	Scenarios
BACnet	Provides group communication objects [2, p. 245]	Provides a Notification object type [2, p. 248,249]	Provides an Event object type and services for alarm notification [2, p. 243,244,254]	Provides a Trend Log object to communicate event logs, however there are no strict guidelines for the logging process. [2, p. 263]	Provides a Calendar and a Schedule object types [2, p. 241,250]	Scenarios can be implemented with the BACnet's Command object [28, Section 12.9]
KNX	Provides a Group object type [53, Section 4.2, 4.5]	N.A.	N.A.	N.A.	N.A.	Scenarios can be implemented by different Datapoint Types related to Scenes (e.g. DPT_SceneNumber, DPT_SceneControl) [54, Section 4.19]
LonWorks	Domains and subnets enable the creation of groups and zones [55, Chapter 3]	Provides SNVTs to handle events [56]	Provides SNVTs to model alarms [107, 56]	N.A.	Provides an SNVT to schedule events [56]	Provides SNVTs to manage scenarios [56]
ZigBee	Provides network group addresses and a cluster to manage groups [61, Chapter 3.6][32, 40]	N.A.	Provides an alarms cluster for sending and configure alarm notifications [61, Chapter 3.11]	N.A.	N.A.	The scenes cluster enables setting up and recalling scenarios [61, Chapter 3.7]
BACnet/WS	Devices can be logically grouped through the model's hierarchy [71]	Provides services to subscribe to event notifications [71]	Provides services to subscribe to alarm notifications [71]	N.A.	N.A.	N.A.
OPC UA	Although OPC does not define the concept of grouping, objects may be used to aggregate other objects [76, p. 75]	Monitored Items offer ways to subscribe to event notifications [80, Section 5.12]	OPC's specification defines an Information Model for Conditions and Alarms with acknowledgement capabilities [82, Chapter 4]	OPC tracks changes in variable attribute's values and in the system's address space [73, Section 2.11]	N.A.	N.A.
oBIX	oBIX objects may aggregate and reference other objects [13, Chapter 10]	Watches allow a client to subscribe to objects state's updates in which those changes are stored in cache until the subscribed clients acknowledge them [13, Chapter 13]	Supports the definition of alarms with acknowledgement. Alarms have to be associated to watch objects in order to cache their updates and notify the client each time it polls the server [13, Chapter 16]	oBIX offers an interface that turns objects into traceable objects whose attributes will be stored by the history record service [13, Chapter 15]	N.A.	N.A.

Table B.1: Mapping of feature support in the analyzed automation technologies, depicting how each feature is described in each technology official specification.

Appendix C

SmartCampus Project

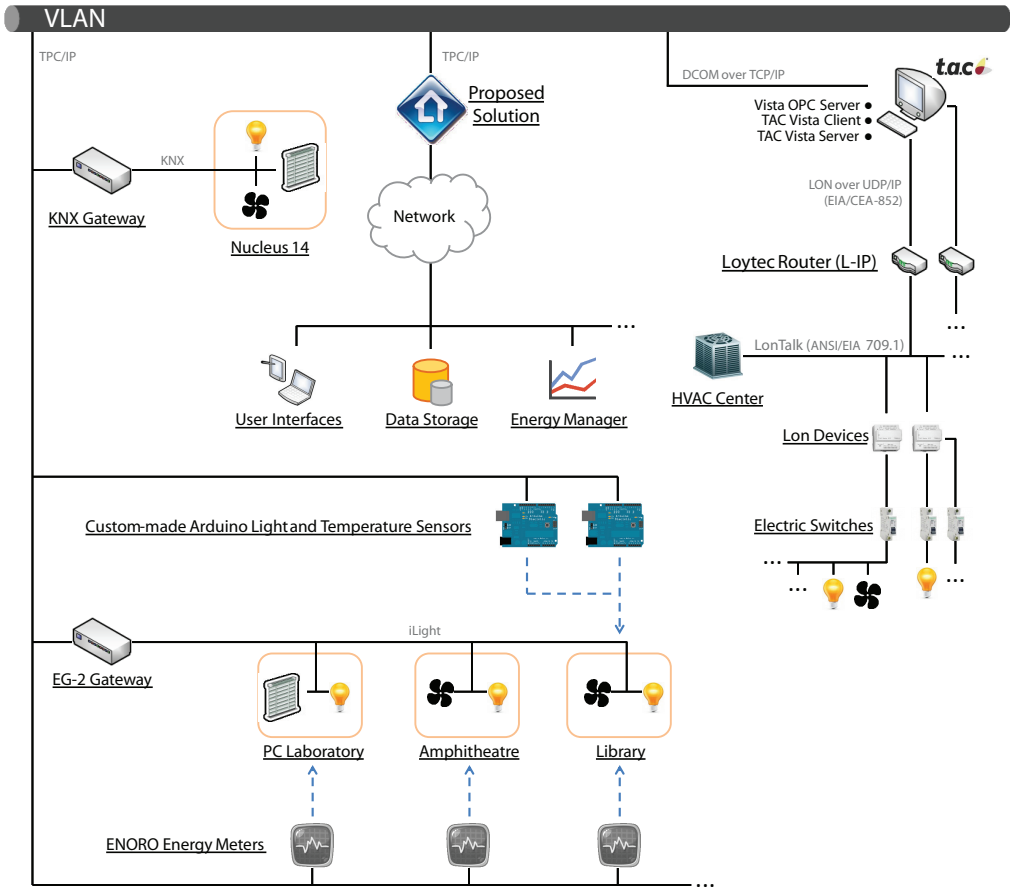

Figure C.1: IST Taguspark infrastructure consisting of a VLAN TCP/IP backbone connecting several automation gateways and devices. At the left we have the KNX network covering all offices in Nucleus 14, Arduino sensors measuring the light intensity and temperature from the library, the iLight network to control the lights and ventilation of three zones, and for each zone a energy meter measuring their energy consumption. At the right we have the LonWorks network, which is exposed as an OPC server, enabling full control of the entire building. Finally we have our proposed solution which communicates with the automation networks through the VLAN and exposes a homogeneous access to these automation networks by third-party applications.

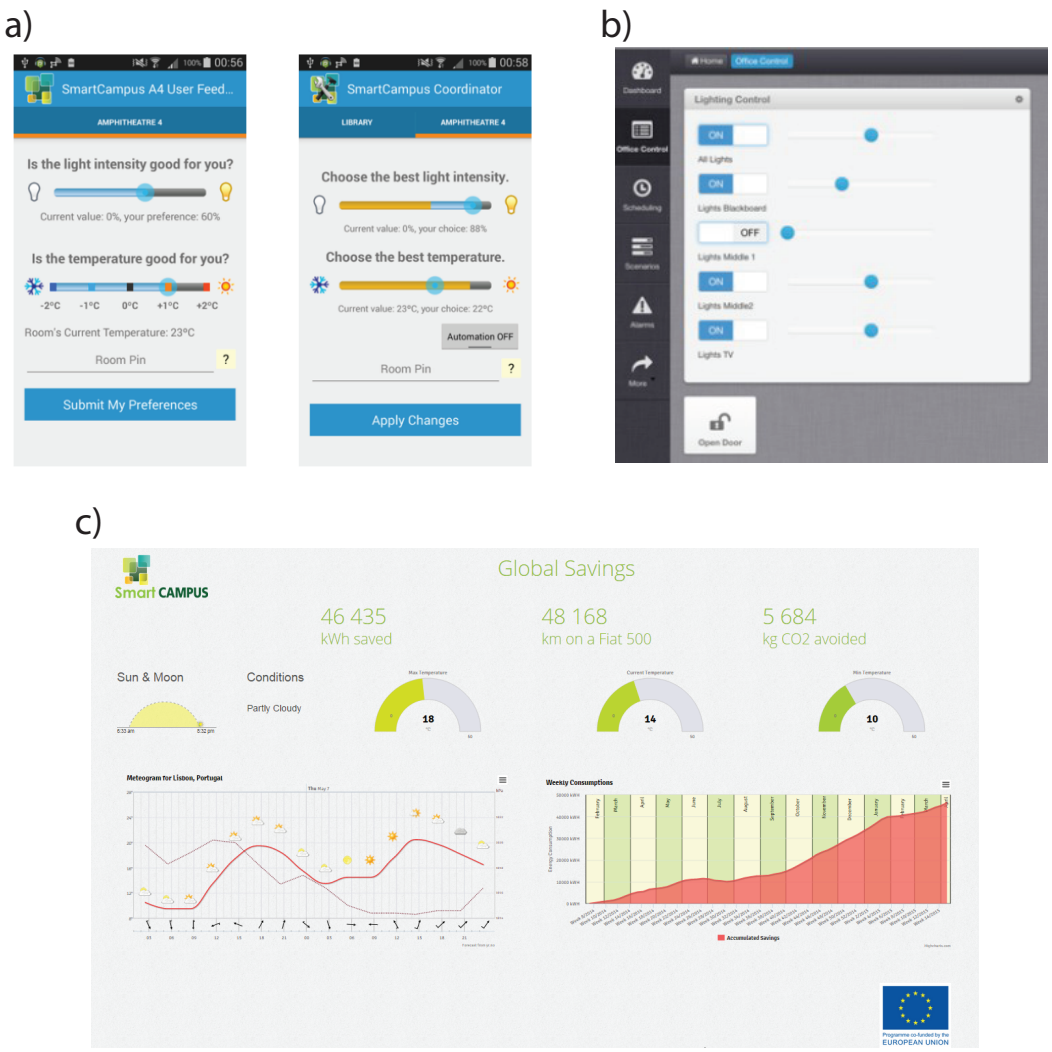

Figure C.2: Some applications developed in the scope of the SmartCampus Project, using our proposed solution as an abstraction layer separating these applications from the underlying automation technologies. (a) The mobile application that receives users' feedback in form of temperature and light intensity votes. (b) The office control application which enables the user to control the office's lights, blinds and HVAC. (c) The energy consumption dashboard for the library, class amphitheater and some computer laboratories of IST Taguspark.

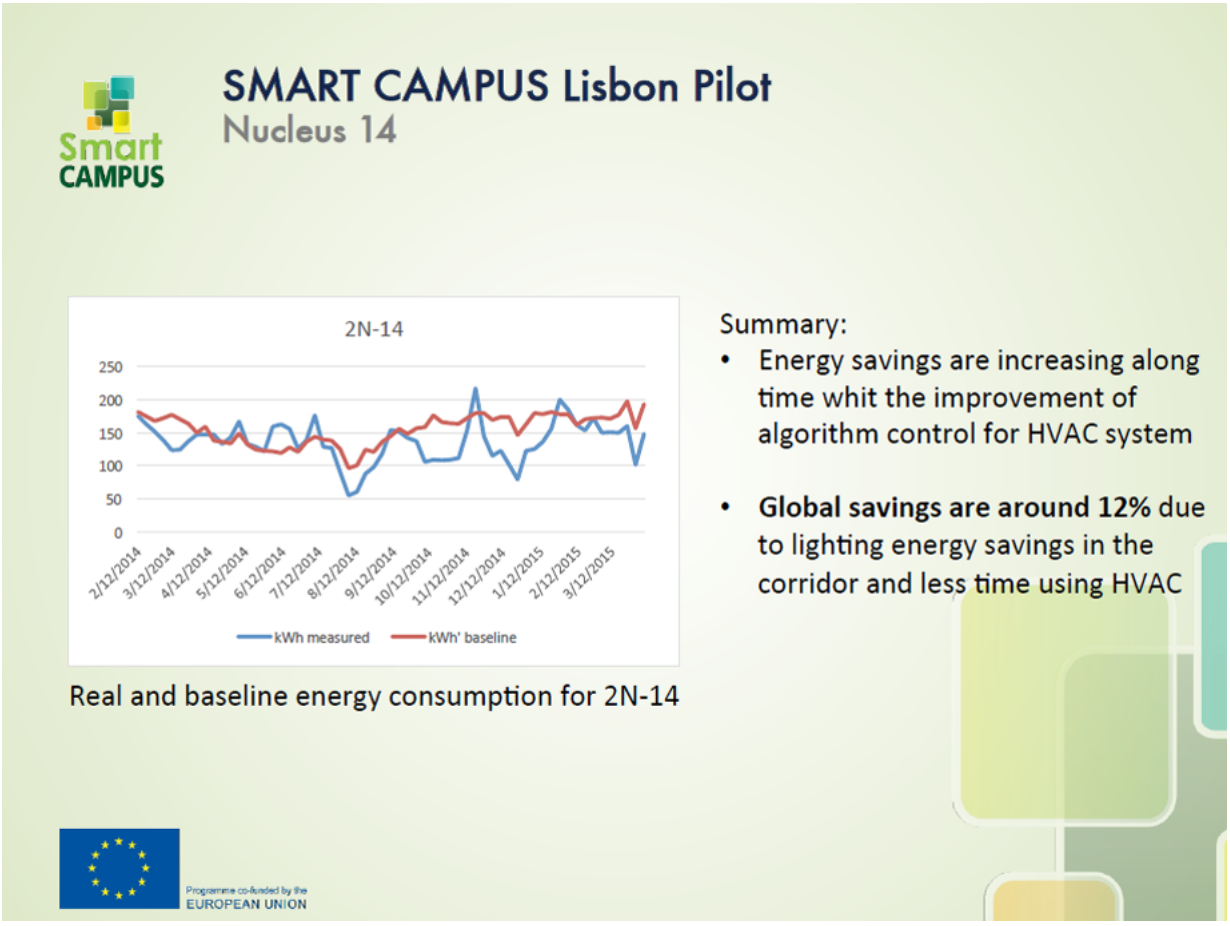

Figure C.3: SmartCampus Energy Efficiency Results in IST Taguspark Nucleus 14, showing energy savings in the order of 12% through the implementation of algorithms atop of our framework.

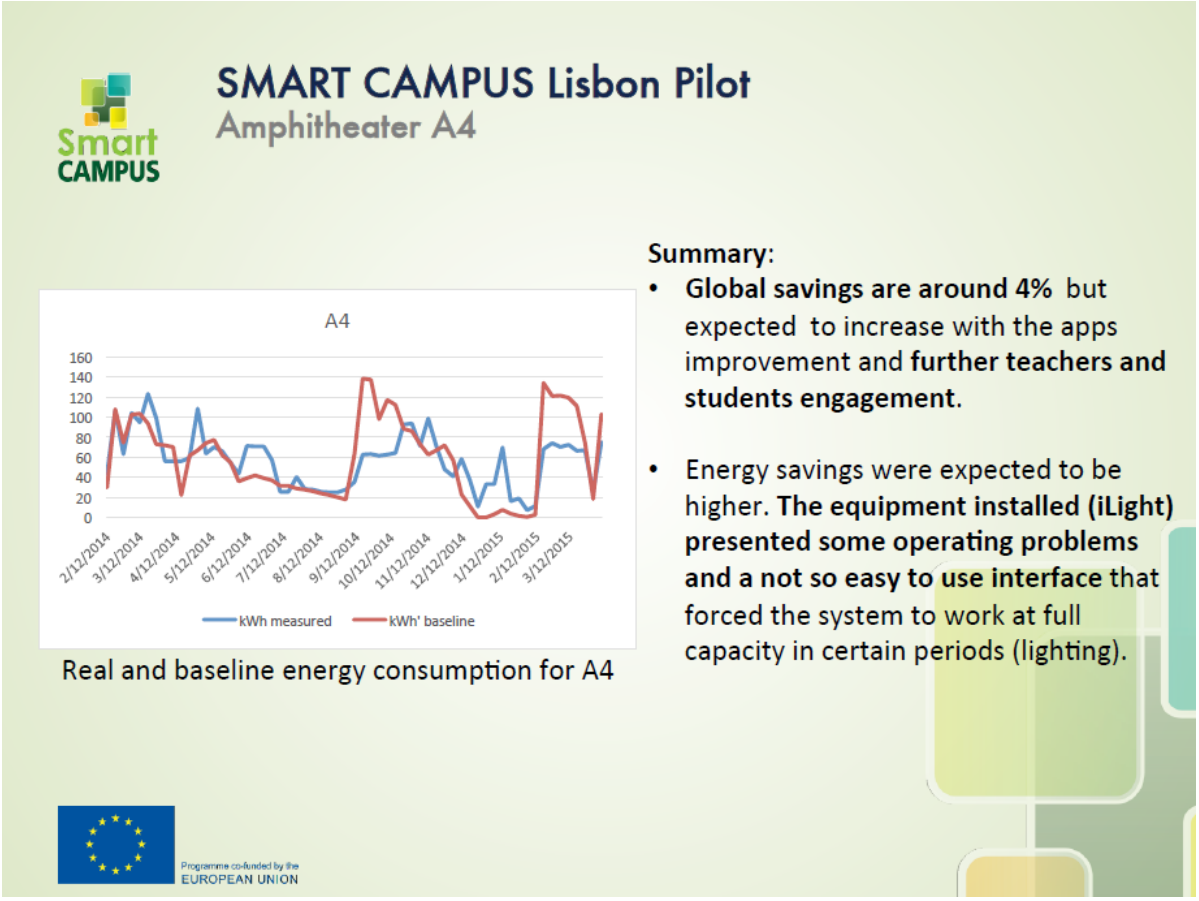

Figure C.4: SmartCampus Energy Efficiency Results in IST Taguspark Amphitheater 4, showing energy savings around 4% through the implementation of mobile applications atop of our framework, used to interact with users.

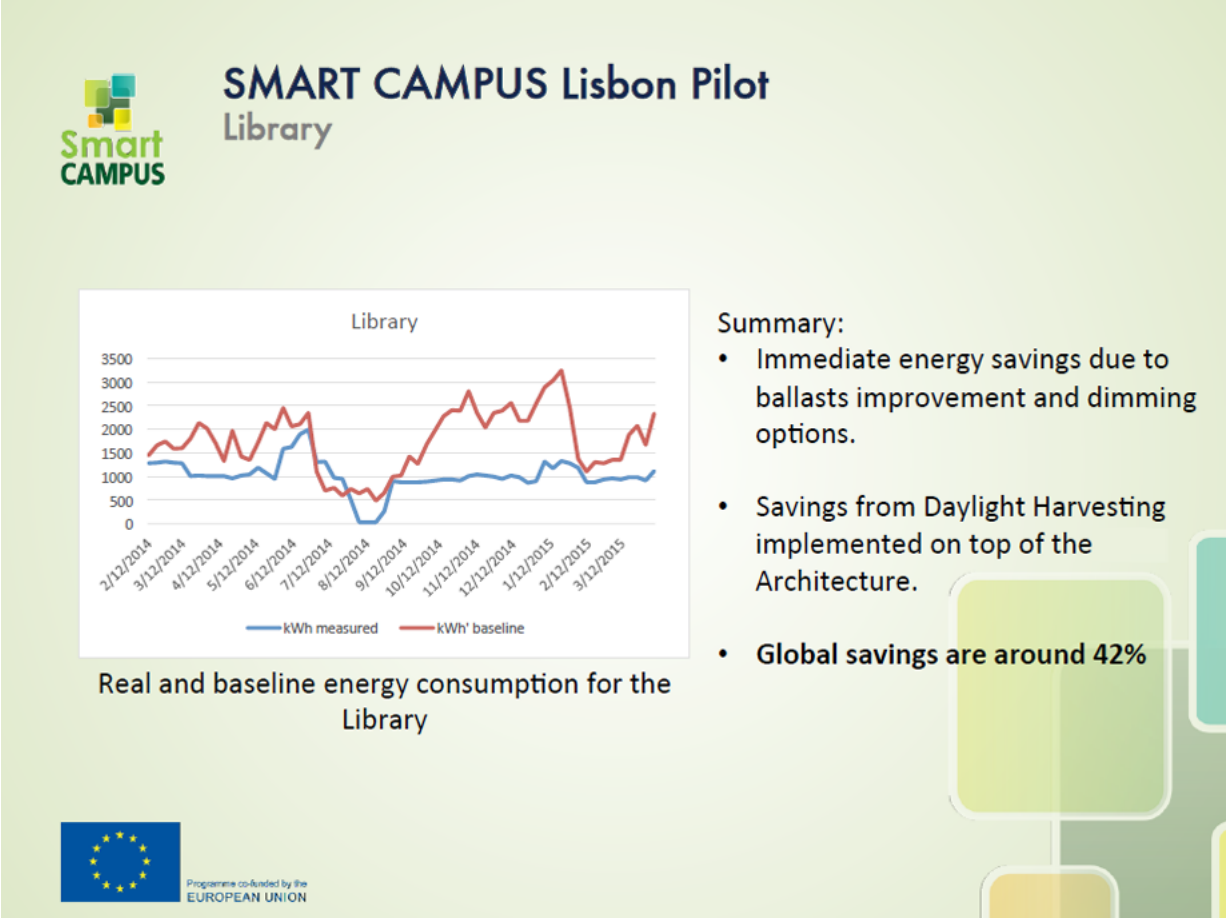

Figure C.5: SmartCampus Energy Efficiency Results in IST Taguspark Library, showing energy savings of 42% through the implementation of a daylight harvesting algorithm using our framework.