KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Application Centric Development in the Internet of Things

Guidelines and Tools for Software Integrators

Ilse Bohé

Supervisors:
Prof. dr. Vincent Naessens
Dr. ing. Jorn Lapon

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

September 2022

Application Centric Development in the Internet of Things

Guidelines and Tools for Software Integrators

Ilse BOHÉ

Examination committee:
Prof. dr. ir. Emmanuel Vander Poorten, chair
Prof. dr. Vincent Naessens, supervisor
Dr. ing. Jorn Lapon, co-supervisor
Prof. dr. ir. Hans Hallez
Prof. dr. Danny Hughes
Dr. ing. Michiel Willocx
Prof. dr. Bert Lagaisse
Dr. Bruno Van Den Bossche
(Docbyte)
Prof. dr. ir. Kris Steenhaut
(Vrije Universiteit Brussel)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

September 2022

Dankwoord

Nu het einde van mijn doctoraatstraject in zicht is wil ik enkele mensen bedanken. Zonder hen was ik nooit op dit punt geraakt.

Eerst en vooral wil ik prof. Vincent Naessens, mijn promotor, bedanken om in mij te geloven en me de kans te geven om aan mijn doctoraatstraject te starten. Jouw adviezen, ondersteuning en vooral ook de aanmoedigingen hebben me geholpen om er te blijven voor gaan en dit werk te kunnen voltooien.

Vervolgens wil ik mijn co-promotor, Jorn Lapon, bedanken. Jorn, bedankt om jouw kennis met mij te delen en me te steunen tijdens mijn werk, zelf op de momenten dat ik het zelf niet meer zo rooskleurig inzag. Het tweede deel van mijn doctoraat was zonder jou nooit zo interessant geweest. Maar nu weten we, älles kan".

Ik wil graag de leden van mijn begeleidings- en examencommissie bedanken. Bedankt prof. Hans Hallez, prof. Danny Hughes en dr. Bruno Van Den Bossche voor de feedback tijdens en na mijn tussentijdse verdedigingen. Bedankt ook aan mijn begeleidingscommissie en andere leden van mijn examencommissie, dr. Michiel Willocx, prof. Bert Lagaisse en prof. Kris Steenhaut, voor hun inzichten en suggesties. Bedankt aan prof. Emmanuel Vander Poorten voor het voorzitten van mijn doctoraatsverdediging.

Zonder mijn collega's was dit werk niet mogelijk geweest. Michiel, de afgelopen jaren zouden niet hetzelfde geweest zijn zonder jou. Eerst en vooral zou mijn doctoraat inhoudelijk niet staan waar het nu staat, maar de dagen zouden ook een stuk langer geweest zijn. Jij maakte mijn tijd als doctoraatsstudent onvergetelijk. Jan en Vincent, dankjewel voor de constructieve brainstormsessies en leuke tijden in het beginjaar van mijn doctoraat. Thomas, het was fijn om even je ëchte"collega te zijn. De vele koffietjes waren telkens een welkome pauze van het werk. Ik zou zeggen, laten we dit blijven doen. Dairo en Victor, ik kon me geen betere bureaumaatjes voorstellen, jullie zijn een waardige vervanger voor Michiel.

Ook wil ik de andere collega's bedanken die ik nog niet genoemd heb: Laurens, Stijn, Alexios, Karel, Ruben, Jonas, Kevin, Jenno, Celien, Alicia, Adriaan en alle andere onderzoekers van DistriNet en CODeS.

Tenslotte wil ik nog mijn vrienden en familie bedanken. Lieve vrienden en vriendinnen, dankjewel voor alle leuke ontspannende momenten de afgelopen jaren. Mariektje, dankjewel om er altijd voor mij te zijn, zonder jou zouden de weekends de afgelopen jaren maar leeg geweest zijn. Laten we nog vele jaren verder bouwen en knutselen. Bart en Gil, dankjewel voor alle leuke dagen, weekenden en vakanties de afgelopen jaren. Jullie steun en hulp in alle projecten die Matthijs en ik uitvoeren zorgt ervoor dat we er samen helemaal voor gaan. Lieve opa, oma, papa en Nathalie, ook al zien we elkaar niet veel, ik weet dat jullie me steunen, en dat gaf me een grote duw in de rug om verder te zetten. Mama, dankjewel voor alle steun. Jij bent mijn rolmodel. Jouw vriendelijkheid en positiviteit is wat ik de afgelopen jaren heb willen nastreven en wat ik zeker wil blijven doen in de toekomst. PS, bedankt dat we de afgelopen en komende maanden je huis mogen kapen. Lieve oma, dankjewel dat je me elke woensdag hebt opgewacht en ik bij je mag blijven logeren, de donderdagochtenden zijn de beste van de week. Liefste zussen, dankjewel om er altijd voor me te zijn, om eens samen ons hart te kunnen luchten. Liefste Thijsie, dankjewel om altijd in me te blijven geloven en mee alle ups en downs te beleven. Dankje voor alle liefde en humor die je me elke dag weer geeft.

Ilse Bohé
7 september 2022

Abstract

The Internet of Things (IoT) landscape is growing at a fast pace. In 2018, there were already 6.1 billion IoT devices which reflects 33 percent of the internet connected machines. Cisco estimates that the amount will grow to 14.7 billion connected IoT devices in 2023, or 50 percent of the connected devices. Despite the large amount of connected devices, a fully connected environment is not realistic in the near future. Almost every device comes with its own dedicated application. Communication between devices of different vendors, often based on different technologies, is a huge challenge.

Standardization efforts aim at circumventing this so-called *vendor lock-in* trap. However, the IoT Standards landscape is highly fragmented and evolves continuously. Hence, compliance with a particular standard seriously restricts the IoT device technologies that can be inserted in practice. Others apply *device centric* development for integrating devices from possible different vendors in the same application. Integrating new devices is then expensive as the entire software development cycle has to be re-run.

Moreover, IoT devices evolve at a fast pace degrading the attractiveness of many IoT applications over time. Also, the amount of sensors and actuators that are rolled out within a single IoT ecosystem may increase exponentially. Hence, sustainable IoT applications must cope with rapidly evolving hardware. Adaptability becomes a key concern when developing IoT applications.

Finally, many IoT applications, although often called smart, lack intelligence. At best, they embrace automation. Applying automatic device configuration, connection management, and user support for solving connectivity issues, is a complex endeavor for application developers. Likewise, integrating and enforcing policies in an IoT application is complex. Moreover, the dynamic nature of IoT systems complicates the development of applications that properly handle changes in the environment.

This PhD presents *application centric* development as an alternative approach to tackle the *vendor lock-in* and maintainability problems in advanced IoT ecosystems.

Firstly, a layered architecture is proposed that supports the design of advanced IoT ecosystems. Applying the architectural principles results in IoT applications

that can easily cope with new technologies that come to the market. On its turn, this increases the lifetime and offers various infrastructural alternatives to end-users. Both an Android and JavaScript framework are implemented to validate the approach. The inner workings of these frameworks are demonstrated by means of the development of an Ambient Assisted Living (AAL) environment.

The second part of this PhD shows that an early-stage ontological effort incorporating the application domain as well as infrastructural conceptualizations and relations can facilitate the development and management of IoT applications within verticals. Developers can now define application behavior in terms of application-domain conceptualizations, after which infrastructural feedback can automatically be extracted.

A last part presents a platform-independent middleware that simplifies the development of actual smart applications. The middleware hosts a modular, event-based logic reasoner, developed in Prolog, communicating with the underlying IoT framework. It not only supports basic automation, but also holds functionality for automatic device and connection management, access control and an abstraction module that decouples the applications from the underlying infrastructure. Moreover, the middleware leverages the actual benefits of Prolog, through complex querying and inference capabilities. This part also takes a closer look at a structural approach that brings access control to logic programming, called ACoP. It allows to constrain access to the knowledge base. The approach supports the use of impure predicates to prevent unauthorized side effects. The solution supports fine-grained access control using both deny and allow list strategies. Overhead is limited to defining access rules. The flexibility in expressing these rules allows to realize different access control mechanisms including role based, relationship based and attribute based access control. A prototype meta-interpreter in Prolog validates the presented approach.

This thesis finally explores validation paths targeting software integrators, and proposes both guidelines and software development tools.

Beknopte Samenvatting

Het *Internet of Things* (IoT) landschap groeit gestaag. In 2018 waren een derde van de met het internet verbonden computersystemen precies IoT-apparaten. Op dat moment waren ze reeds met meer dan 6 miljard. Cisco schat dat dit aantal in 2023 verder zal groeien naar ongeveer 15 miljard, goed voor maar liefst de helft van de apparaten die met het internet zijn gekoppeld op dat moment.

Er duiken wel een aantal uitdagingen op. Bijna elk apparaat wordt geleverd met een specifieke applicatie. Communicatie tussen apparaten van verschillende leveranciers, die vaak terugvallen op verschillende technologieën, is geen sinecure. Standaardisatie-inspanningen kunnen slechts deels de *vendor lock-in val* verzachten. Dit komt omdat IoT-standaarden erg gefragmenteerd zijn en continu evolueren. Focus op een welbepaalde norm kan de technologieën waarop in de praktijk kan worden teruggevallen ernstig beperken.

Anderen hanteren *sensorgerichte* ontwikkeling om diverse apparaten in eenzelfde toepassing te integreren. Het integreren van apparaten van alternatieve leveranciers is in dat geval vaak duur omdat een substantiële inspanning op vlak van softwareontwikkeling typisch vereist is.

Bovendien evolueren IoT-apparaten razendsnel, waardoor heel wat IoT-toepassingen op korte termijn minder aantrekkelijk worden. Daarenboven groeit het aantal sensoren en actuatoren dat wordt uitgerold binnen een enkele IoT-omgeving vaak exponentieel. Duurzame IoT-toepassingen moeten precies kunnen omgaan met snel evoluerende hardware. Dit vermogen om aan te passen doorheen de tijd wordt steeds belangrijker vanuit het standpunt van software integratoren.

Ten slotte missen veel IoT-applicaties intelligentie hoewel ze in de volksmond slim worden genoemd. In het beste geval omarmen ze een beperkte mate van automatisering. Voor applicatieontwikkelaars zijn automatische apparaatconfiguratie en intuïtieve ondersteuning voor het oplossen van verbindingsproblemen een complexe uitdaging. Ook het handhaven van beleidsregels in een IoT-toepassing is vaak erg complex. De dynamische aard van IoT-systemen bemoeilijkt dan weer de ontwikkeling van applicaties die gepast omgaan met veranderingen in de omgeving.

Dit doctoraat presenteert *applicatiegerichte* ontwikkeling als een alternatieve benadering om duuzame IoT-ecosystemen te realiseren.

Het eerste deel stelt een gelaagde architectuur voor die het ontwerp van geavanceerde IoT-ecosystemen ondersteunt. Het toepassen van doordachte architecturale principes resulteert in IoT-toepassingen die flexibel kunnen inspelen op nieuwe technologieën die op de markt komen. Het aanbieden van verschillende infrastructurele alternatieven aan integratoren en eindgebruikers verlengt de levensduur van toepassingen. Zowel een Android- als een JavaScript-*raamwerk* zijn geïmplementeerd om de aanpak te valideren. De ontwikkeling van een *Ambient Assisted Living (AAL)* omgeving demonstreert de werking van het raamwerk.

Het tweede deel vereenvoudigt de ontwikkeling en het beheer van IoT toepassingen door een ontologische inspanning tijdens de vroege ontwerpfase. De modellering van het toepassingsdomein, en infrastructurele conceptualiseringen en relaties staan hierbij centraal. Ontwikkelaars kunnen functionaliteit definiëren in termen van concepten in het applicatiedomein, waarna infrastructurele feedback automatisch wordt geëxtraheerd.

Een laatste deel presenteert een platformonafhankelijke middleware die de ontwikkeling van slimme applicaties vereenvoudigt. De middleware steunt op een modulaire, op *events* gebaseerde logica-redenering, ontwikkeld in Prolog, die communiceert met het onderliggende IoT-*raamwerk*. Het ondersteunt niet alleen basisautomatisering, maar bevat ook functionaliteit voor automatisch apparaat- en verbindingsbeheer, toegangscontrole en een abstractiemodule die de applicaties loskoppelt van de onderliggende infrastructuur. Bovendien maakt de middleware gebruik van de voordelen van Prolog, door middel van complexe query- en inferentiemogelijkheden. Dit derde deel gaat ook dieper in op een structurele benadering die toegangscontrole tot logische programmering brengt, genaamd ACoP. De aanpak ondersteunt het gebruik van onzuivere predikaten om te voorkomen dat ongeautoriseerde neveneffecten plaatsvinden.

Tenslotte worden validatiemogelijkheden ter ondersteuning van software integratoren voorgesteld. Zowel richtlijnen als softwareontwikkelingstools maken hier deel van uit.

List of Abbreviations

AAL Ambient Assisted Living. iv, xv, xvii, 8, 11, 13, 27, 46, 48, 50, 51, 53, 57, 60, 62, 64–67, 70, 71, 73–78, 83–85

ACL Access Control List. 116

AF Atrial Fibrillation. 3

AMQP Advanced Message Queuing Protocol. 21, 22

API Application Programming Interface. 1, 27, 32, 36, 52, 129

ASP Answer Set Programming. 104

BLE Bluetooth Low Energy. 18, 19, 26

CBOR Concise Binary Object Representation. 23–25

CoAP Constrained Application Protocol. 21, 22

COTS commercial-of-the-shelf. 8, 9, 12, 64, 124, 127

ECA Event-Condition-Action. 100

ECG Electrocardiography. 3

EU European Union. 65

FDA Food and Drug Administration. 3

GPS Global Positioning System. 33, 34, 36, 63

GUI Graphical User Interface. 5, 75, 134

HMI Human Machine Interface. xvii, 66, 69, 75–81, 84–86

HTTP Hypertext Transfer Protocol. 21, 22

IBAC Identity Based Access Control. 116, 117, 122

IIoT Industrial Internet of Things. 3, 13, 26

IoT Internet of Things. iii, iv, viii, xv, xvii, 1–16, 19–21, 23–29, 31–38, 40–45, 47, 48, 50–54, 56–70, 72, 73, 75, 77–100, 102, 122, 124, 125, 127–131, 133

IP Internet Protocol. 21, 45, 48

IT Information Technology. 1

JSON JavaScript Object Notation. 23, 24, 69

LAN Local Area Network. 17, 19

LPWAN Low Power Wide Area Network. 20, 61

M2M Machine to Machine. 25

MAC Media Access Control. 44, 45

MDM Mobile Device Management. 86

MQTT Message Queueing Telemetry Transport. 15, 21, 22, 47

NB-IoT Narrow-Band IoT. 20

NFC Near-Field Communication. 17, 18

OrBAC Organizational Based Access Control. 103

OS Operating System. 1

OWL Web Ontology Language. 104

PAN Personal Area Network. 17–19

PC Personal Computer. 1

QoS Quality of Service. 21, 22, 32, 33, 36, 37, 39, 52, 54, 56, 61, 62, 87

RBAC Rule Based Access Control. 103, 116, 117, 122

ReBAC Relationship Based Access Control. 103, 118, 122

REST Representational State Transfer. 21, 22, 27, 36, 47

RFID Radio-Frequency Identification. 17, 18

SDLC Software Development Life Cycle. xv, 2, 4–6, 123, 124, 128, 133

SME Small and Medium Enterprise. 126, 127

TCP Transmission Control Protocol. 21, 22

TLS Transport Layer Security. 22

UDP User Datagram Protocol. 22

UI User Interface. 47, 48

UML Unified Modeling Language. 5, 57

URI Uniform Resource Identifier. 21, 22

URL Uniform Resource Locator. 44

US United States. 3

USB Universal Serial Bus. 1

WAN Wide Area Network. 17, 20

XACML eXtensible Access Control Markup Language. 103–105

XML eXtensible Markup Language. 22–24, 69

XMPP Extensible Messaging and Presence Protocol. 21, 22

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Many companies are currently struggling in realizing and maintaining cost-efficient Internet of Things (IoT) ecosystems. A first major reason is the expensive implementation cycles due to the lack of high level sensor integration support for application developers. A majority of IoT sensors currently on the market only offer a low level Application Programming Interface (API). Hence, application programmers are confronted with low level connectivity and data format details for each type of sensor they want to integrate. Consequently, instead of focusing on business logic, they need to spend a lot of time and coding on sensor integration and data capturing. Second, problems arise after deployment. In many IoT ecosystems, sensors and actuators need to be replaced from time to time due to limited lifetime or harsh conditions in which they are deployed. Many IoT integrations offer no or at least very limited flexibility when sensors need to be replaced. For instance, a temperature sensor from one manufacturer can often not be replaced by another (more robust or cheaper) one from another manufacturer due to lack of flexibility during system design. Vendor lock-in is often mentioned as one of the major problems in current IoT deployments.

This PhD proposes a paradigm shift in designing IoT ecosystems. Today, a lot of trouble is caused due to device centric development. This means that sensors and actuators (i.e., infrastructural components) are selected in a very early stage, and thereafter, designers start to think about wrapping around software applications. However, the end-user applications typically survives the lifetime of low cost, on-the-edge IoT sensors. Moreover, the situation cannot be compared to traditional Information Technology (IT) system software in which peripherals are accessible system-wide (i.e., by all applications). Hence, (plug-and-play) support at Operating System (OS) level is already established for Personal Computer (PC) peripherals such as printers, Universal Serial Bus (USB) sticks and screens. On the contrary, the scope of IoT devices is often limited to a specific IoT application. Moreover, the diversity of IoT devices is much bigger than the diversity in workstation peripherals.

This PhD project focuses on *application centric* IoT ecosystem design. This implies that IoT components are only selected in a second step, and feasible architectural decisions must support flexibility with respect to sensor selection and integration after the IoT ecosystem is actually deployed.

This chapter first takes a closer look at the IoT landscape and its Software Development Life Cycle (SDLC) (Section 1.1). Subsequently, Section 1.2 elaborates the main research question and enumerates the main contributions. Projects that have been carried out in the context of the research and the publications that have arisen from this work are discussed in Section 1.3. This chapter ends with an overview of the remaining chapters.

1.1 Developing Internet of Things Applications

The IoT can be a a nice addition to make life more playful. A lot of IoT gadgets are on the market. Although they might appear to be fun and useful, some of them can be pretty useless. For example, in 2017 the Company Herb & Body, introduced Smalt, a smart salt shaker. Not only can you dispense salt from a button on your phone, you can also control it using Alexa, Amazon's virtual assistant [98, 106]. It can easily be concluded that this is not the most useful IoT application.

Apart from these IoT gadgets, IoT can really make a difference in society. Also during the covid pandemic, IoT solutions have been able to prove its feasibility. Contact tracing using Bluetooth proximity of personal devices, among other things, has proven to be a useful tool to quell the pandemic. But other health applications have also shown to be useful. By doing health monitoring by means of smart devices, patients can leave the hospital earlier without sacrificing appropriate follow-up. On its turn, this can decrease the pressure on the hospital and the health care system as a whole.

1.1.1 IoT in Different Domains

The IoT landscape is mostly known for its applications in the smart home environment. Smart lamps, smart door locks and smart heating systems have already become widespread in many households. However, IoT applications are not limited to this household setting. A lot of potential, both from a financial as well as a non-financial perspective (e.g., improved living standards) are clear in other settings.

McKinsey filed a report in which they define several impactfull IoT application domains [105].

Human & Health Care The first domain is the human and health care domain. A lot of wearable devices are already on the market, think about Garmin[1], FitBit[2] and the Apple Watch[3]. Apart from being cool gadgets, these watches can improve living standards a lot. Both the Apple and FitBit Electrocardiography (ECG) applications, that can perform heart rhythm assessments, are approved by the United States (US) Food and Drug Administration (FDA). Although these smartwatches can not be used as medical tool to diagnose Atrial Fibrillation (AF) – a heart rhythm disorder that increases the risk of a stroke – it can inform people of the possibilities and encourage them to seek medical advice. [83]

Home As mentioned before, the smart home environment is an already widely elaborated environment. Research shows that security and comfort functions (i.e., aid in chores) are most popular among households. From the perspective of house owners, keeping the energy cost under control is an important factor to introduce IoT devices in a smart living setting.

Retail In retail, IoT applications can be used to equip customers with personalized products. Contactless payment and cashierless shops are also heavily relying on IoT applications. The first steps in this direction have already been taken in Belgium. Colruyt has opened its first autonomous supermarket in Ghent in November 2021 [24]. Drones can also be used in the retail sector to autonomously deliver packages to customers. Similarly, Bpost, the Belgian postal company, has stated in their yearly activity report [3] that it is already performing extensive tests to introduce drone delivery in Belgium.

Office In the office, IoT applications can be used to simplify all kinds of tasks. Both access control and time management solutions are already established in many companies. Currently, a badge is often used to provide access to all kinds of rooms and appliances such as coffee machines and printers. Along with the covid pandemic, IoT applications have entered our society. Smart meeting cameras (e.g., OWL[4]) support hybrid meetings without the need for expensive static infrastructure. The Belgian company BARCO also introduces relevant IoT applications for the office on the market. For instance, their ClickShare[5] system allows to share screens without any cable hassle. Recent ClickShare versions also adapt to hybrid meetings and integrate peripherals such as sound and image sources.

Manufacturing and Industrial Internet of Things (IIoT) In the manufacturing sector, IoT also has a substantial impact. Monitoring production lines increases

[1] https://www.garmin.com/c/wearables-smartwatches/
[2] https://www.fitbit.com/
[3] https://www.apple.com/watch/
[4] https://owllabs.com/products/meeting-owl-pro
[5] https://www.barco.com/nl/clickshare

efficiency and hence has a positive impact on the competitiveness of a company. It implies that every step of the production process is monitored thereby relying on sensors. In the event of a malfunction, the right person can be informed quickly so that the production process can be restarted as quickly as possible. In some cases, malfunction can even be predicted before fall-out. An example IoT system that can be used for this is the plug and play system of VersaSense[6].

Vehicles Several smart vehicles are already on the market. These cars are connected to the Internet, so that all kinds of features can be monitored or controlled via an app. Simple button clicks can unlock the doors, switch the lights on or off and even control the heating remotely. This domain has the potential to have an impact at many levels. Road saftey increases by supporting communication between cars and the surrounding infrastructure. Congestion decreases, which has a positive impact on both the environment and personal well-being. IoT solutions are a key building block to develop user-friendly sharing systems. Just think of the e-scooters that people can hire and unlock from an application on your their mobile phone.

Cities As last domain we discuss the smart cities. Many different applications have already been rolled out here. In these settings usually a lot of data is collected. Examples are sensors that measure air quality or traffic volume. Based on that data, decisions concerning the infrastructure and residents can be taken. For example, traffic data aid in the decision process to determine road infrastructure modification. Another example is the air quality data that helps to decide the location of new residential areas and parks.

1.1.2 The Software Development Life Cycle

When advanced software needs to be developed, companies rely on Software Development Life Cycle (SDLC) strategies to efficiently execute a project. Agile methods focus on early and continuous delivery of the software [69]. This is equally the case when IoT software is developed. Figure 1.1 shows a common agile SDLC for IoT systems consisting of five stages executed repeatedly in sprints.

Requirement Analysis In the *Requirement Analysis* phase, the requirements for the software are gathered and elicited. These requirements are split up between functional and non-functional requirements. Functional requirements define the features the software must support, i.e., what the software must do. Non-functional requirements define the quality of the software. Concerns like security and scalability are typically part of the non-functional requirements and complemented with

[6] https://www.versasense.com/

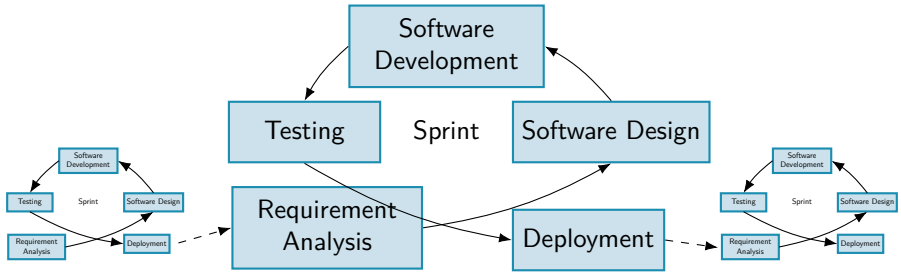


Figure 1.1: Visual representation of the agile SDLC

requirements to improve the user-experience and cost of developing and maintaining the system.

Software Design Based on the previously defined requirements, the software is designed in the *Software Design* phase. System components are determined, together with technologies that will be used. Major design decisions are documented, for example by visualizing the design using Unified Modeling Language (UML) diagrams, and can then be used as reference by the developers during development. Additionally, the Graphical User Interface (GUI) designs are determined by creating mock-ups, later used by front-end developers.

Software Development When the design of the system is determined, the *Software Development* phase can be initiated. Based on the determined design, the software is developed. This development can be split up between development teams, each with a different skill set. Back-end developers take care of the backbone structure of the software. Front-end developers ensure that end-users can work with the system in a user-friendly manner. To decrease development time, developers can rely on existing software tools and libraries.

Testing To verify that the software works as expected, the software must be tested extensively. If problems still arise, it is necessary to return to the *Software Development* phase after which the changes must be verified and tested again.

Deployment After the software has sufficiently been tested, it can be rolled out and used by the end users. If the software solution is part of an entire IoT deployment project, this occurs together with the roll out of the IoT infrastructure.

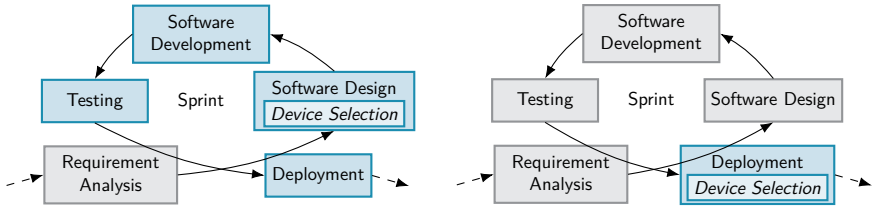
Maintenance After deployment, problems might still arise. IoT devices can be broken or no longer receive software updates, requiring a new type of device to be

added to the software. Bugs that have been overlooked during the *Testing* phase might be discovered. Or maybe a new feature must be added to the application.

1.1.3 Device Centric versus Application Centric Development

Currently, when realizing an IoT project, *device centric* approaches are dominantly applied. This implies that device selection occurs in an early stage of the development life cycle, usually the *Software Design* phase. First, device as well as communication technologies are selected. Subsequent design steps rely on those technologies.

Adjustments to the IoT infrastructure which result in new sensor types or technologies, imply that a large part of the work done during the SDLC needs to be repeated. This increases both development time and cost. Figure 1.2a shows that the SDLC proceeded from the *Software Design* phase. If the new sensor uses different communication technologies, it is necessary to restructure a substantial part of the software. These modifications must be adapted in the software in the *Software Development* phase and changes must be extensively tested. Only after these steps, the new sensor can be deployed and used in the system.



(a) Device centric SDLC (b) Application centric SDLC

Figure 1.2: Sensor selection and new sensor integration in the agile SDLC

In contrast to *device centric* design, in *application centric* design, device selection can happen at a later stage, namely during *Deployment*. *Software Design* and *Software Development* are largely device and technology agnostic. This results in applications that are highly maintainable and withstand the dynamic nature of IoT environments.

When applying the *application centric* approach, the development cycle can be reduced drastically, as integrating a new device only requires changes in the *Deployment* phase (see Figure 1.2b). Note, however, that a one time coding effort might be required to support a new device. However, this will not impact the design of the software application itself, as it is device technology agnostic. Thus, designing software with respect to the dynamic nature of IoT systems in mind has a positive impact on the further maintainability and reconfigurability of such systems.

1.2 Research Questions and Contributions

The work conducted during this PhD focusses on the following question.

Main Research Question: How can we support software companies in developing maintainable IoT applications for dynamic IoT environments.

This main question can be split in multiple questions.

Research Question 1 What sensor/actuator abstractions are appropriate towards application developers? What are the functionalities and tasks that middleware must provide to support these abstractions?

Research Question 2 Can application programmers reason in terms of assets (i.e., objects at application level) instead of sensors? Up to what extent can an architecture hide the underlying sensor complexity towards application developers?

Research Question 3 Can appropriate architectural decisions lead to increased flexibility with respect to sensor selection and replacements, and hence, contribute to vendor lock-in avoidance?

Research Question 4 Can architectural tactics also support the realization of other non-functional concerns like adaptivity, reconfigurability and security?

The following contributions were made during the PhD research, based on the predefined research questions.

Contribution 1 - Application Centric Architecture An architecture for application centric IoT development enables application developers to build complex, though maintainable, IoT ecosystems. The proposed design principles allow developers to focus on the business logic by abstracting low level IoT protocols, and communication and security mechanisms. The thesis also shows the steps that are required to couple IoT devices to applications for different classes of IoT devices with the architectural principles provided by the SMIoT architecture.

Contribution 2 - Ecosystem Modeling Guidelines Ecosystem modeling guidelines to build and maintain scalable yet reconfigurable IoT ecosystems by applying three key tactics. First, clear *separation-of-duties* between app designers and IoT infrastructure managers improves manageability. Second, *loose coupling between business logic and IoT infrastructure* advances reconfigurability. Third, we propose *late and selective binding* of sensors and actuators to applications in order to achieve favourable scalability, security and privacy properties. Technology-agnostic application policy definition is a central building block in our approach. The app

behaviour is expressed in terms of asset methods and states, and is subsequently mapped to operations on IoT infrastructural elements. This alleviates the design of new apps within the same IoT ecosystem, the redefinition of behaviour of already existing apps and modifications in the underlying infrastructure. Note that our work mainly focuses on cost-efficient integration of commercial-of-the-shelf (COTS) devices, rather than building dedicated endpoint hardware. We demonstrate the impact of our approach throughout the whole life cycle of an IoT ecosystem, and apply the proposed tactics to the development and operations of an Ambient Assisted Living (AAL) environment.

Contribution 3 - Reasoning Middleware A reusable **IoT middleware** that supports application developers in building dynamic IoT applications. The middleware can be used in different locations of the IoT ecosystem such as client applications, gateway applications, and even inside IoT devices. It comprises an event-based architecture running a logic reasoner in the background. It hosts a number of IoT modules specially devised to handle different functionalities, such as the handling of contextual changes, managing and enforcing access control and supporting a full featured automation engine. This makes it possible for the applications to handle the dynamic nature of IoT ecosystems end respond to external stimuli.

The logic reasoner is implemented using Prolog [81] and is the basis to add more advanced intelligence to the IoT applications. It allows applications to automatically infer knowledge and provides querying capabilities to gain insights in the IoT system. Tasks such as root cause analysis can thus be performed based on real-time data extracted from the system. Retrieving information from the systems state, such as a list of all active devices in a certain room is trivial and requires no additional programming.

To demonstrate its feasibility, the middleware has been integrated in a server and mobile application. For the demonstrator, the middleware was built on top of a JavaScript port of the SMIoT framework (Contribution 1 - Application Centric Architecture) providing generic access to IoT devices. Nevertheless, other frameworks could be used as well.

Contribution 4 - Access Control in Logic Programming A solution that evaluates access control policies during resolution in logic programs, taking special care for impure predicates. It provides a high expressiveness and fine-grained control of the program and makes it a widely applicable approach. A *deny as soon as possible* strategy is used, but decisions are postponed until they can be decided with certainty. Moreover, as enforcement occurs during inference, the approach easily extends to the dynamic case such as a reactive system (i.e., one that responds to external inputs). In this approach, access rules are defined as part of the program

logic. In other words, the rules can take advantage of the program's knowledge base. Hence, expressing access control strategies, such as resource based, role based and relationship based access control, is straightforward.

To validate and demonstrate the approach, an implementation is provided as a Prolog meta-interpreter, named ACoP. It can easily be integrated in existing Prolog programs with minimal effort. Overhead is limited to defining the access rules, also in Prolog.

1.3 Projects and Publications

The work proposed in this dissertation has been developed within the scope of multiple research projects and conducted with industrial stakeholders as well as actors from the non-profit sector. The continuous interplay between the research activities and valorisation increases the feasibility of the research results. The research has resulted in several publications, which can be found at the end of this dissertation.

SMIoT - Smart and Mobile IoT environments

A two year Vlaio TETRA project, started in January 2017.

The SMIoT project investigates how a diverse set of IoT devices, both custom-made and COTS, can be easily integrated into an advanced IoT ecosystem. By minimizing the overhead for developers and facilitating maintainability the solution is applicable for a diverse set of large environments such as ambient assisted living, fleet management and smart manufacturing.

SPITE - Security and Privacy in an IoT environment

A four year FWO SBO project, started in March 2017.

The SPITE-project aims at finding appropriate solutions for realizing advanced access control in an IoT setting. The new mechanisms will allow for increased flexibility, adaptability and security by making use of IoT devices to determine the user's context in a trustworthy manner, and based on this context adapting the required authentication procedure and the prevailing authorization rules. Moreover, some of the solutions can be applied to protect a subset of the IoT devices managed by the user.

Development Project with APEX

A two year Vlaio O&O project, started in January 2019.

This project was in collaboration with APEX, a Belgian based company and manufacturer of professional audio and audiovisual equipment. The goal of the

project is to integrate the existing APEX products with external IoT products. The end user should be able to configure new scenarios and atmospheres, and integrate new IoT devices easily. It must be taken into account that the developed solution must later be used in other domains such as smart home and ambient assisted living.

1.4 Overview of the Chapters

This section outlines the scope of the remaining chapters of this manuscript.

Chapter 2: Background Basic concepts and terminologies used in the remainder of the manuscript are listed and explained in detail. The different types of IoT architectures are first discussed, followed by commonly used technologies. The chapter ends by introducing an AAL use case that is used in the remaining chapters as working example.

Chapter 3: The SMIoT Architecture and Framework Integrating a diverse set of IoT devices in a single environment is no mean feat. This chapter examines how physical IoT devices can be integrated by means of virtualization. Not only devices but also environmental concepts are virtualized, in this way virtualization is taken one step further than in other related research. Actions no longer need to be performed at the device level but can be performed at a higher asset level. As a consequence of this virtualization, maintainability is increased. An architecture is proposed to support developers in setting up IoT environments, focusing on large system integrator companies. An Android framework is developed to validate the architecture and used to develop several applications in the AAL use case. *This maps to contribution 1.*

Chapter 4: Designing IoT Ecosystem Environments When integrating IoT devices in an environment, modeling the environment in a structural way prior to the development and deployment phase can significantly reduce the total costs. In this chapter the steps that are required during this modeling phase are defined. Three tactics are kept in mind while composing the method, namely separation of duties, loose coupling of the business logic and IoT infrastructure and finally an application centric development where physical devices are only selected in a late stage. *This maps to contribution 2.*

Chapter 5: Creating Advanced IoT Applications This last technical chapter addresses the automation of IoT ecosystems. A middleware containing a logic reasoning engine is proposed to automate several common tasks in IoT systems, among which are business logic automation, data preprocessing, connection management and access control. The reasoner is developed in Prolog and communicates with the underlying IoT middleware. By using Prolog its querying and inference capabilities are automatically inherited by the middleware. Support with the architecture proposed in Chapter 3 and modeling strategy as proposed in Chapter 4 is integrated by providing a conversion module that automatically binds the IoT device data to the correct environmental item (e.g., room or person) and vice versa. *This maps to contribution 3 and contribution 4.*

Chapter 6: Supporting Software Integrators in Building IoT Applications Valorisation possibilities for the concepts proposed in Chapter 3 through Chapter 5 are elaborated. Both guidelines and software tools are proposed to support application centric development in the Internet of Things. The focus lies on software integrators, integrating COTS devices in various IoT applications in multiple domains.

Chapter 7: Conclusion This last chapter concludes the conducted work. The contributions are summarized and reflected upon the predefined research questions. Future research and valorisation possibilities are elaborated.

Chapter 2

Background

The Internet of Things (IoT) landscape is continuously evolving. Due to the emergence of new technologies, other insights are acquired, which means that existing solutions can be adapted or improved. This leads to a very fragmented landscape with different ecosystem approaches, many different communication technologies and even more different device technologies. Due to this large diversity, applications can be developed in many domains, ranging from home automation, to the more complex Industrial Internet of Things (IIoT). The different requirements can be accommodated by the correct choice of approaches, communication technologies and devices.

In this background chapter an overview of the current IoT landscape is given. We take a look at the possible infrastructural approaches and existing communication and device technologies. Subsequently we enumerate a non-exhaustive yet representative list of existing solutions to build IoT applications. We finish this chapter with the elaboration of an Ambient Assisted Living (AAL) use case which is used as running example in this dissertation.

2.1 IoT Infrastructure Approaches

This section classifies IoT ecosystem approaches according to three categories. Note that we do not focus on how different end devices are connected to each other. A first category are cloud-based systems. In this approach, IoT devices are connected directly to the cloud. The devices can be controlled and the data can further be processed at remote servers typically installed in large server rooms. The second class are edge-based systems. Data is processed locally, either on the IoT device or on a local dedicated server. In between are the hybrid systems, a mixture between edge and cloud based systems. The possibilities to combine cloud and edge approaches are endless. For each use case, a different approach can be devised. Since each setup has its own advantages, it must be determined which setup is

most suitable based on the requirements of the use case under study.

Cloud Based Systems Figure 2.1 presents a basic setup of a cloud based system. IoT devices are connected to the cloud, possibly via a gateway. Devices are controlled via the cloud, and data is collected and processed in the cloud.

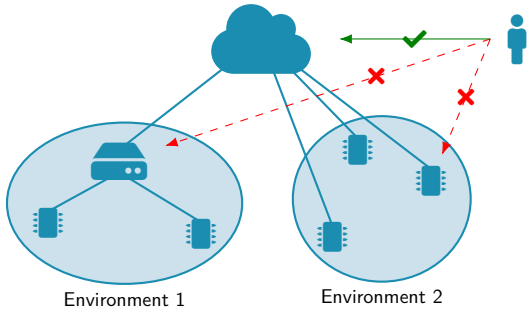


Figure 2.1: Example setup for a cloud based IoT system

A major advantage of cloud-based systems is the central point where data is made available. Data can be accessed at any time, from anywhere and combining data from different ecosystems is straightforward. However, direct access to the IoT devices, is not possible.

Another advantage is simple deployment. There is no need to manage and maintain proprietary infrastructure as multiple big tech companies provide easy-to-setup and maintainable cloud solutions. Examples are AWS, Azure, IBM Cloud and Google Cloud Platform. These providers often have IoT middleware solutions to support application development. Moreover, the middleware enables smooth software integration and deployment. Examples of such middleware cloud solutions are AWS IoT, Azure IoT Hub, IBM Watson IoT and Google Cloud IoT [58]. Such middleware solutions are one of the possibilities to enable interoperability between devices. In Section 2.3.2 we take a closer look at IoT middleware and list their advantages and current shortcomings.

Scaling is also no longer a concern, as the cloud provider foresees tactics for straightforward upscaling. Advanced processing power can be disposed to execute heavy processing tasks, as is typically required by machine learning algoritms. The latter is often not available on small IoT devices, or requires expensive hardware.

Applications for which cloud-based solutions are ideal are applications collecting and analyzing data from different, widely distributed devices. One can think, for example, of smart city applications, where data such as air quality or noise are monitored. Healthcare applications in which medical data from patients can be shared among

various stakeholders such as the patient himself, the general practitioner and the hospital can also rely on cloud-based architectures.

However, cloud based applications also expose major drawbacks. Due to the possibly limited bandwidth and delays, it is not feasible to rely on cloud-based solutions for time-critical applications [45]. Reliability with respect to network stability can not be guaranteed with cloud-based solutions either[18]. Controlling actuators remotely in the cloud requires continuous connectivity between cloud servers and edge infrastructure. While direct connections are possible, it is by far no sinecure. Publish-subscribe technologies (e.g. Message Queueing Telemetry Transport (MQTT)) are a frequently applied alternative, but this strategy is not recommended for a large number of time-critical applications.

Edge Based Systems In edge based systems, data are stored and processed locally, either on the IoT device itself or on other systems in the physical proximity. Figure 2.2 gives an example of two edge based systems. Note that the disadvantages of cloud based systems are at the same time advantages of edge based systems. Accessing data remotely and combining data from different ecosystems is hard. On the other hand, edge based setups are favorable from a privacy and security perspective. Data is only kept locally and not passed on to external parties, giving the owner full control over the data [118]. Management of the infrastructure, in this situation, is thus also the responsibility of the owner.

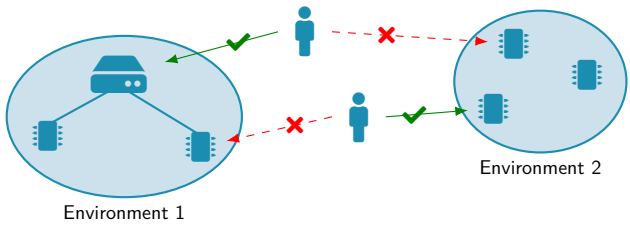


Figure 2.2: Example setup for an edge based IoT system

Hybrid Systems and Fog Computing As both cloud and edge based systems expose advantages and disadvantages, often, a combination thereof brings the best of both worlds. In hybrid setups, privacy sensitive data can be stored and processed locally. At the same time, processed data that must be easily accessible can be stored remotely.

While many IoT ecosystems rolled out today are hybrid ones, some specific cases require the data sharing possibilities of cloud setups and low latency of local setups. Fog computing, as firstly introduced by Cisco, extends cloud computing to the edge of the network (i.e., a fog is a cloud close to the ground). In time-critical

applications incorporating mobile nodes (e.g., connected vehicles), local setups do not suffice. As a node's location is dynamic and changes over time, a single local network is insufficient.

Figure 2.3 shows an example of a hybrid system. Environment 1 and 2 are connected to the same fog. Selected data of these two environments can thus be combined close to the edge.

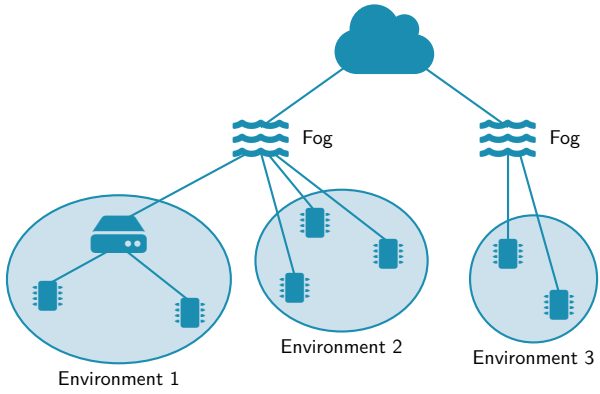


Figure 2.3: Example hybrid IoT system using fog computing

2.2 Communication in the IoT domain

Not only the architectural types of IoT systems are diverse. A wide variety of communication technologies, protocols and different data formats exist and their selection is based on requirements of the applications under study. For some applications high data rates are essential, while others rely on long ranges. Moreover, many IoT devices are battery powered. If so, energy consumption due to communication may be a determining factor. As for communication protocols, some applications require a publish-subscribe protocol to easily reach multiple clients. In other applications, a request-response protocol is sufficient or desirable. Lastly, the requirements regarding data format can also differ per application. A binary format may be sufficient to transmit sensor values, in other cases more structured data is preferred to transmit large amounts of data.

In this section we provide an overview of commonly used communication technologies, protocols and data formats in the IoT landscape. Every commercial IoT product on the market builds on a combination of the elements below. This shows once again how diverse the IoT domain is.

2.2.1 Wireless Communication Technologies

Although wired connections are possible for static devices, this section – and even so the major part of this thesis – focuses on wireless technologies. The advantage of wireless devices is their mobility. They are easier and often cheaper to enroll in an existing network as no additional cabling needs to be provided.

The range that can be achieved, the maximum data rate and the energy consumption affect each other. Depending on the type of application, an optimal trade-off between these three characteristics must be found. Based on the range of a network, a distinction can be made between Personal Area Networks (PANs), Local Area Networks (LANs) and Wide Area Networks (WANs) [23]. PANs range up to 10 meters. A good example of a PAN is a prototypical office environment. A smartwatch communicates with a mobile phone and a wireless headset is connected to a local computer. If an individual moves away from the computer with the headset, (s)he will unfortunately no longer be able to enjoy the music.

Although smaller ranges typically result in higher data rates, the data rate requirements in PANs are often, apart from some exceptions, limited. Body sensor data or text messages are often constraint in both size and speed. An exception, however, is the data stream when listening to music through a wireless headset. LANs can range up to 5 km and usually have a higher data rate. LANs can be set up in buildings such as homes, schools offices and factories. A home network is also a LAN, connected to the internet. Individuals want a high data rate to stream their favorite series to a laptop. Lastly, WANs cover areas up to 50 km. A prototypical WAN example is a mobile phone network. In densely populated areas, cell towers are available within the proximity of 1 km which is required to have coverage. In open areas, coverage on a mobile phone is still possible when the nearest cell tower is up to 40 km away. Further developments can result in WANs data rates that are as high as in LANs, but this is often at the expense of energy consumption.

Several comparative studies can be found in the literature [23, 70, 91, 97]. This section will further give a high level overview of the most commonly used technologies, without going into the technical details.

We begin with the technology with the shortest range and lowest data rate, which is used in PANs, and gradually shift to the technologies that have a higher data rate. We thus come to frequently used LAN technologies and end with technologies for WAN applications.

Radio-Frequency Identification (RFID) and Near-Field Communication (NFC) RFID technology is a very short range communication technology in which a tag and reader/writer have to be placed in close proximity of each other in order to exchange data. The reader can read/write the data on the tag. Hence,

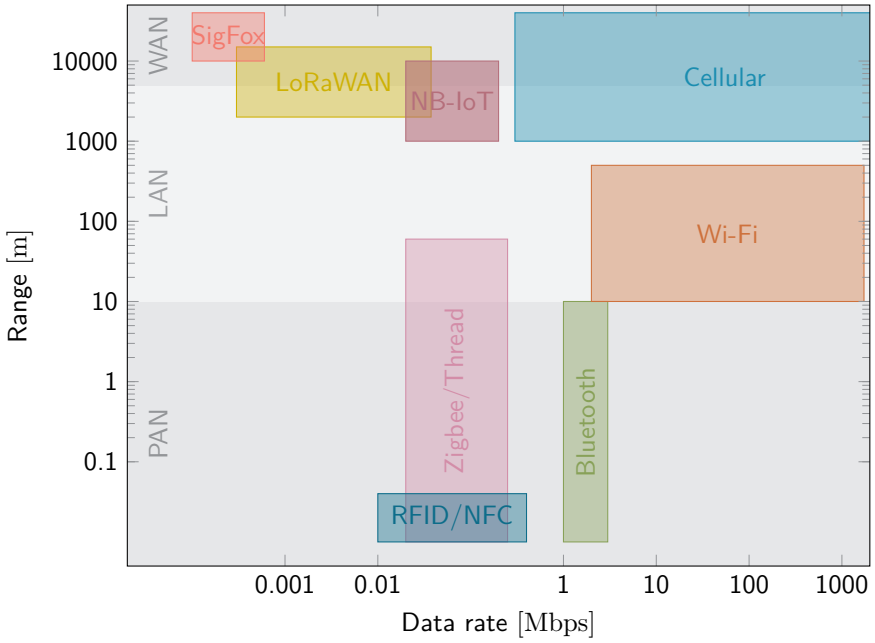


Figure 2.4: Range and data rate for various wireless technologies, based on the work of Cheruvu et. al. [23]

RFID communication is a one-way communication strategy. RFID technologies are widely used in retail and in supply change applications for tracking and inventory purposes. NFC is a subset of RFID. Data can be exchanged through nearby contact between 2 devices. The difference with RFID is that an NFC device can act as both a reader and a tag. NFC tags are integrated in cards, phones and other wearable devices (e.g., watches and rings). NFC is widely used for contactless payments and access control in PANs.

Bluetooth, Bluetooth Low Energy (BLE) and BLE Mesh Bluetooth is a well-known wireless communication technology. It was first introduced by Ericsson in 1994 and was initially introduced to transmit data wireless between computing devices like printers and headsets. It is currently still widely used. For instance, many headphones are connected to computers or smartphones over Bluetooth today. [86, 119]

Since October 2010, BLE, also known as Bluetooth Smart, has been introduced. A new mode, in addition to Bluetooth Classic, is optimized to decrease power consumption. Also the throughput is increased from 1 to 2 mbps. A star

network topology where multiple nodes can connect to the same device is another improvement over the classic Bluetooth connections [91]. Starting from version 4.0, one can choose to either use Bluetooth Classic or the low energy mode. BLE is currently not optimized for audio streaming. In these cases, Bluetooth Classic offers the solution to support audio streaming between devices [117].

More recently, in 2017, BLE Mesh was proposed, Bluetooth is no longer limited to a star topology, but now also supports mesh networks. Nodes can be connected directly. It means that nodes no longer need to be connected to the central point. The distance between a node and the central point can thus be elevated, provided that other devices can mediate communication. All Bluetooth devices supporting the low energy mode are also able to use of the mesh capabilities [117].

ZigBee ZigBee is a widely used communication technology for PANs and LANs. Zigbee is based on IEEE 802.15.4 and adds two extra layers of security on top. Zigbee uses a mesh network of devices to ensure reliability. In the event of a device failure, the network heals itself and the connection with other devices is automatically restored. Mesh networks can also cover larger areas as long as the distance between two nodes is acceptable. For Zigbee, the maximum distance between two nodes is between 75 and 100 meters depending on the strength of the power source and the environmental infrastructure. Currently this technology is widely used in the smart home domain and especially in lightning applications.

Thread Thread is an emerging IPv6 based communication technology and designed with smart home applications in mind. A data rate up to 250 kbps can be achieved with a range up to 30 meters. Using a mesh network, all devices in a home can be covered. [94] The advantage of Thread over Zigbee is that Thread is an IP based protocol, so integration into other IP networks is easy. For larger packets, Thread has a lower latency than Zigbee [6]. Additional border routers can be set up to cover an even wider area. Thread networks are "self-healing", on device failures or when a device becomes available, the network will configure itself [49, 107, 108].

Wi-Fi Wi-Fi is a communication technology commonly used in grid powered IoT applications. It enables devices to wirelessly connect devices to a LAN. Devices connected to the same local access point can communicate with each other. The access point can make wired connections to other local networks and the internet. This makes it possible to address the IoT devices from outside the local network. Wi-Fi supports both a 2.4 Ghz and 5 GhZ frequency. A higher frequency results in a higher data rate, while a lower frequency results in a lower data rate but a longer range up to 100 meters.

LoRaWAN LoRaWAN, is a low power communication technology for WANs, also called a Low Power Wide Area Network (LPWAN) technology. It is a bidirectional communication technology to send small data packets up to 256 bytes. Gateways are used to connect devices. Due to being low power, it is suited for battery-powered sensors.

Sigfox Sigfox is a long range communication technology up to 50 km. It is low power but can only send small data packages up to 12 bytes. A star topology is used. Each device must connect to a Sigfox station in order to send data. That data is sent to the Sigfox cloud, where it is made available to the users. Although Sigfox has a bidirectional functionality, it is usually used to send data from an end point to a base station. The advantage of Sigfox, as with cellular technologies, is that it can send data whereever there is coverage. The current coverage state, which can be found on their website[1], shows that Europe is almost entirely covered.

Narrow-Band IoT (NB-IoT) NB-IoT is a cellular type of IoT communication technology. Key features are its long range, however limited to 1km in urban areas and 10km in rural areas, the low energy consumption and high reliability. NB-IoT has a data rate downlink of 200kbps and the uplink is limited to 20 kbps. The maximum payload size for a message is 1600 bytes. NB-IoT is less suited for time critical applications as latency can be up to 10 seconds. [32, 73]

Cellular 3G/4G/5G LTE Probably the best known long range technology is the cellular communication technology. Devices connect to cellular radio towers in order to communicate with each other. Its bidirectional connection enables to send and receive data simultaneously. Depending on the density of the surrounding environment, the technology has a range of 1 km in cities up to 40 km in open areas. However, this technology has a high power consumption and is less appropriate for battery powered devices whose battery can not be replaced or charged easily. Of all the discussed long range communication technologies, the cellular technology, is most suited for time critical application, however, there can still be a latency up to 1 second.

2.2.2 The IoT Stack

Regardless of the communication technology used by IoT devices, the data of each device is often made available in some way via the internet or in a local network. This is either done via a gateway, or a direct connection to the internet or local network. Figure 2.5 depicts the IoT protocol stack, which is very similar to the web protocol stack. [77] We will not dive deeper into the network, internet, and

[1] https://www.sigfox.com/en/coverage

transport layer. However, we will take a look at the application layer and the data formats used in the IoT stack.

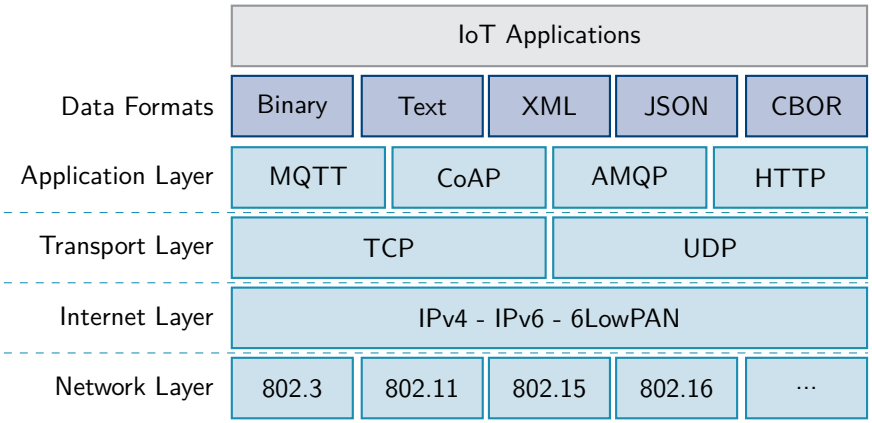


Figure 2.5: Representation of the IoT stack

2.2.2.1 IP Messaging Protocols

Internet Protocol (IP) messaging protocols can be divided into two categories. On the one hand, request/response protocols start with a request from a client (i.e. application) to a server (i.e. IoT device). The response is sent back from the server to this client. On the other hand, publish/subscribe protocols handle the published messages by peers. These messages are forwarded by a broker to all interested (i.e. subscribed) peers. The advantage of publish/subscribe over request/response protocols is that a message coming from a single device can be sent immediately to multiple applications. Publish/subscribe mechanisms also generally have better latency and better network utilization. The downside, however, is that the central broker is a single point of failure. If the broker is no longer available, not a single device publishing through that broker will be available. [48] Five commonly used messaging protocols, supporting either publish/subscribe, request/response or both, in the IoT stack are Hypertext Transfer Protocol (HTTP), Message Queueing Telemetry Transport (MQTT), Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP) and Extensible Messaging and Presence Protocol (XMPP). These protocols will briefly be explained below.

HTTP [43, 77, 115] HTTP is a web messaging request/response protocol. It supports the Representational State Transfer (REST) architecture and Uniform Resource Identifiers (URIs) are used to identify the endpoints. HTTP does not guarantee Quality of Service (QoS) and introduces large overhead. HTTP runs on top of Transmission Control Protocol (TCP) which provides reliability and can

transport large amounts of data. In resource constrainted environments, this can, however, be a disadvantage.

MQTT [7, 77] MQTT is a lightweight publish/subscribe protocol. Messages are pushed to the broker on a so called 'topic'. Clients can subscribe to multiple topics. MQTT offers three levels of QoS. The lowest level sends each message at most once, possibly resulting in data loss. The second level sends each message at least once, it is however possible that messages are send and received multiple times by the receiver. The highest level, sends each message exactly once, giving the best and most reliable QoS. As does HTTP, MQTT runs on top of TCP.

CoAP [33, 104] MQTT is another lightweight protocol that supports both the request/response and publish/subscribe topology. As in HTTP, URIs are used to identify endpoints and is also based on the REST architecture. Publish/subscribe support is added in a later version of CoAP by tagging GET-requests with an observe option. In contrast to HTTP and MQTT, CoAP runs on top of User Datagram Protocol (UDP) which is unreliable. However, QoS in CoAP is supported by its own reliability mechanism using *comfirmable* and *non-comfirmable* messages. The former must be acknowledged by the receiver, the latter do not need to be acknowledged.

AMQP [33, 76, 95] Both the request/response and publish/subscribe mechanism are supported by AMQP. It runs on TCP and provides the same three levels of QoS as MQTT. In older version, an AMQP broker consists of an *exchange queue* and a *message queue*. The *exchange queue* obtains publisher messages and redirects them to the correct message queue, which represents a topic. Subscribers listen to these queues for new messages. Newer versions of AMQP follow a peer-to-peer mechanism. There is no longer need of a broker in case messages can be directly send to other peers, increasing flexibility for different setups. However, a broker is needed in case messages must be stored until a peer becomes available. Due to its multiple features, AMQP has high power and memory requirements. Thus, in contrast to MQTT and CoAP, AMQP is more suited for settings without bandwidth and latency restrictions.

XMPP [33, 95] XMPP is a text-based protocol, based on eXtensible Markup Language (XML) and also supports both the request/response and publish/subscribe mechanism. XMPP runs on top of TCP and already incorporates Transport Layer Security (TLS) mechanisms in its specification. Thus, of all presented protocols, XMPP is the most secure. However, due to using XML messages sizes are rather large, making XMPP less suited for networks with bandwith constraints.

2.2.2.2 Data Formats

Not only the communication technologies and messaging protocols differ from IoT product to IoT product, the way in which data is structured is also different for each product. Plain data formats (i.e. binary and text) are mainly used when the maximum data rate is limited, the advantage of text format is that it is human readable, it can be quickly interpreted and errors can be discovered more easily compared to binary data . Structured formats (i.e. XML, JavaScript Object Notation (JSON) and Concise Binary Object Representation (CBOR)) usually contain not only the data sensed by the device or to actuate the device, but also metadata about the device (e.g. device id, connection state and battery level). Structured data formats that are frequently used within IoT products are briefly explained below. Although the described data formats are structured, none of these data formats define a strict syntax. Two IoT products that use the same data format can therefore not necessarily communicate with each other, because the data can still be structured in a different way (e.g. one device uses the key *parameter*, while another uses the key *property*).

eXtensible Markup Language (XML) [67, 72] XML is a data format adopted from the web stack. It is a human readable text based format. Listing 2.1 gives an example of XML structured data from an IoT temperature and humidity sensor. The disadvantage of XML is the overhead created by header information and the opening and closing tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <data>
      <element>
         <parameter>temperature</parameter>
         <unit>°C</unit>
         <value>26.4</value>
      </element>
      <element>
         <parameter>humidity</parameter>
         <unit>%</unit>
         <value>43.6</value>
      </element>
   </data>
   <id>9f2bd4dd-0d9c-41f9-811c-26e940ec226b</id>
   <timeStamp>2022-07-07T16:39:57</timeStamp>
</root>
```

Listing 2.1: Example of XML structured IoT data

JavaScript Object Notation (JSON) [19, 63, 67] Just like XML, JSON is a human readable data format and widely used for the web stack. It is a lightweight

format consisting of key-value pairs. Unlike XML, JSON must not be decoded. Many programming languages come with built-in libraries to easily retrieve JSON data, making it an easy to integrate data format. Although JSON is lightweight, the text representation still causes an increase in message size. Listing 2.2 shows the same temperature and humidity sensor data as before in JSON format.

```
{ "id": "9f2bd4dd-0d9c-41f9-811c-26e940ec226b",
  "timeStamp": "2022-07-07T16:39:57",
  "data": [
   { "parameter": "temperature",
     "unit": "°C",
     "value": 26.4},
   { "parameter": "humidity",
     "unit": "%",
     "value": 43.6}
  ]
}
```

Listing 2.2: Example of JSON structured IoT data

Concise Binary Object Representation (CBOR) [26, 63] In contrast to XML and JSON, CBOR is a binary based format which optimizes message size. It is thus not human readable, which cearly can be seen in Listing 2.3. CBOR is based on JSON and one of the major motivators to develop CBOR was the need for small messages in the IoT.

2.3 Existing Solutions for Device Interoperability

Communication technologies and varying device types are often combined in complex IoT ecosystems. The strengths and constraints of these solutions are evaluated in many papers [37, 40, 66, 89]. However, integrating multiple technologies in a single application requires a substantial programming effort.

Two types of solutions exist to increase the interoperability of IoT devices. One strategy aims at complying with standards. During hardware design and development, the manufacturer fixes the way in which the devices interact. A seconds strategy consists of middleware solutions that aim at increasing interoperability without adapting the hardware.

2.3.1 Standardization Efforts

In order for all IoT devices to be able to communicate with each other, all devices should communicate in the same way. Thus all messages must be constructed in the same way. By defining standards, a uniform communication can be established.

```
//raw
0xA3626964782439663262643464642D306439632D343166392D383131632D323665393430656
↪ 3323236626974696D655374616D7073323032322D30372D30375431363A33393A35376464
↪ 61746182A369706172616D657465726B74656D706572617475726564756E697463C2B0436
↪ 576616C7565FB403A666666666666A369706172616D657465726868756D69646974796475
↪ 6E697461256576616C7565FB4045CCCCCCCCCCCD
//split into meaningful chunks
A3                                  // map(3)
   62                               // text(2)
      6964                          // "id"
   78 24                            // text(36)
      3966326264/*...*/6332323662   // "9f2bd4dd-0d9c-41f9-811c-26e940ec226b"
   69                               // text(9)
      74696D655374616D70            // "timeStamp"
   73                               // text(19)
      323032322D/*...*/33393A3537   // "2022-07-07T16:39:57"
   64                               // text(4)
      64617461                      // "data"
   82                               // array(2)
      A3                            // map(3)
         69                         // text(9)
            706172616D65746572      // "parameter"
         6B                         // text(11)
            74656D7065726174757265  // "temperature"
         64                         // text(4)
            756E6974                // "unit"
         63                         // text(3)
            C2B043                  // "°C"
         65                         // text(5)
            76616C7565              // "value"
         FB 403A666666666666        // primitive(4628124157067290214)
      A3                            // map(3)
         69                         // text(9)
            706172616D65746572      // "parameter"
         68                         // text(8)
            68756D6964697479        // "humidity"
         64                         // text(4)
            756E6974                // "unit"
         61                         // text(1)
            25                      // "%"
         65                         // text(5)
            76616C7565              // "value"
         FB 4045CCCCCCCCCCCD        // primitive(4631332971801791693)
```

Listing 2.3: Example of CBOR structured IoT data

An example is oneM2M [102], which aims at establishing a standardized Machine to Machine (M2M) service layer platform for globally applicable and access-independent M2M services (i.e., horizontal standardization). OneM2M is a widely used standard since 2012 that aims to allow IoT devices to communicate across different domains.

This facilitates setting up cooperative systems in which infrastructures from different organizations work together (e.g., in smart cities).

Other standardization efforts focus on specific application domains (i.e., vertical standardization) such as smart homes [110, 121], smart cities [27] and the IIoT [75]. Although these standardization efforts target interoperability between IoT components from different stakeholders, the IoT market is still very fragmented. It is almost impossible to make the already highly fragmented market comply to one single standard. Hence, the flexibility of IoT ecosystem providers is significantly decreased if they are restricted to IoT components adhering to a specific standard.

Another emerging standard is Matter[2]. Matter, previously known as Project Connected Home over IP (CHIP), is a standardization effort in the smart home domain and incorporates BLE, Zigbee, Thread and Wi-Fi, [110]. Several manufacturers already announced that Matter will be integrated in their upcoming devices including Amazon, Apple, Google, Samsung and Signify (the company behind Philips hue) [1, 4, 25, 88, 101]. An advantage of Matter is that – as it is built on top of Wi-Fi 6 –, the end devices do not need mesh capabilities. As Wi-Fi 6 supports this, devices using different technologies can connect in the same network when using Matter [110].

The downside of these standardization efforts is that there are many of them. It is difficult for a hardware developer to guess the most sustainable standard. Complying to multiple standards, even a small subset, considerably increases development cost and can even impact the device characteristics.

2.3.2 Middleware Solutions

An alternative approach for managing the heterogeneity in IoT components and technologies is relying on middleware solutions. IoT middleware is an intermediary software system or layer between IoT devices and the applications [78].

By providing a middleware between the infrastructure and applications, interoperability of different devices and technologies can be achieved.

Many solutions are cloud based [8, 109], commercial examples are Siemens MindSphere[96], GE Predix[80], Zetta[120], ThingSpeak[55] and DeviceHive[31]. Many proposals have already been made in academia too. For instance, Lea et al. [61] and Demirkan et al. [29] use a cloud-based hub for developing respectively smart city and healthcare applications.

Others run on a dedicated personal server, for instance Home Assistant[50], OpenHAB[79], Macchina.io[64] and Blynk[9]. A local gateway setup is also used by

[2] https://csa-iot.org/all-solutions/matter/

Yang et al. in their MicroPnP [116] platform. It is a generic zero-configuration, plug-and-play wireless sensor platform consisting of a gateway that interacts with nodes on which sensors/actuators can be added without requiring any additional configuration. Access to the sensors is provided via a REST interface on the gateway.

A hybrid setup with a combination of a local gateway and cloud hub is used by Soliman et al. [99] and Desai et al. [30] in the smart home application domain.

Middleware solutions are also adopted by many commercial organizations. For instance, Google[3], Apple[4] and Samsung[5] provide their own smart home IoT platform. Each platform defines Application Programming Interfaces (APIs) that can be supported by third-party IoT device developers to enable interoperability. Using a cloud or gateway platform to bootstrap access to IoT devices significantly simplifies management and application development. However, typically multiple applications can be developed in the context of an IoT ecosystems. Each of these applications can have different requirements concerning, among others, privacy, real-time constraints and access control. To fulfill advanced requirements, often a hybrid distributed setup that combines both direct sensor-application interactions as well as interactions mediated via gateways or a cloud platform are required [85].

The majority of these solutions collect the data centrally or provide one generic application to interact with the devices. Building dedicated applications on top of such middleware solutions is often not supported. In this thesis we propose a middleware solution that supports the development of a diverse set of applications. Moreover, the focus is not on a single domain. On the contrary, the middleware can be reconfigured based on domain and application requirements.

2.4 An Ambient Assisted Living Use Case

To show the applicability of the research results, a representative use case in the care home domain is used throughout the remainder of this thesis.

A scalable AAL ecosystem, consisting of numerous care home units and residents, is proposed. Each unit consists of multiple rooms (living room, bathroom, bedroom...) and each room is equipped with sensors (e.g., for sensing environmental parameters such as temperature, pressure and humidity) and actuators (e.g., lamps and thermostats). Moreover, residents can wear one or multiple body sensors. Examples are heart rate monitors and fall detection sensors. Figure 2.6 demonstrates the setup. The set-up of each individual care home unit may be different. A heterogeneous set of devices can be deployed and rooms do not necessarily contain the same set

[3] https://developers.nest.com
[4] https://www.apple.com/us/shop/accessories/all-accessories/homekit
[5] https://www.smartthings.com/

of sensors and actuators. Moreover, each home unit can rely on different types of sensors and/or actuators to achieve certain goals.

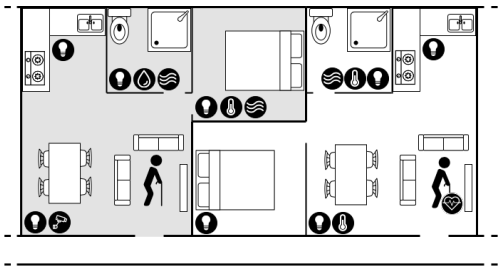


Figure 2.6: A visual representation of two care home units

Several types of end-users are active in the ecosystem: residents, caregivers and the maintenance crew. Each end-user has different requirements and applications are adapted to their needs. Hence, each type of end-user has a different view on the IoT ecosystem.

Resident Application The resident application provides access to the resident's care home unit, and allows to retrieve sensor values and control actuators in that unit. The resident's view on the environment is restricted to its own unit and does not change frequently. The view only changes when devices are added, removed or replaced.

Caregiver Application The caregiver needs to control the sensors and actuators of a subset of care home units, namely the ones that correspond to the residents that the caregiver needs to visit. The view on the environment, along with its configurations, can change according to the work schedule of the caregiver. Each time new residents are added to or removed from the schedule, the application configurations change.

Maintenance Application The maintenance application has a different view on the IoT environment. Crew members need access to particular devices based on repairs and checks that need to be done. Their connection with *Assets* is of minor importance.

2.5 Conclusion

This chapter gave a broad overview of the IoT landscape. The possible infrastructural approaches, each with their advantages and disadvantages, were discussed. Subsequently, the diversity of possible communication technologies shows that

the IoT landscape is very fragmented. In addition to the many possibilities for inter-device communication, IoT data is also made available on a local network or the internet. While discussing the application layer of the IoT stack, it once again becomes clear how diverse the different IoT products offer their data. A uniform use of communication technologies, protocols and data formats is first of all impossible to accomplish. There is no single combination of technologies that can meet all possible application requirements. Secondly, the use of the same technologies, protocols and data formats does not guarantee interoperability, since the data itself follows a manufacturer-specific syntax.

Possible solutions have already been offered to enable communication between devices and to develop integrated application. Standardization attempts are proposed to unify the communication between the devices. However, this has a major impact on hardware development and becomes very difficult to enforce in the current advanced stage of the IoT landscape. Middleware solutions, on the other hand, are a viable alternative to enable interoperability. However, existing proposals are very cloud-centric. Data is centralized to a single cloud environment where it is made available to applications.

In the following chapters, a middleware architecture is first presented that is not limited to cloud-based systems, but also focuses on edge based and hybrid systems. Next, guidelines for modeling ecosystems are proposed, in order to simplify integration of new devices and adaptations of applications. Finally, extensions are proposed to develop advanced applications in a simple way, taking into account the diverse nature of IoT environments.

Chapter 3

The SMIoT Architecture and Framework

The content from this chapter is previously published in:

- M. Willocx et al. "Developing Maintainable Application-Centric IoT Ecosystems". In: *2018 IEEE International Congress on Internet of Things (ICIoT)*. San Fransisco, CA, USA, July 2018, pp. 25–32
- I. Bohé et al. "An extensible approach for integrating health and activity wearables in mobile IoT apps". In: *2019 IEEE international congress on Internet of Things (ICIoT)*. IEEE. Milan, Italy, 2019, pp. 69–75
- M. Willocx et al. "QoS-by-Design in reconfigurable IoT ecosystems". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE. Limerick, Ireland, 2019, pp. 628–632
- I. Bohé et al. "SMIoT: a software architecture for maintainable internet-of-things applications". In: *International Journal of Cloud Computing* 9.1 (2020), pp. 75–94

As mentioned in Chapter 1, *device centric* development is the predominant paradigm when building Internet of Things (IoT) ecosystems. This implies that sensors and actuators are selected and anchored in a very early design stage. Thereafter, system integrators start to think about the design and realization of attractive software applications. However, the lifetime of advanced software applications often outreaches the lifetime of IoT devices due to their limited cost or the harsh conditions in which they are deployed. Unfortunately, many existing IoT integration

solutions offer no or very limited flexibility when devices need to be replaced, as devices are selected at the very beginning. For instance, a temperature sensor from one manufacturer is often not replaceable by another – more robust or cheaper – one from another manufacturer without major code changes. This occurs due to the lack of flexibility during system design. *Vendor lock-in* is thus often mentioned as one of the fundamental problems in current IoT deployments. In addition, implementation efforts are often expensive due to the lack of high-level sensor integration support towards application developers. Many IoT devices currently on the market only offer a low level Application Programming Interface (API), confronting application programmers with low-level connectivity and data representation problems. On its turn, this increases the development time.

This chapter proposes the SMIoT architecture that facilitates *application centric* design, and supports the development of complex and maintainable IoT applications. The architectural guidance allows for dynamic and reconfigurable IoT device integration, and hides low-level implementation details from application developers. Hence, the latter can focus on business logic without being expert in IoT device technologies.

The proposed architecture is especially useful for software companies focusing on extensive IoT solutions in a specific domain or sector. A typical example is a software integrator focusing on innovative healthcare environments, or a company building software ecosystems for fleet management. With extensive, we mean that the IoT ecosystems consists of various IoT applications used by different stakeholders in the domain. For instance, in a healthcare environment, applications are developed for caregivers, elderly people, family, doctors, nurses, governments and insurances. Each stakeholder has a partial view on the overall IoT ecosystem.

The software may evolve over time, and its lifetime is typically much longer than the lifetime of the device technologies that are plugged in. In addition, the development department in such software companies often consists of teams with complementary skills. Some focus on implementing business logic; others are experienced in supporting the right software abstractions for IoT device technologies. The layering offered by the SMIoT architecture offers the right approach for such development teams with mixed skills. This separation of concerns makes it possible to efficiently develop IoT applications

Replacing a device by a similar one from a different vendor might however still cause difficulties. For instance, a broken sensor monitoring the temperature of boxes during organ transport cannot be replaced by whatever other temperature sensor on the market. Quality of Service (QoS) properties restrict the feasible options. Hard constraints can be imposed on acceptable polling frequency and accuracy. Soft constraints can be imposed on battery life. Similarly, fall detection systems need to be reliable, and guarantee data transmission within a small time interval. Whereas

an application may depend on device specific properties only available in a certain set of devices, the proposed design tactics offer important advantages on how to cope with those hard and soft constraints and how to inform application developers of degradations.

Contributions Firslty, this chapter proposes a layered architecture based on the virtualization of physical elements, providing support for *application centric* IoT development. Secondly, QoS challenges during the development and operations cycle are tackled. Device QoS annotation is proposed during the design phase and QoS cataloging is proposed to steer the device selection process. Finally, the design principles derived from the architecture are applied in both an Android and JavaScript Framework, supporting developers to develop mobile and web applications for the IoT, respectively. Integrating different low level communication protocols is no longer the job of the application developer and he can focus on the business logic of the applications. Maintenance of the IoT environment is easier as a device replacement only results in a configuration change, while business logic remains unchanged.

The remainder of this chapter is structured as follows. Section 3.1 gives an overview of the SMIoT architecture. In Section 3.2, we take a look at how QoS support is added. Subsequently, a description of how IoT devices are bound to applications follows in Section 3.3. Framework support is described in Section 3.4 followed by a validation of the architectural concepts in two use cases (Section 3.5 and 3.6). Section 3.7 discusses the proposed architecture.

3.1 Architecture

This section gives an overview of the SMIoT architecture.

As befits a good software development project, this section begins with an analysis of the requirements. Subsequently, the design of the layered architecture is elaborated, together with the concepts at each layer.

3.1.1 Requirements

Intuitive interfaces. Developers want to rely on interfaces that make abstraction of details of the underlying infrastructure. In many cases, they even do not want to be confronted with any IoT sensors or actuators at all. This means that application developers want to invoke methods on assets (such as patients, rooms and cars) instead of sensors (such as fall detectors, light sources and Global Positioning System (GPS) sensors).

Separation of concerns. The architecture facilitates mixed development teams consisting of members with complementary skills. Some developers focus on IoT device integration while others provide domain-specific intuitive interfaces to application developers. Finally, application developers focus on realizing application logic and do not want to be confronted with the underlying IoT infrastructure such as the sensor model or vendor and communication technology.

Reconfigurability. IoT devices can be replaced by others – possibly provided by other sensor manufacturers – without affecting the application. Supporting cost-efficient sensor replacement is essential to tackle vendor lock-in and enables the use of more accurate or less expensive sensors over time, depending on the specific application needs. For example, replacing a Bluetooth temperature sensor by an alternative one that pushes its data directly to the cloud should be possible with limited implementation effort. The application logic remains unchanged.

Context-awareness. The optimal IoT configuration is loaded and initialized based on contextual parameters, and context may evolve over time. The application relies on sensor data (such as beacon technology and GPS data), user information and external data (e.g., the current date and time) to determine the current context. For instance, while a caregiver visits multiple patients, the application only needs to access and show the sensor devices of the patient he is currently visiting.

3.1.2 Design

The SMIoT architecture consists of four abstraction layers which are depicted in Figure 3.1.

Infrastructure Layer The *Infrastructure Layer* represents the hardware and software that senses and actuates the physical world, and that stores historical values. This layer consists of IoT devices, gateways and back-end IoT platforms. IoT devices are physical sensors and actuators. They interact with the application directly (e.g., via Bluetooth), are mediated by a local gateway, or push their data into an IoT platform. The gateway is either connected to the application over a local network or pushes the data to an IoT platform. The infrastructure layer controls a heterogeneous set of devices which may evolve over time. For instance, different types of temperature sensors, potentially relying on different communication technologies, can be combined in one and the same application.

Virtual Device Layer The *Virtual Device Layer* consists of one or more *Virtual IoT Devices*. Each *Virtual IoT Device* defines a sensor or actuator in the IoT ecosystem. A type and technology are tied to each *Virtual IoT Device*. Examples are a temperature sensors, lamps and locks. The technology defines the brand

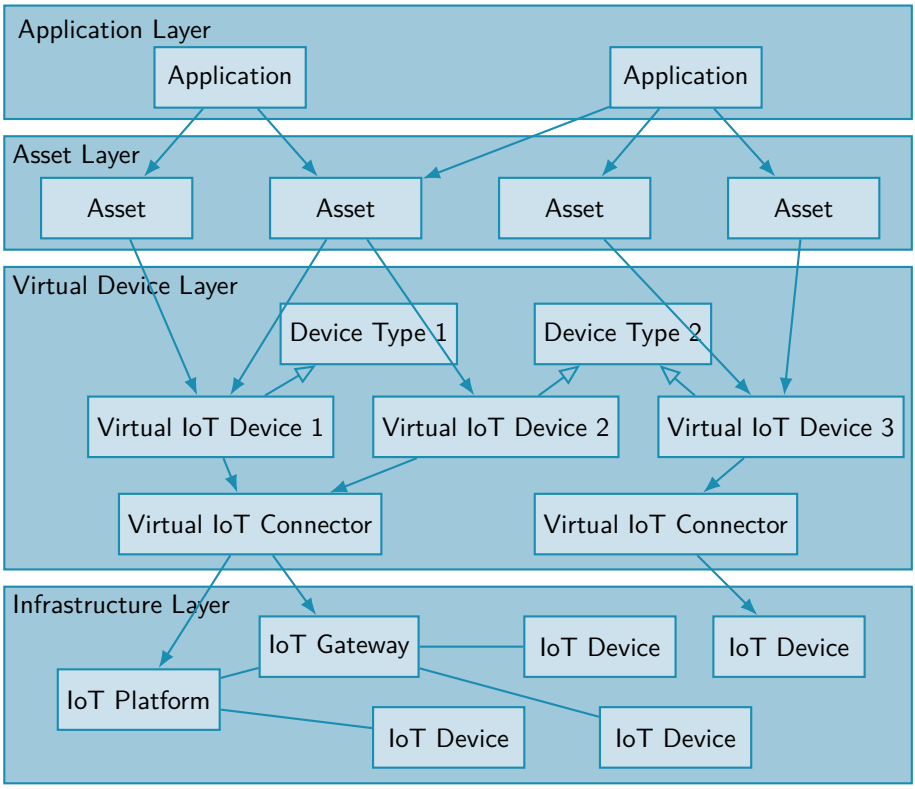


Figure 3.1: Software layers in the the SMIoT architecture

and model. Examples of lamp technologies are Philips Hue lightstrips/bulbs and Osram Lightify models. A uniform interface is assigned to each device type. *Virtual IoT Device* implementations present device functionalities to the upper layer in a uniform way. They also define a uniform representation of the device's attributes. For example, a temperature value can be retrieved in °C, °K, or °F. The *Virtual IoT Device* implementation transforms the values if necessary and passes a uniform representation to the upper layers. Application developers rely on those implementations to use the obtained data and invoke methods on IoT devices. If a device is replaced by another one, provided by a different manufacturer, a new *Virtual IoT Device* must be created. Multiple *Virtual IoT Devices* can point to the same infrastructural element. For instance, multiple sensors can be plugged on a single embedded device. Similarly, access to a set of sensors can be provided via a gateway or a cloud platform. To reduce code redundancy and resource usage

in the previous cases, *Virtual IoT Connectors* are defined. *Virtual IoT Connectors* handle communication between the application and each IoT device, gateway or IoT platform. These communication handles implement a part of the communication protocol (e.g., wrap data in a Representational State Transfer (REST) request) and the authentication protocol used by the infrastructural element. It provides an API towards the *Virtual IoT Devices* to access the infrastructural element. *Virtual IoT Connectors* are internal software components and, hence, not exposed to upper layers in the architecture.

Asset Layer The *Asset Layer* models the application domain and consists of a set of *Assets*. Each *Asset* represents a physical component in the application domain. For instance, rooms and patients can be *Assets* in a care environment. Similarly, trucks and trailers can be *Assets* in a fleet management ecosystem. Each *Asset* defines an intuitive interface to interact with the physical component. The *Asset* implementations rely on the implementations of the *Virtual Device Layer*. A method in an *Asset* can invoke one or more methods in a *Virtual IoT Device*. For instance, retrieving the heart rate of a patient *Asset* will simply result in the invocation of a heart rate method in a heart rate sensor. Similarly, to get the location of a truck, the location of a GPS sensor will be requested. In some circumstances, the mapping is less trivial. For example, consider an *Asset* that contains a heart rate. Other than a `getHeartRate()` method, this *Asset* could also provide a `isHeartBeatHigh()` method to developers that notifies the application when a threshold heart rate is exceeded.

Application Layer The *Application Layer* contains the applications in the IoT ecosystem. These applications invoke the interfaces provided by (a set of) *Assets* and interact with the physical environment. Hence, application developers do not require knowledge of IoT communication technologies or protocols.

3.2 Quality of Service Support

To tackle QoS challenges, annotation of QoS requirements and IoT device specification cataloging are proposed. In four steps, QoS support can be added to existing IoT systems. We begin with eliciting relevant QoS properties of device types. Device technology properties are subsequently documented in creating a device catalog. By annotating the desired requirements during application design, and combining those with the catalog, selecting and coupling feasible IoT devices is simplified.

3.2.1 Elicitation of Relevant QoS Properties

For each device type in the IoT ecosystem, a list of relevant QoS properties is compiled along with one or more expected value types and their semantics, together with a list of supported units. For instance, the brightness of a lamp can be expressed in *lumen* and an *integer* value can be assigned as maximum value. Value types, as well as units, are not obligatory. If no values are given, the properties reflect about the presence or absence of a property in a particular device type (i.e., the brightness and/or color of certain lamps can be changed). Defining the units allow for automated translations between different units in tooling support.

Table 3.1 lists the relevant QoS properties of two device types (i.e., *Lamp* and *LightSensor*). Note that the number of quality properties that can be assigned to a device type can be huge. Besides the quality properties currently kept in the table (i.e., brightness, color and response delay), others could be added such as expected lifetime (in terms of lighting hours) or humidity resistance. At least the quality properties that are crucial for the application under design should be included.

Lamp		
property	**value type**	**supported SI units**
change_brightness	-	-
change_color	-	-
brightness	maxValue(int)	lm
response_delay	maxValue(int)	ms, s
LightSensor		
property	**value type**	**supported SI units**
response_delay	maxValue(int)	ms, s
polling_frequency	minValue(int), maxValue(int)	s^{-1}, min^{-1}, h^{-1}, d^{-1}
precision	value(double)	lx
data_range	minValue(int), maxValue(int)	lx

Table 3.1: Relevant QoS parameters for a lamp and a light sensor

3.2.2 Device Technology Catalog Creation

During device catalog creation, a set of device technologies (i.e., manufacturer and/or model) are assigned to each device type. For each technology, the list of QoS properties is instantiated together with a pointer to the plugin that can be used to add the product to the app without any substantial development effort (i.e., the device integrations from Section 3.1). Absence of such a plugin means that the app developer still has to bind the product to the application (i.e., develop the device integration).

Table 3.2 shows a device catalog for Lamps and Light sensors. Analogue tables can be constructed for the other device types and products and can be reused across multiple application domains. Note that some quality properties (such as *brightness* of *Lamps*) can be extracted from the product specification. Others are defined by experimental set-ups.

	Lamps		
	Philips Hue White	Philips Hue White and Color	IKEA TRÅDFRI
change_brightness	✓	✓	✓
change_color	✗	✓	✗
brightness	$800lm$	$800lm$	$960lm$
response_delay	$1s$ *	$1s$ *	$1s$ *
plugin	✓	✓	✗
	LightSensors		
	Arduino Bluetooth with Grove Light Sensor**	Arduino LoRa with Grove Light Sensor**	Versasense
response_delay	–	$1s$ *	$1s$ *
polling_frequency	$[1s^{-1}, \infty[$ *	$[12h^{-1}, \infty[$ *	$[6min^{-1}, 1d^{-1}]$
precision	$1lx$	$1lx$	$0,01lx$
data_range	$[0, 2000lx]$	$[0, 2000lx]$	$[0 - 16496lx]$
plugin	✓	✓	✓

Table 3.2: Device technology definitions for lamps and light sensors
*determined by experimental set-up
**http://wiki.seeedstudio.com/Grove-Light_Sensor/

3.2.3 Application Design

As described before, the application developer relies on technology-agnostic *Virtual IoT Device* calls to interact with sensors and actuators. During application design, it is still undefined which products will be coupled to the application. Device selection occurs at deployment time and depends on the preference of the IoT ecosystem owner. Some users or companies are willing to integrate very durable but expensive technologies. Others prefer cheaper solutions.

By creating a configuration file, the application developer can define the requirements for proper functioning of the application. The file contains the set of devices used in the application, together with their requirements (`QoSRequirements`). Each `QoSRequirement` contains the name of the property together with one or more

```
{
   "Lamp": {
      "id": "meetingRoomLamp",
      "QoSRequirement": [
         {  "importance": "required",
            "property": "max_response_delay",
            "value": "2",
            "unit": "sec",
            "purpose": "bigger delays can cause safety hazards" },
         {  "importance": "desired",
            "property": "change_brightness",
            "purpose": "changing room brightness",
            "degradation": "no brightness modifications" }
      ]
    },
   "LightSensor": {
      "id": "meetingRoomLightSensor",
      "QoSRequirement": [
         {  "importance": "required",
            "property": "minimal_polling_frequency",
            "value": "2",
            "unit": "requests/hour",
            "purpose": "historical data" },
         {  "importance": "desired",
            "property": "minimal_polling_frequency",
            "value": "1",
            "unit": "requests/minute",
            "purpose": "automatic brightness control",
            "degradation": "no automatic brightness control" }
      ]
   }
}
```

Listing 3.1: Example QoS configuration file

values and units when applicable. Furthermore, the developer defines the criticality of each quality property. This can either be *required* or *desired*. *Required* properties are needed for proper functioning of the application. Failing to meet *desired* requirements can result in service level degradation (i.e., functions might not be available in the application, but they do not break the entire application). Lastly, the developer can assign a purpose to each QoS requirement, and annotate the specific service level degradation for desired requirements. Including this information results in meaningful feedback towards infrastructure managers. The infrastructure manager can then rely on both the device catalog and the configuration-file delivered by the application developer for selecting the devices.

Listing 3.1 gives an example configuration file, it lists QoS constraints that can be expressed by app developers in an application containing lamps and lightsensors.

3.3 Integrating Internet of Things Devices

Previously, *Virtual IoT Devices* were introduced as a first-class software abstraction in the SMIoT architecture. Each *Virtual IoT Device* represents a sensor or actuator in the system. This software abstraction allows application developers to access each edge device in an intuitive way. It means that application programmers are not confronted with complex technology-specific communication handling or data semantics. In fact, the complexity of accessing IoT device functionality is reduced to calling methods with straightforward parameters on local objects.

Device selection is tackled by the device cataloging and annotating of the requirements of the application.

However, coupling the devices to the application is yet another concern. For instance, the application must have the right endpoint information (like name, address...) and credentials to access the IoT device. Setting up an IoT device (i.e., deploying and binding it in a concrete IoT ecosystem) cannot be done fully transparently. The wide variety of IoT platforms and authentication/authorization technologies complicates application binding.

This section classifies edge devices according to three categories, and shows how each category can be modeled in the SMIoT architecture. Thereafter, we distinguish three phases that must be foreseen to integrate edge devices in IoT ecosystems and thereafter couple them to applications.

3.3.1 Access Types of Internet of Things Devices

IoT devices can be separated in three groups, based on the way to access the data coming from the device. The proposed classification covers the wide majority of edge technologies currently on the market.

Direct Access The platform on which the end user application is deployed directly communicates with the IoT device. A prototypical example is a Bluetooth connected smart watch. Direct communication setups are often applied in single user or ad-hoc systems. Access control policies are often simple and static. Every application running on behalf of a user in possession of the credentials can access all device functions. The *Virtual IoT Device* virtualizes the physical device whereas the *Virtual IoT Connector* handles the communication with the physical device. The *Virtual IoT Device* passes the requests to its corresponding *Virtual IoT Connector*.

Gateway Mediated Device Access The user communicates with a gateway that can act as a broker for the IoT device. Smart lamps (like Hue en Osram) that are connected to a bridge are examples. There is only indirect access between the

application and the IoT devices. For instance, applications can only steer Philips Hue lamps via the Hue bridge (i.e., the gateway) when they are on the same local network. *Virtual IoT Connectors* support communication with the gateway instead of interacting with the IoT device directly.

IoT Platform Access The end user application can communicate to a cloud server to retrieve (historical) sensor data, or to actuate sensors. As with gateway mediated access there is only indirect access between the application and the IoT devices. Cloud platforms are often combined with one of the previous two access types. The data provided by the cloud platform can be different to the data provided directly by a device or a gateway, as it might already be aggregated. Users then have the option to choose the best way to access the IoT device based on requirements such as resource usage and sampling rate.

3.3.2 Integration Steps

Three steps often precede calling methods on local objects (i.e., device sensing or actuation). The ultimate goal is to couple an IoT device to an application running on behalf of a principal. During these steps access information and credentials are brought into the scope of the application.

Enrollment During this phase, physical IoT devices are deployed in the IoT ecosystem. The procedure is technology dependent. Thereafter, access information (both endpoint/address info and credentials) is maintained (either locally or centralized). Finally, each IoT device is linked to one or more assets. For instance, a Philips Hue lamp can be coupled to a certain room. Similarly, a heart rate sensor can be coupled to an individual. Enrollment typically occurs once per device, unless some sensors are re-used in other settings after a while.

Registration During this phase, access information and credentials are brought in the context of an application. The application interacts with a registrar that passes the required access info and credentials after successful authentication of the principal to the registrar. The registrar checks the access rights of the principal before it grants permissions to the application.

Device Access After successful registration, the application can access IoT devices. The application proves to be in possession of the required credentials. The device can optionally further constrain access based on contextual information (such as date, time and location). After successful access, the devices can be sensed or actuated (i.e., methods can be called on *Virtual IoT Devices*).

3.4 Framework Implementations

In this section, two frameworks are proposed to support developers in the implementation of the *Virtual Device Layer*. The approaches are for different platforms, the first one is an Android framework that can be used to build mobile Android IoT applications. The second framework is a JavaScript framework which can be used to develop various web applications.

The focus in this section is on the *Virtual Device Layer*. We don't go into detail on the *Asset Layer* because the work in further chapters have an influence on how the *Asset Layer* is modeled.

3.4.1 Android Framework

This section presents a software framework that supports developers with the implementation of the *Virtual Device Layer*. Although the framework presented in this section specifically targets Android, a similar approach can be taken for other platforms.

The source code of the Android framework can be found on
https://github.com/ilse-bohe/SMIoT-Framework

Figure 3.2 provides an overview of the file hierarchy of the framework. The framework code is split in two folders: `interfaces` and `implementations`. The former contains generic framework code and defines uniform interfaces to interact with devices in an intuitive way (e.g., `Lamp.java` and `TemperatureSensor.java`). The latter contains implementations of these interfaces for specific IoT technologies (e.g., `HueLamp.java` and LightifyLamp.java). The interfaces are defined as abstract methods in abstract classes.

For each type of device an abstract subclass of `VirtualIoTDevice` is defined that specifies the interface. Two types of sensing interfaces can be distinguished: *request-based* and *monitoring*. Request-based calls are used when a sensor value is needed just once (e.g., requesting the temperature in a room). Monitoring is used when a continuous stream of data is required (e.g., monitoring the heart rate of a patient). With request-based interfaces, the application expects a single response, while a call to a monitoring-type interface typically returns periodic responses.

The interfaces are defined as asynchronous. A callback parameter is added to each method declaration. This shields the component or application developer from the complexity of executing the interaction with IoT devices in a separate thread. This is required since interaction with IoT devices can introduce delays that are

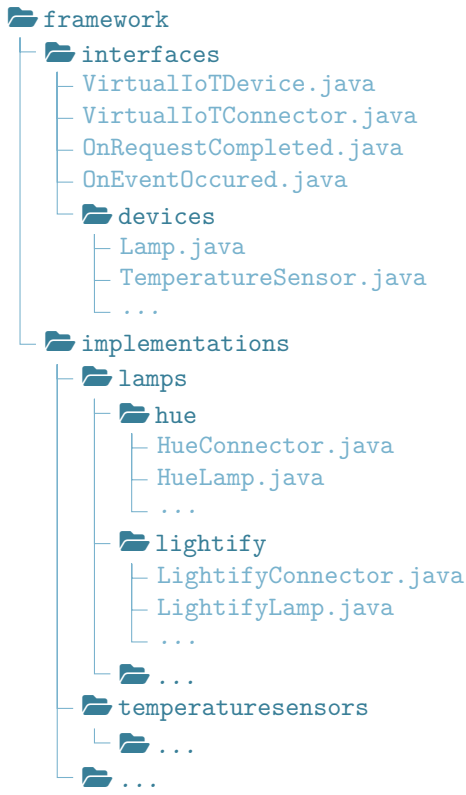


Figure 3.2: File hierarchy of the *Virtual Device Layer* in the Android framework

undesirable on the main thread. The framework defines two callback interfaces, one for request-based sensing/actuating (see `OnRequestCompleted`, Listing 3.3) and one for monitoring (i.e., `OnEventOccured`, Listing 3.2). An instance of this interface is passed as a parameter with each call to an IoT device by the application or component developers. These instances handle the response from the IoT device.

```
public interface OnRequestCompleted<T> {
  public void onSuccess(T response);
  public void onFailure(Exception exception);
}
```

Listing 3.2: `OnRequestCompleted` callback interface for request based device access

```
public interface OnEventOccurred<T> {
    public void onUpdate(T response);
    public void onErrorOccurred(Exception exception);
}
```

Listing 3.3: `OnEventOccured` callback interface for monitoring based device access

Since the (type of) parameters required to initialize *Virtual IoT Devices* and *Virtual IoT Connectors* is typically different for each type of IoT device, a set of keys is defined by the developer that specify each parameter required to initialize the *Virtual IoT Device* or *Virtual IoT Connector*. For the *Virtual IoT Connectors*, these parameters will typically be related to the used communication and authentication technology (e.g., Uniform Resource Locator (URL) or Bluetooth Media Access Control (MAC) address). For the *Virtual IoT Devices*, these parameters will be related to the identification of the specific device available on the connector endpoint and also a *Virtual IoT Connector* instance itself. Note that, if multiple *Virtual IoT Devices* reside on the same infrastructural element or are provided by the same gateway or IoT platform they are linked to the same *Virtual IoT Connector*. However, in case multiple infrastructural setups of the same technology are rolled out, multiple *Virtual IoT Connectors* must be constructed. For instance, if multiple Philips Hue bridges are deployed in a physical environment, a separate *Virtual IoT Connector* is constructed for each bridge. Based on the initialization interfaces, a generic device manager can automatically instantiate and initialize *Virtual IoT Devices* and *Virtual IoT Connectors* based on configuration files containing the required parameters.

A configuration file is kept by each application instance. Each configuration file describes a subset of the IoT environment, namely the subset of elements that are relevant for that application in a particular epoch. The latter means that the configuration files may vary over time. Each file consists of a set of *Assets*, *Virtual IoT Devices* and *Virtual IoT Connectors*. *Virtual IoT Devices* are coupled to *Virtual IoT Connectors* and *Assets*. A sample configuration file is given in Listing 3.4. Table 3.3 gives an overview of the parameters in the configuration file.

The location of configuration information depends on the specific ecosystem. A list of all *Virtual IoT Connectors*, *Virtual IoT Devices* and *Assets* in the IoT ecosystem can be kept centrally. However, this may negatively impact security and scalability. Distributing configuration info may be likely in many settings. A subset of *Virtual IoT Connector*, *Asset* and *Virtual IoT Device* settings are passed to the application instances, and stored in a configuration file. A registration phase typically realizes this step. Software objects are constructed in an application instance based on the content of the configuration file. For instance, *Virtual IoT Connectors* can be initialized based on the info in the configuration file. Also, *Virtual IoT Devices*

Virtual IoT Connectors	
`systemId`	A unique identifier chosen by the system designers. This identifier is technology-independent.
`type`	A technology dependent type of the *Virtual IoT Connector*.
`settings`	The settings needed by the *Virtual IoT Connector* to access the physical device, gateway or IoT platform. These settings depend on the technology and can contain values such as IP-address and access token.
Virtual IoT Devices	
`systemId`	A unique identifier chosen by the system designers. This identifier is technology-independent.
`type`	The device type defines the sensor or actuator class to which the *Virtual IoT Device* belongs. Examples are temperature-sensor, lamp, lock etc.
`technology`	A brand and model of the IoT device. It determines the underlying implementation that must be loaded in the application.
`settings`	Settings for identifying the physical IoT device such as the unique identifier or MAC address.
`connector`	The corresponding *Virtual IoT Connector* the *Virtual IoT Device* must be connected to, defined by the `systemId` of the *Virtual IoT Connector*.
Assets	
`assetName`	A unique name identifying the *Asset*.
`childAssets`	A list of *Assets*, defined by `assetName`, which are a part of this *Asset*.
`devices`	A list of *Virtual IoT Devices* connected to the *Asset*, defined by the `systemId` of the *Virtual Device*.

Table 3.3: Paramters in the SMIoT configuration file

can be created and linked to *Virtual IoT Connector* objects. Finally, *Assets* can be initialized, and each *Virtual IoT Device* can be added to the right *Asset(s)*. Replacing a physical device results in an update of the configuration file. The application must be notified in case an infrastructural modification or update occurs, and a delta configuration file must be stored in the context of the application. Devices that no longer need to be accessible by the application can be removed from the configuration file, and newly added devices must be initialized in the application together with their *Virtual IoT Connector*– if not yet present in the application. Thus infrastructural changes do not imply a change of the application logic and no additional implementation effort is needed.

```
{ "connectors": [
    { "systemId": "HGW1",
      "type": "PHILIPS_HUE_CONNECTOR",
      "settings": {
        "ip": "<ip-address of Hue bridge>",
        "authId": "<userToken>"
      }
    }
  ],
  "devices": [
    { "type": "lamp",
      "model": "PHILIPS_HUE",
      "systemId": "LAMP1",
      "settings": {
        "uniqueId": "<philips hue unique device identifier>"
      },
      "connector": "HGW1"},
    { "type": "lamp",
      "model": "HUE",
      "systemId": "LAMP2",
      "settings": {
        "uniqueId": "<philips hue unique device identifier>"
      },
      "connector": "HGW1" }
  ],
  "assets": [
    { "assetName": "Carehome",
      "childAssets": ["Room1","Room2"],
      "devices": []},
    { "assetName": "Room1",
      "childAssets": [],
      "devices": ["Lamp1"]},
    {"assetName": "Room2",
      "childAssets": [],
      "devices": ["Lamp2"]}
  ]
}
```

Listing 3.4: Example configuration file for an Ambient Assisted Living (AAL) system

Developers can add new types of sensors or actuators by creating a new abstract subclass of `VirtualIoTDevice` and defining the interface for that specific device. Developers can add support for specific IoT devices by subclassing the `VirtualIoTConnector` class and the abstract class(es) that define uniform interfaces of the IoT device, defining the initialization parameters and implementing the initialization and uniform interface methods. Support for IoT devices containing one type of sensor/actuator is contained in one class. If an IoT device consists of multiple types of sensors/actuators, a subclass for each sensor/actuator is created, sharing the same `virtualIoTConnector`. The framework already contains support for several IoT devices. Examples are *Hue lamps* and *Osram lamps*. Since the framework defines asynchronous interfaces, the implementation needs to delegate the interaction with the IoT devices to a separate thread and trigger the callback when the operation is completed. To realize this, the implementations provided with the framework use the RXAndroid framework, which is an Android port of ReactiveX[1]. This framework gives fine-grained control over the execution of operations on different threads. Since Android only allows User Interface (UI) operations on the main thread, our framework uses RXAndroid to execute the interactions with the IoT device on a worker thread while triggering the callback on the main thread. The obtained sensor values can then be show in the UI by the application developer, without overhead of inter-thread communication.

3.4.2 JavaScript Framework

This section describes a JavaScript framework to incorporate the SMIoT architecture in an IoT ecosystem. As with the Android framework, although the framework presented in this section specifically targets JavaScript, a similar approach can be taken for other platforms.

The source code of the JavaScript framework can be found on
`https://github.com/ilse-bohe/smiot-js`

The main difference with the Android framework described in Section 3.4.1 is the way *Virtual IoT Connectors* are implemented. In this JavaScript framework, *Virtual IoT Connectors* are more generalized, they are no longer defined based on device technology (e.g., Philips Hue, Versasense) but on communication technology (e.g., REST, Message Queueing Telemetry Transport (MQTT) and Bluetooth). A single *Virtual IoT Connector* is thus used by devices of different manufacturers.

Different driver types can be defined. Examples are REST, MQTT and Bluetooth. Each driver has a version, this to prevent broken code on driver updates. Each

[1] `http://reactivex.io`

specific driver is a subclass of the abstract class Driver and must implement the four abstract methods, `performWrite`, `performRead`, `performMonitor`, `stopPerformMonitor`.

For each device technology a configuration file must be provided. The configuration file defines the parameters of a device, together with more information on those parameters. It also defines the possible actions that can be taken on the parameters. These actions are one or more of the following:

- `read`, to request a parameter value once.
- `write`, to write the value of the parameter.
- `monitor`, to request a parameter value multiple times.

Together with each action, the driver that must be used to complete the action is defined. Settings used by the driver are defined, optionally with placeholders for device specific settings such as id's and IP-addresses. Then, to perform a `read`, `write` or `monitor` action, the driver method `performRead`, `performWrite` and `performMonitor` is used, respectively. Listing 3.5 shows part of the configuration file for Philips Hue lamps.

The central point of the JavaScript framework is the central engine. This engine is configured based on two configuration files. First, we have the configuration of the IoT devices. Secondly, the configuration of the assets in the ecosystem. Listing 3.6 shows a device configuration file, that defines the devices in an AAL ecosystem. Each device has an `id` and a `type`, a `type` corresponds to a device technology configuration file. Settings are defined that are used by the drivers to communicate to the physical device. Note that the settings of a specific device correspond with the placeholders in the device technology configuration files. A UI could be provided that dynamically creates the device configurations and asks the device manager about the necessary settings.

The second configuration file that is used to set up the engine defines the assets in the ecosystem. We are not going into further detail on these assets at the moment, as they are covered in more detail in Chapter 4.

The configured engine can then be used to request devices and or assets and perform actions on those devices and assets, respectively. As with the Android framework, asynchronous communication calls to the IoT devices must be taken into account. The Javascript port of ReactiveX can be used, and works with the same principles as the Android port.

Listing 3.7 shows an example of of a device (`lamp_1`) being requested from the engine, and being monitored.

```
[
  { "parameterReference": "lightstatus",
    "parameterTypeInfo": {
        "parameterType": "Boolean",
        "unit": null,
        "validator": {
            "type": "Boolean",
            "values": null
        }
    },
    "actions": {
      "Read": {
        "driverType": "RestDriver",
        "version": "v0.0",
        "settings": {
            "request-type": "GET",
            "access-point": "http://{ip}/api/{auth-id}/lights/{id}",
            "interpreter": "return response.state.on"
        }
      },
      "Write": {
        "driverType": "RestDriver",
        "version": "v0.0",
        "settings": {
            "request-type": "PUT",
            "access-point": "http://{ip}/api/{auth-id}/lights/{id}/state",
            "body": "{\"on\":<value>}"
        }
      },
      "Monitor": {
        "driverType": "RestDriver",
        "version": "v0.0",
        "settings": {
            "request-type": "GET",
            "access-point": "http://{ip}/api/{auth-id}/lights/{id}",
            "interpreter": "return response.state.on"
        }
      }
    }
  },
  { "parameterReference": "brightness",
    "parameterTypeInfo": {
        ...
    },
    "actions": {
        ...
    }
  },
  ...
]
```

Listing 3.5: Configuration file for *Philips Hue* lamps

```
[
        {
                "deviceId":"lamp_1",
                "type":"philips-hue-lamp",
                "settings" :
                {
                        "ip":"192.168.42.5",
                        "auth-id":"WpIBuOBDnR6s-TvM63sq1TFQ9AyqFehwMwDBSTXC",
                        "mac":"00:17:88:01:04:d2:9f:bb-0b",
                        "id": "1"
                }
        },
        {
                "deviceId":"lightsensor_1",
                "type":"versasense",
                "settings" :
                {
                        "mac": "00-17-0D-00-00-30-E9-62",
                        "pid1":"9803",
                        "pid2": "9805"
                }
        },
        {
                "deviceId":"heartrate_1",
                "type":"polar-bluetooth-device",
                "settings" :
                {
                        "mac":"A0:9E:1A:3B:37:60"
                }
        }
]
```

Listing 3.6: Example configuration file for the devices in an AAL system

```
import {Engine} from "smiot";

var engine = new Engine();
engine.setup('./configuration-files/assets.json',
↪  './configuration-files/devices.json');

var lamp = engine.getDevice("lamp_1");

lamp.monitorParameter("lightstatus", value => {
    console.log("lamp status is " + value)
});
```

Listing 3.7: Example JavaScript code to request and monitor an IoT device

3.5 Use Case 1: An Ambient Assisted Living Ecosystem

This section applies the previously described architectural principles to the AAL use case described in Section 2.4. We focus on the design of the *Asset Layer* and *Virtual Device Layer* in the care ecosystem, based on the Android framework described in Section 3.4.1.

The Virtual IoT Device Layer For each device type used in the AAL ecosystem, a uniform interface is defined. For actuators, this interface includes all actions that can be performed by that type of device. For example, Listing 3.8 provides the uniform interface of a Lamp.

```
public abstract class Lamp extends VirtualIoTDevice {
    abstract void turnOn(Callback<Boolean> cb);
    abstract void turnOff(Callback<Boolean> cb);
    abstract void changeColor(String clr, Callback<String> cb);
    abstract void changeBrightness(int bri, Callback<Integer> cb);
}
```

Listing 3.8: Code example for the abstract class `Lamp`

As described in Section 3.4.1 each method has a callback (`Callback<...> cb`) that returns the result of the call asynchronously.

In the case of a sensor, the uniform interface usually consists of two methods: one for requesting the current value, and one for monitoring purposes over a certain time interval. In Listing 3.9, the uniform interface for a heart rate sensor is provided. Analogously a uniform interface is provided for the other sensors and actuators.

```
public abstract class HeartRateSensor extends VirtualIoTDevice{
    abstract void requestHeartRate(Callback<Integer> cb);
    abstract void monitorHeartRate(Callback<Integer> cb);
}
```

Listing 3.9: Code example for the abstract class `HeartRateSensor`

For each device technology type (i.e., model and vendor), a one-time implementation effort is required to create an implementation that is compliant with the uniform interface. If a specific sensor/actuator technology does not support a method of the interface, an error is thrown.

The Asset Layer Using Figure 2.6, two parent *Assets* can be defined: the `Carehome` and the `Resident`. The `Carehome` consists of three child *Assets*: the

`Bathroom`, the `Bedroom`, and the `Livingarea`. On its turn, the latter has a `Kitchen` and `LivingRoom` as child *Assets*. Each *Asset* includes the complete set of all IoT devices that it can contain, and defines the API that is provided to the *Application Layer*. Note that units are not required to be equipped with all devices defined in the design. The *Asset* diagram is presented in Figure 3.3.

The Application Layer The *Application Layer* relies on the APIs defined in the *Asset Layer*. Depending on the application instance (and thus the type of user), a different subset of *Asset* APIs and *Virtual IoT Devices* is used. For example, the application for the resident or caregiver receives configurations that allows the application to load all *Virtual IoT Devices* related to one care home unit and one resident, and provides functionality to sense and control all devices located in that room. The maintenance application requires access to devices located in several units. However, the set of device types they can access is limited to their current assignment.

3.6 Use Case 2: Integrating Health and Activity Wearables in Mobile Applications

In this section, together with the architectural principles, the QoS support is validated by another Android prototype to monitor body parameters.

3.6.1 Approach

Our approach consists of three steps. First, technology agnostic interfaces are defined. Second, an inventory of meaningful IoT technologies is built, and relevant QoS parameters are assessed for each technology in the catalog. Finally, the plugins are developed, at least for a subset of devices in the inventory. We concentrate throughout the description on the domain of well-being and health, and especially focus on how communication technology abstraction is established. The approach is applied to heart rate sensors that can later be used for different purposes. However, the approach can be applied for the integration of other sensors as well.

Technology Agnostic Interface Definition A uniform interface is defined for each type of sensor. This interface is independent of the applied device technology (i.e., the brand and model) and underlying communication protocol. In our example, a uniform interface will be defined for heart rate sensors. Each interface consists of a type-independent part for management and identification purposes, and a type-dependent part that focuses on functionality. Generic management and identification methods are assigned to a common *Virtual IoT Device* that can be inherited by concrete sensor types. Three non-functional methods are assigned to a *Virtual IoT*

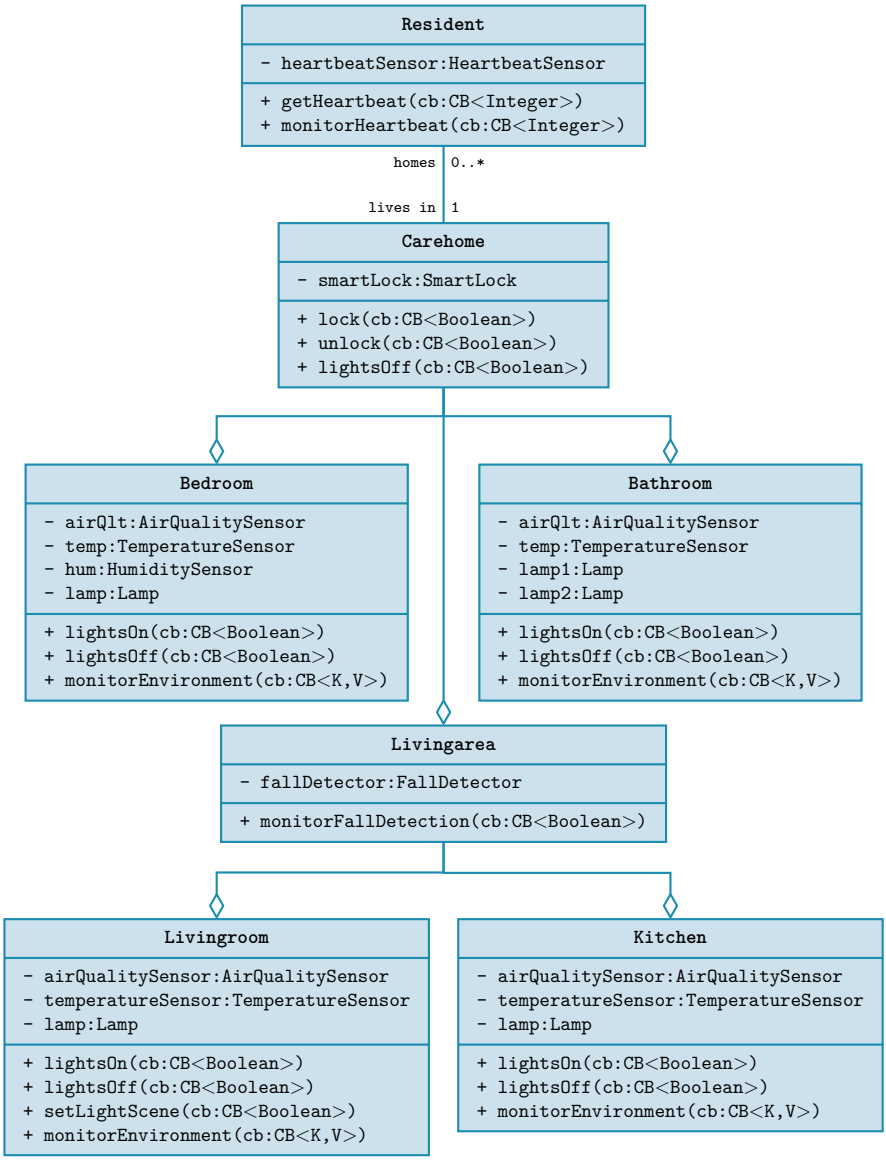


Figure 3.3: Representation of the *Asset Layer* in an AAL ecosystem

Device. The method `isReachable()` checks whether the IoT device can be reached from within the application. The methods `connect()` and `disconnect()` provide the possibility to connect and disconnect the IoT device to/from the application

respectively. Each of the aforementioned methods returns a boolean. For the `isReachable()` method this boolean defines whether the IoT device is accessible. The return value of the other methods `connect()` and `disconnect()` report about the success or failure of the executed action.
The type-dependent part needs to be redefined for each device type. For example, a heart rate sensor, body temperature sensor and an activity tracker have different methods. Note that an implementation for this interface is needed for each device technology. Each interface defines a uniform representation of the device's functional methods, together with its attributes and their type. In the case of a heart rate sensor, methods will be foreseen to retrieve and monitor the heart rate. Some heart rate sensors provide the heart rate in BPM and others in Hz. Our interface fixes the unit (either BPM or Hz), and interface implementations must eventually convert sensed values to meet the type definitions. The functional methods can be categorized according to different characteristics. A distinction can be made between methods requesting real-time data (RT) and historical (H) data. Applications requesting real-time data often interact with the sensor directly, whereas historical data is often available at an external source like a gateway or a cloud platform. Not all sensor manufacturers necessarily support access to both real-time and historical data by third-party apps. For example, in case of FitBit technology (like many other IoT technology providers), external application developers can only request historical data via the FitBit cloud platform. Direct access to real-time data generated by the activity tracker is not supported for third party developers (due to closed APIs). A second distinction can be made between request-based and monitoring methods. Request-based methods are used when a (set of) sensor value(s) is needed only once. Monitoring methods are used when a stream of sensor data sensed over a certain time interval is beneficial. The functional methods that are defined for a heart rate sensor are listed in Table 3.4. The method `requestHR()` is a request-based method. It expects no input parameters and returns the real time heart rate value. The monitoring method, `monitorHR()`, expects no input parameters and returns the real-time heart rate value periodically until the `unmonitorHR()` method is called. Finally, the `historicalHR(startDate, endDate, granularityLevel)` method is a request-based method. As input parameters it requires a start and end date (and time). The historical values within that time interval must be retrieved with a certain granularity level. The latter defines the maximum acceptable time interval between two successive values. The method returns a list of tuples containing timestamps and the corresponding heart rate value. In case the sensor provider does not support historical data, an error must be returned. An error can also be thrown if the required granularity level cannot be met.

QoS Cataloging After the interfaces are defined, specific sensor technologies can be coupled to the device types. Our sample catalog consists of one chest band, two wristlets and one smart watch each sensing heart rate. The selected chest band is

Method	Param	Result	RT/H	Type
requestHR	-	value	RT	request
monitorHR	-	value	RT	monitoring
endMonitorHR	-	-	—	—
historicalHR	startDate endDate granularity	list[dateValuePair]	H	request

Table 3.4: Functional Methods for a Heart Rate Sensor

	Polar H7	Polar M600**	Fitbit Charge 2	Garmin Forerunner 35
Hardware Platform	chest band	smart watch	wristlet	wristlet
Real-time hr	✓	✓	✗	✗
sampling rate	$1s$***	$2.2s$***	-	-
accessibility	Bluetooth	Bluetooth	-	-
Historical hr	✗	✓	✓	✓
sampling rate	-	$5s$	$15s$	unkown
accessibility	-	Bluetooth	Cloud	Cloud
Accuracy*	1	0.7	0.2	0.5
Precision	$1bpm$	$1bpm$	$1bpm$	$1bpm$
Autonomy	$200h$	$2d$	$5d$	$9d$

Table 3.5: Selected heart rate sensor technologies
*compared to the Polar H7
**a dedicated application was deployed on the Polar M600 which makes real-time and historical heart rates available via Bluetooth.
***from experimental results

the Polar H7[2]. The wristlets are the Fitbit Charge2[3] and the Garmin Forerunner 35[4]. The selected smart watch is the Polar M600[5]. The Polar H7 is a dedicated heart rate sensor integrated in a chest band. Electrodes are used to measure the heart rate. The accuracy of electrode technology outperforms the ones embedded in writlets and smart watches. The latter contain an optical sensor to measure heart rate frequencies. The Polar M600 is a smart watch running Wear OS[6]. Custom wearable applications can be developed for smart watches running Wear OS. A dedicated application that can retrieve the user's heart rate and share these values

[2] https://support.polar.com/en/support/H7_heart_rate_sensor
[3] https://www.fitbit.com/en/shop/charge2
[4] https://buy.garmin.com/en-US/US/p/552962
[5] https://www.polar.com/uk-en/products/sport/M600-GPS-smartwatch
[6] https://wearos.google.com/

with other (mobile) platforms over Bluetooth has been developed and deployed on the Polar M600.
A non-exhaustive but relevant set of quality properties are assigned to each heart rate technology. The device selector can opt for a particular heart rate technology based on those properties. The properties for the aforementioned technologies are listed in Table 3.5. The Polar devices expose an open API to retrieve real-time heart rate values. Although these values can also be retrieved by a closed application in the Fitbit and Garmin technologies, those technologies do not expose an external API to provide real-time heart rate values to third-party apps. Furthermore, historical data is available via Bluetooth with Polar M600, and both Fitbit and Garmin expose a cloud API. The precision and autonomy for each device were extracted from the specification in the device manuals. Quantifying the accuracy is complex but accuracy can be a decisive parameter in critical applications. To quantify this QoS parameter, we assign a value $x \in [0, 1]$ to this parameter for each device, and rely on a comparative study that takes the Polar H7 as baseline (i.e., the accurary of this chest strap is defined as 1). The accuracy of the other technologies is calculated as a function of the overall variance v of the technology compared to the chest strap, namely $1/(1 + v)$.

Plugin Development The interface must be implemented for at least a subset of products in the catalog. The set of technologies that are selected highly depend on QoS needs that are imposed by the applications under development. Thereafter, the devices can easily be integrated in various IoT applications without any additional effort by application developers. It is trivial that multiple implementations are necessary if multiple device technologies are used. These device plugins can then be packaged with the applications to let them function properly depending on the used sensors and actuators. Alternatively, novel plugins can be added as new devices come available.

Note that there is not necessarily a one-to-one mapping between *Virtual IoT Devices* and physical devices.
On the one hand, one *Virtual IoT Device* can consist of multiple physical devices. This strategy increases the reliability and/or the accuracy of the data received in the application. Reliability can thus be improved in situations where a device fails. If so, measurements can automatically be taken over by redundant devices. Similarly, accuracy can be increased by fusing multiple sensor values. As certain sensor technologies are more reliable than others, weighted averages can be calculated. For instance, electrode-based chest heart rate monitors are more accurate than wristlet monitors [51]. Incorrect device functioning can be discovered by comparing sensor values of multiple devices. Multiple implementations can be foreseen, balancing non-functional concerns like accuracy, reliability, performance, and battery power.
On the other hand, one physical device can be mapped to multiple *Virtual IoT Devices*. In that case, the device must implement the interface of each *Virtual IoT*

Device. For instance, in the AAL scenario, two *Virtual IoT Devices* can be defined, namely a *FallDetector* and a *VideoSensor*. The former can be used to detect falls of elderly people, and at least contains a method `monitorFall()`. The latter allows relatives to access a video stream after a fall was suspected by the fall detector. Fall detectors can be realized by various technologies, ranging from pendants [90], over wrist bands [53, 57] to floor mats [38, 60]. However, a camera can be used for both fall detection and video streaming. In this case, both interfaces need to be implemented. Other examples are sensor kits[47, 71] to which multiple sensors are plugged in.

3.6.2 Prototype

The proposed approach is applied to a health and wellness application integrating heart rate sensing, monitoring and analytics as well as functionality supporting maintenance purposes. Although multiple applications can easily be developed that offer partial functionality, our prototype is representative for a wide range of applications by integrating all functionality in one mobile application. The application is developed in Android, the most used platform for health and activity tracking applications [114].

IoT Care Application The mobile application can be split in three functional parts. First, owners of the application can retrieve and monitor real-time heart rate values. Second, historical values can be retrieved and passed to medical staff after user consent. Finally, the application supports sensor (re)configuration. The latter means that a heart rate sensor can be replaced by another sensor technology meeting threshold quality requirements.

The heart rate sensors that are integrated in the prototype are the Polar H7 and the Polar M600. The decisive selection criteria was the fact that real-time heart rate data should be available to the app. The Fitbit and Garmin technologies do not support that functionality. Moreover, a hybrid version was developed to increase reliability. It allows to detect malfunctioning or disconnected devices rapidly. Figure 3.4 depicts a part of the Unified Modeling Language (UML) class diagram for the AAL system. The `VirtualIoTDevice` abstract class contains the three abstract non-functional methods `isReachable()`, `connect()` and `disconnect()`. The abstract class `HeartRateSensor` extends the `VirtualIoTDevice` and contains the functional methods listed in Table 3.4. For both the Polar H7 and the Polar M600 an implementation is made of the `HeartRateSensor` class. Thus these software components implement the abstract methods provided by both `VirtualIoTDevice` and `HeartRateSensor`. A hybrid heart rate sensor is provided combining the Polar H7 and the M600. This implementation increases reliability by providing both sensors simultaneously. If one sensor falls out due to communication or device failures, the remaining one can still be used to track the user's heart

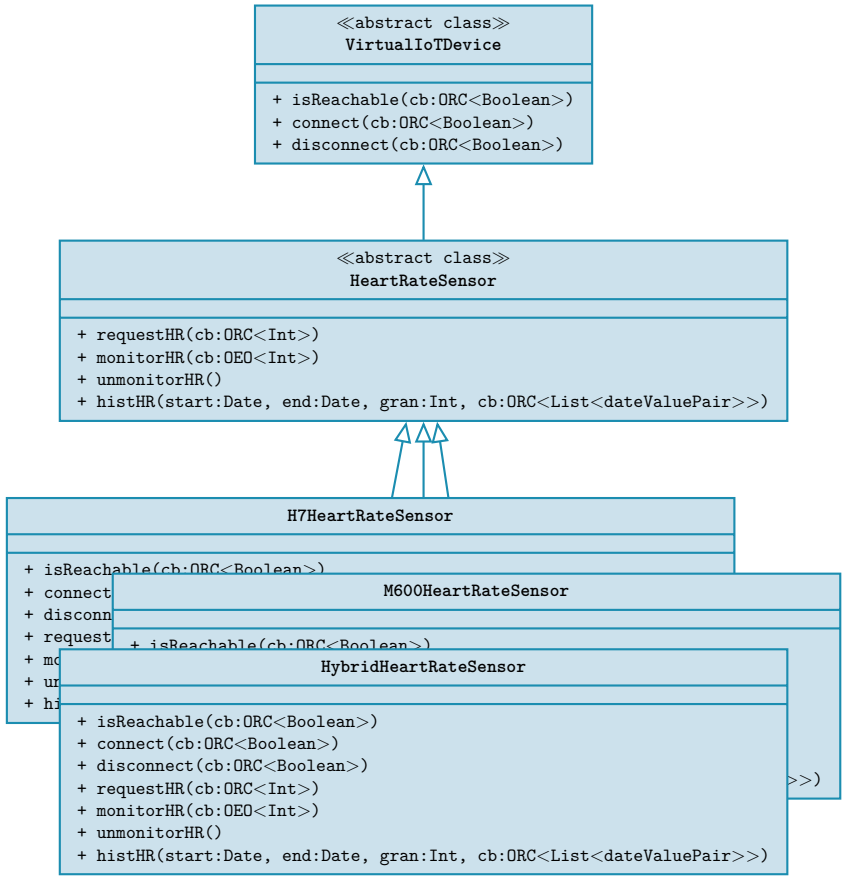


Figure 3.4: The `HeartRateSensor` abstract class and its implementations

rate. The `HybridHeartRateSensor` nests both an `H7HeartRateSensor` and `M600HeartRateSensor`, low-level connectivity implementations must therefore only be carried out once for each physical device.

Listing 3.10 shows how a *Virtual IoT Device* is used. A heart rate sensor `hrs` and text view `tv` are declared. When monitoring the heart rate sensor, the `newHeartRate` is displayed in the text view on each update. In case an error occurrs, the `onErrorOccurred(Exception exception)` method is executed. Note that the listing clearly demonstrates that the application code is independent of the device brand nor the underlying communication technology.

As the interfaces define asynchronous methods, the interaction with the devices

must be executed on a different thread than the main thread. This is handled by the *virtual IoT device* implementations and is transparent towards application developers. This can be achieved using RxAndroid, an Android port of ReactiveX[7] which is a Framework for asynchronous programming. RxAndroid makes it is easy to pass device interaction to a worker thread while the result can be observed and handled on the main thread.

```
HeartRateSensor hrs;
TextView tv;
//...
hrs.monitorHR(new OnEventOccurred<Integer>() {
    @Override
    public void onUpdate(Integer newHeartRate) {
        tv.setText(newHeartRate +" bpm");}
    @Override
    public void onErrorOccurred(Exception exception) {
        exception.printStackTrace();}
});
```

Listing 3.10: Code example for the use of the abstract class `HeartRateSensor`

Mobile Application Assessment The mobile application has been applied during work-out sessions. Both the Polar H7 and the Polar M600 were coupled to the mobile application to monitor a cyclist's heart rate. The heart rates that were returned by a prototypical bicycle ride of 20 minutes are depicted in Figure 3.5. The time indicates the moment at which the heart rate was received by the mobile device. It can clearly be seen that the values of the Polar H7 were received slightly earlier than the values of the polar M600 (i.e., the Polar M600 has longer communication delays). Also, the Polar H7 presents a higher frequency than the M600. Based on multiple observations the frequency of both devices is calculated. The average frequency of the polar H7 is 1 Hz. This means that the value updates approximately every second. The Polar M600 has an average frequency of 0.45 Hz. Each 2.2 seconds an update is received from the Polar M600. During the ride presented in Figure 3.5 connection was lost several times with the Polar H7. When using the `H7HeartRateSensor` implementation for the `HeartRateSensor` this would result in periods where no values are retrieved. The `HybridHeartRateSensor` monitors both the H7 as the M600. Hence, heart rate values of the M600 are still received even if the Polar H7 does not send out values anymore. This increases the reliability and thus the user experience of the monitoring application. Note that an `AdvancedHybridHeartRateSensor` implementation could bring the accuracy of the sensors into account and calculate the heart rate more accurately.

[7] http://reactivex.io/

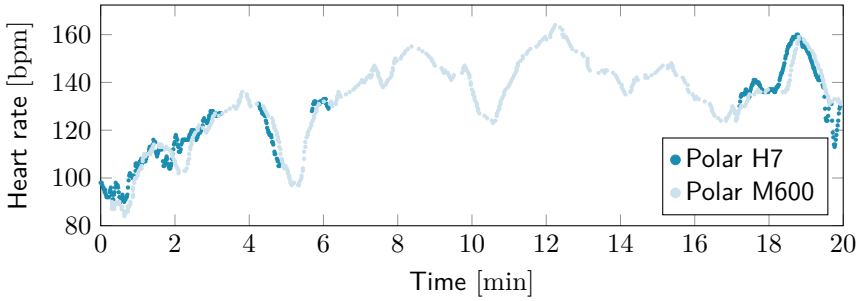


Figure 3.5: Heart rate values during a bicycle ride

3.7 Discussion

This section discusses the impact of more complex setups on the architecture and deliberates several aspects that need to be taken into account when using the architecture.

In many circumstances, multiple applications, possibly on behalf of different stakeholders, require access to the same physical device. For many commercial devices, it is not trivial to maintain connections to multiple applications. Examples are many Bluetooth devices or other devices that wipe access tokens each time another device authenticates. In those situations, it is often advantageous that an IoT device remains connected to one application for a longer time. That application can then act as a proxy for other applications. Hence, other applications are able to retrieve sensor data or send actuation commands via the proxy application. For example in the AAL ecosystem, a heart rate sensor can be connected to a health application running on a resident device. Caregivers can request or monitor those values via that proxy application. Similarly, a tablet that is mounted in a fixed location in the living room can be persistently connected to lamps in a care home. Both the resident and caregiver application can then control the lamps via that tablet. In addition, the proxy approach can also improve the security level. First, credentials to access edge devices can be stored in the context of the proxy application. Hence, loading IoT credentials on multiple end-user applications is no longer required, which improves security management. Moreover, this approach can tailor access control to the needs of the IoT ecosystem and their applications.

Advanced access control can be incorporated by granting access based on a set of contextual parameters such as the time and the role and identity of the principal.

Currently, commercial smartphone platforms support permission handling based on user consent. Examples are permissions to access user data (i.e., contacts) and system features (i.e., camera). Similarly, users controlling an IoT application would benefit from a permission-based mechanism that regulates access towards the IoT

devices in the system. However, IoT applications, developed using the architecture and framework, present a high level of dynamics and heterogeneity. Different application instances installed form the same build can lead to totally different types and technologies that need to be coupled. The static permission handling currently provided by Android and iOS has shortcomings and are not sufficient to provide the required fine-grained access control for larger IoT applications.

Several non-functional aspects related to IoT device selection (i.e., aspects other than which IoT device interfaces are supported) have a significant impact on the behavior of the application. Hence, it is not always possible to replace an IoT device with any other device providing the same interfaces. Some applications require a specific QoS from IoT devices to ensure the desired behavior. This is specified via QoS parameters such as measurement resolution, latency, sampling frequency, security, form factor and communication range. For example, IoT devices using Low Power Wide Area Network (LPWAN) technologies typically have a longer latency compared to devices using low-range communication technologies such as Bluetooth. Depending on the use case a device with a specific measurement resolution or latency is selected. For instance, for temperature monitoring in smart city applications, an IoT device with long-range communication, low measurement resolution and sampling frequency may be selected while these sensors are not adequate for climate control applications. Other applications rely on advanced features only present in a subset of devices. For instance, applications aiming at creating various atmospheres in a room, rely on color and brightness. Not all lamps support those properties.

The *Virtual IoT Device* framework makes abstraction of the monitoring frequency. The system administrator can specify the frequency at which sensor values are monitored. For IoT devices that support monitoring (i.e., publish-subscribe messaging protocols such as MQTT [74]) this is typically defined as a parameter on the IoT device. For devices that do not support these types of protocols, the monitoring interface can be implemented via software-based polling by the *Virtual IoT Device*. Although this enables more flexibility in the specification of the monitoring frequency (i.e., the monitoring frequency can be provided as a parameter to the *Virtual IoT Device*), it can negatively impact the performance of the application (e.g., computational load, battery life). This is especially the case if sensor values should not be monitored at a fixed frequency but only if they pass a certain threshold (e.g., trigger an alarm if the temperature is too high). In this case, polling also significantly reduces the battery life of the sensor.

The *Virtual Device Layer* and *Asset Layer* provide a high-level abstraction of the underlying IoT infrastructure to application developers. In some cases, even significant changes in the underlying infrastructure have no impact on the *Application Layer*. For instance, presence of people can be detected either via infrared sensors or via cameras. In some care homes, cameras are used for presence detection because

they are also used for security services. In other care homes, infrared sensors are selected for cost reasons. An application developer uses the same presence interface on the *Asset* (e.g., `Room`), regardless of the underlying infrastructure.

Most applications contain both IoT and non-IoT related functionality. This is also reflected in the interfaces provided by the *Assets* to application developers. For example, care home applications typically require access to medical records of the *Resident*. Hence, the IoT-related interfaces of *Assets* are often complemented with interfaces supporting more traditional application-level operations.

For systems that are retrofitted to include IoT applications, often *Assets* can wrap existing object representations and add the IoT-related interfaces to the already existing functionality.

3.8 Conclusion

This chapter presented an IoT architecture to support application centric development in IoT ecosystems. Taking into account the need for separation of duties due to the mixed development skills in large development teams, a layered architecture is proposed that decouples the applications from the IoT infrastructure. The *Virtual Device Layer* contains virtualizations of the physical IoT devices and presents intuitive interfaces to the upper *Asset Layer*. This *Asset Layer* contains *Assets* representing items of interest in the ecosystem and also provides intuitive interfaces to the upper *Application Layer*. Application developers can thus focus on business logic, without taking into account the underlying device and communication technologies. Device integration can be done at a later stage and is separated from the application development. To maintain QoS, device cataloguing is proposed. Depending on the proposed QoS requirements, the best applicable devices can be selected. This ensures that reconfigurable applications can be developed that are easy to maintain.

Based on the proposed architecture, both an Android and JavaScript framework are developed and validated using the AAL use case. By developing a mobile application for health and activity tracking, the work on QoS was also validated.

Chapter 4

Designing IoT Ecosystem Environments

The content from this chapter is previously published in:

- I. Bohé et al. "Untangling the Physical-Digital Knot When Designing Advanced IoT Ecosystems". In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things (M4IoT)*. UC Davis, CA, USA, 2019, pp. 1–6. ISBN: 9781450370288

In a good Internet of Things (IoT) application, devices are made transparent to the user. IoT applications originate from a higher goal. A fitness application, for instance, is developed to monitor health and fitness parameters of a person. A smart home application, on the other hand, is used to simplify user interactions in a house. The goal of fleet management applications is to track and monitor trucks, trailers and cargo. For the users of a fleet management application, it is hardly ever of any concern if the location of a trailer is tracked using Global Positioning System (GPS) technologies or using cellular location tracking technologies. From the upper examples, we can deduce that users reason based on *Assets*.

A framework to support *Asset*-oriented development is already proposed in Chapter 3. It is, however, of great importance that the environment is modeled in a proper way. The dynamic nature of IoT ecosystems must be taking into account, as well as the diverse skill set present in a development team.

Contributions The work in this chapter presents an application-centric approach to model and maintain scalable yet reconfigurable IoT ecosystems by applying three key tactics. First, clear *separation-of-duties* between application designers and IoT infrastructure managers improves manageability. Second, *loose coupling between business logic and IoT infrastructure* advances reconfigurability. Third, we propose *late and selective binding* of sensors and actuators to applications in order to achieve favourable scalability, security and privacy properties. Note that our work mainly focuses on cost-efficient integration of commercial-of-the-shelf (COTS) devices, rather than building dedicated endpoint hardware. We demonstrate the impact of our approach throughout the whole life cycle of an IoT ecosystem, and apply the proposed tactics to the development and operations of an Ambient Assisted Living (AAL) environment.

The rest of this chapter is structured as follows. Section 4.1 refines the scope and elucidates the requirements. Section 4.2 describes the general approach. The impact on the IoT environment and application design and development are discussed in Section 4.3. Section 4.4 focuses on infrastructure management and operations, followed by a discussion in Section 4.5.

4.1 Scope and Challenges

The methodology aims at supporting the development and maintenance of advanced IoT ecosystems. Within that area, we especially focus on durable software applications in edge systems. Opposed to centralized (or cloud) IoT ecosystems, edge applications directly interact with the IoT endpoints or via a local gateway. Durability means that the lifetime of the software applications need to survive the IoT device lifetime. Malfunctioning or outdated sensors and actuators should easily be replaceable by cheaper or more qualitative alternatives over time. Moreover, the approach aims at facilitating the integration of COTS solutions in software systems. The COTS component internals are thereby considered as black box. Finally, the following assumptions are made at development time.

1. The specific structure of the physical world nor the IoT components and technologies that are rolled out are known.
2. The device types can be modeled but knowledge about the specific instances and the specific sensor instances (i.e., model and vendor) is lacking during development.

3. The application developers are no experts in IoT device and communication technologies. They must be able to focus on application logic instead of low-level communication functionality and data conversions.

The use case presented in Section 2.4 is also used in this chapter to clarify the concepts. The rest of this section recites the major non-functional challenges when modeling IoT ecosystems.

4.1.1 Non-functional Concerns

Scalability and (re)configurability are key non-functional concerns in our approach. Both challenges bring along security and privacy requirements.

Scalability Taking into account the number of stakeholders, applications and IoT devices that are rolled out, scalability can be expressed. In the AAL environment, various stakeholders (elderly, relatives and caregivers) using different applications interact with the physical world via a growing number of IoT sensors and actuators.

(Re)configurability Multiple reconfigurability dimensions can be defined. First, multiple application instances are rolled out. Different assets and IoT devices are coupled to each application. On top of that, various IoT technologies are used in different environments. Second, application instances can evolve over time. Both assets in the physical world as well as the IoT infrastructure are tailored to the use case. The set of elderly people under supervision of a caregiver evolve, additional IoT sensors rolled out or existing ones are often replaced or removed.

Security In edge oriented IoT systems, security can be complicated. Asset, IoT device and credential management is no sinecure in distributed IoT ecosystems in which a huge number of sensors and actuators are rolled out.

Privacy As sensitive data is collected and processed in many application domains, it is important to consider the privacy of the user. IoT infrastructure deployed in houses or attached to individuals collect a lot of personal sensitive information. Similarly, IoT systems deployed in companies may collect and process a lot of sensitive business data. User control and transparency became mandatory in recent EU privacy legislation [44]. This implies, amongst others, that individuals need to have control about the behaviour of their software applications. They need to give explicit consent for data monitoring, and to be informed about the purpose of personal data collection and processing.

4.1.2 Case Study

To demonstrate its applicability, our methodology is applied to the design and operations of the AAL ecosystem, proposed in Section 2.4. Elderly people and their residences are key assets in that perceived physical world. Elderly health and activity parameters are monitored by means of body area sensors and IoT devices

that are rolled out in their residences. Moreover, the ecosystem supports monitoring and controlling of environmental parameters (like air quality, heat and clarity).

Two applications are developed in the AAL system. Elderly and their relatives can interact with a *Human Machine Interface (HMI) Application* installed in the residence to monitor well-being and control the residence parameters. Moreover, caregivers can rely on a *Mobile Application* to control a restricted set of residence and health parameters. Caregivers can only inspect parameters of the elderly people and residences under their supervision, and only if they are present in the residence of a particular individual, or in case of an emergency. Note that both applications were implemented on the Android platform proposed in Section 3.4.1.

4.2 Modeling Approach

This section first introduces basic terminology. Thereafter, the architectural tactics are discussed in details. Finally, a general overview is presented of the different phases in a typical IoT ecosystem life cycle.

4.2.1 Basic Concepts

An IoT ecosystem defines a set of software applications running on behalf of users (or principals) interacting with the physical world by inspecting and controlling IoT devices (i.e., sensors and actuators). We define two perspectives on an IoT ecosystem, namely a design perspective and an operational perspective. The former refers to the modeling concepts, the latter to an instance of the model in specific settings. A lot of concepts defined in the design perspective have a dual in the operational perspective.

Design Perspective	Operational Perspective
$Type_{Asset}$	*Asset*
$Model_{Environment}$	*Environment*
$Type_{Device}$	*Device*
Application	$Instance_{Application}$

Table 4.1: Dual concepts in the design and operational perspective of the IoT ecosystem

The following concepts are defined in the design perspective.
***Type*$_{Asset}$** An *Asset Type* defines an item of interest in the physical world for an IoT ecosystem. Elderly, caregiver and residence are *Asset Types* in the AAL use case. Similarly, truck, trailer, truck driver and cargo can be *Asset Types* in a fleet management use case.

***Model*$_{Environment}$** The IoT environment model includes the set of *Asset Types* and defines relations between them, together with a cardinality. For instance, one or more elderly live in a residence and a set of elderly are assigned to a caregiver. Similarly, multiple cargo items can be coupled to a trailer which, on its turn, can be assigned to a truck. A truck driver navigates a truck.
***Type*$_{Device}$** A *Device Type* defines a sensor or actuator type of interest in an IoT ecosystem. Examples are lamps, air quality sensors, cameras and heart rate sensors in the AAL ecosystem. *Device Types* can be coupled to one or more *Asset Types*. If so, it implies that an *Asset Type* can be monitored or controlled by a particular *Device Type*. For instance, one or more lamps and/or air quality sensors can be coupled to a residence. Similarly, a heart rate sensor can be assigned to an elderly.
Application An *Application* defines a software application that interacts with the physical world. A set of *Application Policies* can be assigned to an *Application*. An *Application Policy* defines functional behavior of an *Application* expressed in terms of (conditional) operations on *Assets*. *Application Policies* are independent of the underlying infrastructure (i.e., *Device Types*).

Dual concepts are defined from the operational perspective.
Asset Each *Asset* defines a specific instance of an *Type*$_{Asset}$. For instance, both `Alice` and `Bob` are elderly. Similarly, both `R1` and `R2` are residences.
Environment An *Environment* defines relations between *Assets*, and its composition is restricted by *Model*$_{Environment}$. For instance, `Alice` lives in residence `R1` , `Bob` lives in residence `R2`.
Device A specific IoT sensor or actuator is defined by a *Device*. Each *Device* belongs to a *Type*$_{Device}$, is part of the IoT infrastructure and is assigned to at least one *Asset*. For instance, an instance of a Philips Hue lamp is assigned to residence `R1`. Similarly, an instance of a Polar heart rate sensor is assigned to the resident `Bob`.
***Instance*$_{Application}$** An *Application Instance* defines a specific deployment of an *Application*. Different caregivers can for example have different deployments of the same *Mobile Application*.

4.2.2 Architectural Tactics

Three major tactics are applied throughout the whole life cycle of an IoT ecosystem. They aim at tackling the concerns discussed in Section 4.1.1.
Clear separation-of-duties Between software designers, developers and IoT environment managers, there must be a clear separation-of-duties. This improves manageability of large-scale and reconfigurable IoT ecosystems. During the software design phase, designers model the IoT environment *Asset Types* and relations between them. IoT *Device Types* are modeled to meet the monitoring and control needs of *Assets* in the physical world. The software developers can then, during software development, get to work with the design to develop the *Application*. IoT environment managers are responsible for maintaining an inventory of *Assets*

and IoT *Devices* in a specific IoT environment, and for binding IoT *Devices* to *Assets*. This happens during the deployment and maintenance phases. End users (or principals) need to give informed consent before an *Application* can actually start sensing or actuating IoT *Devices*.

Decoupling the software applications from the IoT infrastructure Decoupling facilitates reconfigurability and is favored by application-centric development. This means that application developers focus on business logic from the early design stages. Binding IoT infrastructure (i.e., specific IoT technology instances) to the software *Application* is postponed to a later stage. This strategy is opposed to sensor-centric development in which IoT device technologies are fixed in early stages. Note that many sensor-centric IoT ecosystems lack flexibility and suffer from vendor lock-in due to inferior design decisions. IoT device virtualization can increase reconfigurability. Applying layered software architectural principles further allow to separate business logic from IoT infrastructure, as described in Chapter 3.

Selective and late device binding Scalability and security are favored by enabling selective and late binding. Five zones are proposed and depicted in Figure 4.1. Each zone can reduce the number of sensors and actuators by a multitude.

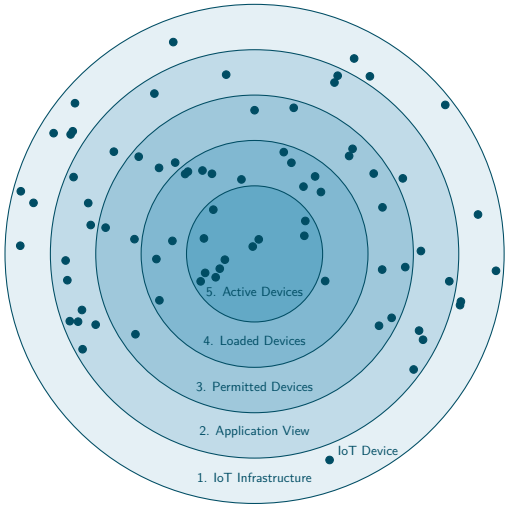


Figure 4.1: Selective and late IoT device binding

1. The outer zone defines the complete set of IoT *Devices* in the IoT ecosystem infrastructure.

The other zones are application-specific.

2. The *Application* view defines the subset of *Devices* that could be used by an $Instance_{Application}$. For instance, the *Application* view of Alice's *HMI Application* will only include her personal body sensors and the IoT infrastructure rolled out in her own residence.

3. Zone 3 defines the subset of *Devices* in zone 2 for which explicit user consent was given by means of a permission handling mechanism. Similarly, an end user can also revoke permissions which moves *Devices* back to zone 2.

4. Zone 4 defines the subset of *Devices* for which configuration info (like access point and credentials) is loaded in the application. This can either occur transparently or via user interaction, but always after user consent.

5. The inner zone defines the subset of loaded *Devices* that are sensing or actuating at a given time, possibly triggered by a contextual behavior defined in *Application Policies*.

Selective sensing and actuation does not only improve privacy but can also help in restraining resource usage (e.g., battery and memory usage).

4.2.3 General Overview

Our modeling approach is split in two phases. The first phase (presented in Section 4.3) consists of environment and application design and development, and results in practical output that can be applied during roll-out and operations. The scope of output is presented in tables for clarity but design tool support to transform the output to eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) format is propsed in Chapter 6. The second phase (presented in Section 4.4) incorporates asset and infrastructure management and operations, and consists of five steps.

4.3 Design and Development of IoT Environments and Applications

Before an *Application* is designed and developed, the environment must be designed. This is explained in detail and step by step in Section 4.3.1. After that, application design can start, followed by the development of the *Application* (i.e., Section 4.3.2).

Figure 4.2 shows the steps taken during environment and application design and development. A clear separation of duties can be established. Each step is either assigned to the environment designer, or application developer. *Application Design* can already start after the *Domain Definition*, as the applications logic is designed device technology agnostic.

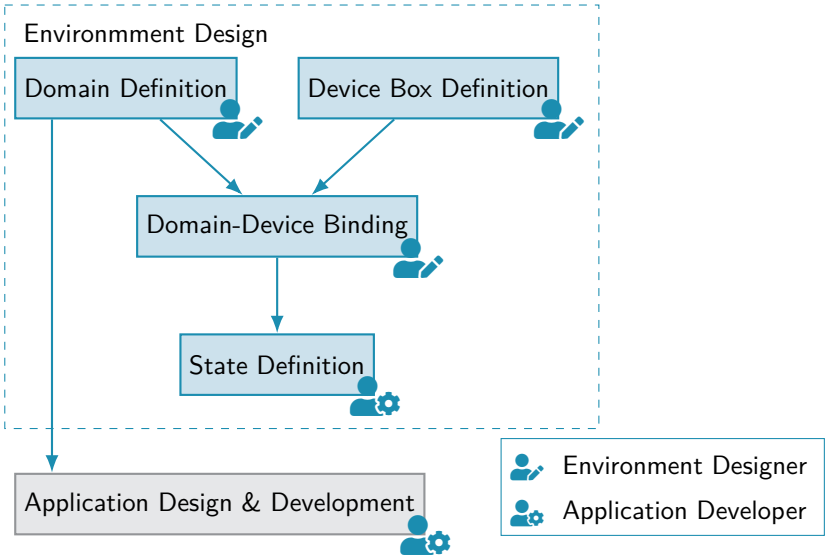


Figure 4.2: Steps to design and develop an IoT environment and application

4.3.1 Environment Design

The environment design consist of four steps, namely *Domain Definition*, *Device Box Definition*, the binding between them in *Domain-Device Binding*, and lastly the definition of *States*, which are explained further, in *State Definition*. As shown in Figure 4.2 the *Domain Definition* an *Device Box Definition* are independent.

Domain Definition During the *Domain Definition*, the *Environment Designer* defines the *Asset Types* ($Type_{Asset}$) in the IoT ecosystem under design, together with the $Model_{Environment}$. The latter defines the relations between the *Asset Types*. Figure 4.3 depicts the $Model_{Environment}$ for the AAL ecosystem. Four *Asset Types* are defined in the AAL ecosystem, namely `Residence`, `Person`, `Elderly` and `Caregiver`. Both caregivers and elderly are persons. For simplicity, the current model assumes that exactly one elderly person lives in each residence, and residences are not further split in separate rooms. A caregiver is assigned to each elderly and a caregiver may be assigned multiple elderly.

Multiple *Parameters* may be assigned to one or more *Asset Types*. Moreover, a set of *States* can be assigned to some *Parameters*. Figure 4.4 depicts the *Asset Type-Parameter-State* metamodel. Table 4.2 returns the *Parameters* together with their accompanying $Type_{Asset}$, and the *States* these *Parameters* have in the AAL ecosystem.

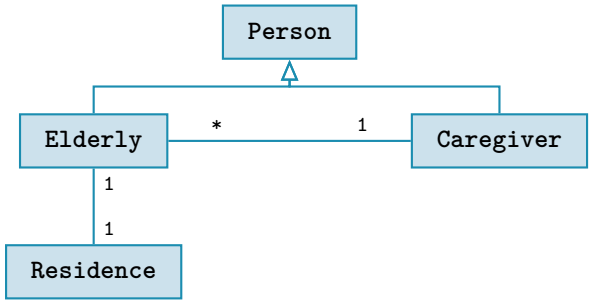


Figure 4.3: $Model_{Environment}$ in the AAL ecosystem

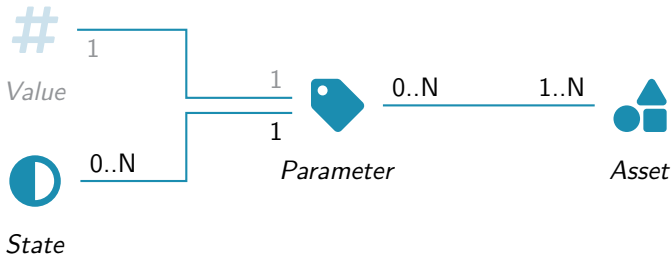


Figure 4.4: *Asset Type-Parameter-State* metamodel

Parameters are either static or variable. In contrast to static *Parameters*, the value of variable *Parameters* can change due to modifications in the physical world. A static *Parameter* assigned to `Residence` is `address`. Similarly, `name` is a static *Parameter* assigned to `Person`. Their value cannot be altered by a change that

Parameter	$Type_{Asset}$	*State*
`address`	`Residence`	-
`heat`	`Residence`	`cold, normal, hot`
`clarity`	`Residence`	`dark, normal, bright`
`appearance`	`Residence`	-
`name`	`Person`	-
`presence`	`Person, Residence`	`present, absent`
`heartRate`	`Elderly`	`low,normal, high`
`healthiness`	`Elderly`	`normal, alarming`
`workStatus`	`Caregiver`	`work, free`

Table 4.2: *Parameters* in the AAL ecosystem

is caused by influences from the physical world. Variable `Residence` *Parameters* are `heat` and `clarity`; `heartRate` is a variable `Elderly` *Parameter*. Note that a *Parameter* can be coupled to multiple *Asset Types*. For instance, `presence` only makes sense with respect to both `Residence` and `Person` (either a caregiver or an elderly). *Parameter Values* can be grouped in *States*. Therefore, a discrete and finite set of *States* can be assigned to a *Parameter*. A *State* declaration, if specified, must be complete. Each *Parameter Value* then belongs to one of these *States*. For instance, the `clarity` in a `Residence` can either be `dark`, `normal` or `bright`. Note that it is not mandatory to couple *States* to each *Parameter*. Both `address` and `name` are *Parameters* whose *Values* do not belong to a *State*.

Device Box Definition During the *Device Box Definition* step, the IoT *Device Types* ($Type_{Device}$) are modeled and kept in an inventory. As depicted in Figure 4.5, each *Device Type* is either a *Sensor* or *Actuator*, and has one or more *Parameters* assigned. For instance, `heartRate` is assigned to `HeartrateSensor`. Similarly, `brightness` and `color` are assigned to `Lamp`. Some *Parameters* can be assigned to multiple *Device Types*. For instance, the *Parameter* `temperature` is sensed by `TemperatureSensor` and actuated by `Thermostat`.

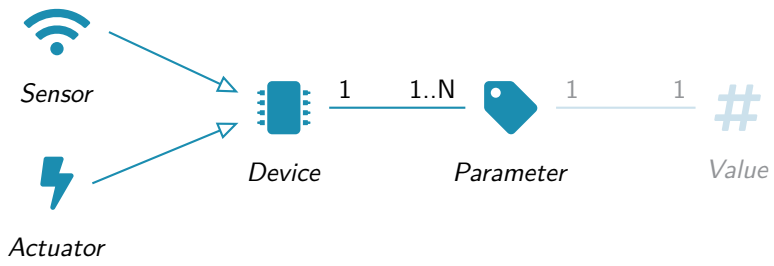


Figure 4.5: *Device-Parameter* metamodel

Both *Asset Types* and *Device Types* have *Parameters*. For clarity, we make a distinction between the two, However, *Devices* can be equated with *Assets* and treated in the same way.

Domain-Device Binding During the *Domain-Device Binding*, IoT *Device Types* (i.e., elements of $Type_{Device}$) can be coupled to one or more *Asset Types* (i.e., elements of $Type_{Asset}$). For instance, a residence can have a lamp, light sensor and temperature sensor. Similarly, an elderly can wear a heart rate sensor. A cardinality (or cardinality range) must be assigned to each $Type_{Asset}$-$Type_{Device}$ binding. The default cardinality is 1 and means that exactly one device of a certain type is coupled to an *Asset*. For instance, we can define that each residence must be equipped with exactly one thermostat. A cardinality range 0..1 implies that a thermostat

$Type_{Device}$	*Sensor / Actuator*	*Parameter*
`TemperatureSensor`	*Sensor*	`temperature`
`Thermostat`	*Actuator*	`temperature`
`LightSensor`	*Sensor*	`intensity`
`Lamp`	*Actuator*	`brightness, color`
`Camera`	*Sensor*	`videofeed`
`PositionSensor`	*Sensor*	`position`
`HeartrateSensor`	*Sensor*	`heartRate`
`FallDetector`	*Sensor*	`status`

Table 4.3: $Types_{Device}$ in the AAL ecosystem

can be lacking. A 0/1..N cardinality means that none, one or more sensors of a certain type can be assigned to an *Asset*. A residence could be equipped with multiple light sensors, but it is not mandatory. Throughout the rest of this section, we assume that all bindings between *Asset* and *Device Types* have cardinality 1, however, increasing the cardinaliy to higher values is straightforward. Table 4.4 gives an overview of the $Type_{Asset}$-$Type_{Device}$ bindings in the AAL system, all having a default cardinality of 1.

$Type_{Asset}$	$Type_{Device}$
`Residence`	`TemperatureSensor`
	`Thermostat`
	`LightSensor`
	`Lamp`
	`Camera`
`Person`	`PositionSensor`
`Elderly`	`HeartrateSensor`
	`FallDetector`

Table 4.4: $Type_{Asset}$-$Type_{Device}$ AAL bindings

During *Domain-Device Binding* the *Parameters* are also connected to IoT *Device Types*. A binding implies that the IoT *Device Type* is used to inspect the corresponding *Asset Parameter* if the former is a *Sensor*. An *Actuator* connected to an *Asset Parameter* implies that an instance of the $Type_{Device}$ is used to modify the *Asset Parameter*. For instance, a `LightSensor` inspects the `clarity` in a `Residence`. The `clarity` in a `Residence` is controlled by a `Lamp`. Table 4.5 gives an overview of AAL *Parameter-Device* bindings.

State Definition A set of non-overlapping *States* can be coupled to each *Asset Parameter*. Furthermore, a *State Definition* is assigned to each *State*. Table 4.6

Parameter	$Type_{Asset}$	$Type_{Device}$
heat	Residence	TemperatureSensor Thermostat
clarity	Residence	LightSensor Lamp
presence	Person, Residence	PositionSensor
heartRate	Elderly	HeartrateSensor
healthiness	Elderly	HeartrateSensor FallDetector
appearance	Residence	Camera

Table 4.5: *Parameter-Device* bindings in the AAL ecosystem

Parameter	*State*	*State Definition*
heat[R]	cold	R :: TemperatureSensor.temperature ≤ 15
	normal	$15 <$ R :: TemperatureSensor.temperature < 22
	hot	R :: TemperatureSensor.temperature ≥ 22
clarity[R]	dark	AVG($\forall$(R :: LightSensor).intensity) ≤ 20
	normal	$20 <$ AVG($\forall$(R :: LightSensor).intensity) < 100
	bright	AVG($\forall$(R :: LightSensor).intensity) ≥ 100
presence[P,R]	present	P :: PositionSensor.position WITHIN R.address
	absent	presence[P,R] $\neq$ present
heartRate[E]	low	E :: HeartrateSensor.heartRate ≤ 50
	normal	$50 <$ E :: HeartrateSensor.heartRate < 160
	high	E :: HeartrateSensor.heartRate ≥ 160
healthiness[E]	normal	heartRate[E] = normal $\wedge$ E :: FallDetector.status = normal
	alarming	heartRate[E] = low $\vee$ heartRate[E] = high $\vee$ E :: FallDetector.status = fall
workStatus[C]	work	*derived from a working schedule*
	free	*derived from a working schedule*

Table 4.6: *Parameter States* in the AAL ecosystem
(R=Residence; P=Person; E=Elderly; C=Caregiver)

lists the set of *State Definitions* in the AAL system. Each *State Definition* consists of a set of conditions or their negation that can be combined by logic operators. On their turn, each condition includes a relational operator and operands. A relational operator can be an (in)equality or more complex relation between operators (like

WITHIN in the `present` and `absent` *State Definitions*). An operand can be an *Asset Parameter* (e.g., `R.address`), a *Parameter* of a *Device* that was coupled to an *Asset Parameter* at an earlier stage (e.g., `R::TemperatureSensor.temperature`), a *Device Parameter* value (e.g., `15`, `normal` or `fall`), or another *Parameter State Value* (i.e., `normal`, `low` and `high`). Having this knowledge, it can easily be seen from Table 4.6 that the `presence` of a person `P` in a residence `R` depends on the `address` of the residence `R` and the `position` of the `PositionSensor` that is coupled to the person `P`. Similarly, the `healthiness` of an elderly `E` depends on her `heartRate` *State* and the `status` of the `FallDetector` coupled to `E`. Note that the formerly defined $Type_{Asset}$-$Type_{Device}$ bindings and their cardinality have an impact on valid *State Definitions*. For instance, the heat in each residence can be expressed in terms of the temperature sensed by the temperature sensor as we assumed that exactly one temperature sensor is coupled to each residence. However, if the $Type_{Asset}$-$Type_{Device}$ bindings would allow for one or more temperature sensors assigned to the same residence (i.e., cardinality range 1..N), the heat state can be determined as a function of the sensed temperature of all temperature sensors in that residence. A feasible function would be a (weighted) average.

4.3.2 Application Design and Development

Based on the *Environment Design*, applications are created to support different business logic functionalities. Access control, automation and data preprocessing are several examples. Using policies, the working of these functionalities can be defined in a user friendly way. In Chapter 5 this will be discussed in more detail. For now, a high level access control example will be elaborated. First the policies are specified, subsequently we dive deeper into the development of the application.

Application Policy Specification The set of *Asset Parameters* that can be inspected and/or controlled for each *Application* in the IoT ecosystem are defined. Table 4.7 gives an overview of the *Application Policy* specification for the two *Application* in the sample AAL ecosystem. The *HMI Application* is managed by an elderly E_P (i.e., the owner of the residence where the application is deployed) and consists of two versions. $HMI_{v1.0}$ is a fully automatic application which does not support any user interaction. In contrast, in $HMI_{v2.0}$ elderly can interact via an intuitive Graphical User Interface (GUI) (i.e. by pushing software buttons or moving sliders). The *Mobile Application* is managed and used by a caregiver C_P. For instance, the residence's clarity can be inspected and controlled by both *Application*. Similarly, the residence's heat can only be inspected and controlled by the *HMI Application*, and the heart rate of an elderly can be inspected by both the*HMI Application* and the *Mobile Application*. A set of conditions can restrict inspection and/or control of *Asset Parameters*. Analogue to the *State Definitions*, conditions can be concatenated by logic operators. Each condition

HMI Application **(principal: Elderly E_P)**		
Parameter	Action	Condition
heat[R]	inspect	$R \rightarrow Elderly = E_P$
	control	$R \rightarrow Elderly = E_P$ $[\wedge$ heat[R] $=$ cold|hot$]^*$
clarity[R]	inspect	$R \rightarrow Elderly = E_P$
	control	$R \rightarrow Elderly = E_P$ $[\wedge$ presence[P,R] $=$ present$]^*$
presence[P,R]	inspect	$E = E_P \wedge R \rightarrow Elderly = E_P$
heartRate[E]	inspect	$E = E_P \wedge$ presence[E,R] $=$ present $\wedge$ $R \rightarrow Elderly = E$
healthiness[E]	inspect	$E = E_P \wedge$ presence[E,R] $=$ present $\wedge$ $R \rightarrow Elderly = E$
appearance[R]	inspect	$R \rightarrow Elderly = E_P$ $[\wedge$ healthiness[E_P] $=$ alarming$]^*$
Mobile Application **(principal: Caregiver C_P)**		
Parameter	Action	Condition
heat[R]		*never*
clarity[R]	inspect & control	$R \rightarrow Elderly \rightarrow Caregiver = C_P$ $\wedge$ presence[C_P,R] $=$ present $\wedge$ workStatus[C_P] $=$ working
presence[P,R]	inspect	$C = C_P \wedge R \rightarrow Elderly \rightarrow Caregiver = C_P$
healthiness[E]	inspect	$E \rightarrow Caregiver = C_P$ $\wedge$ presence[E, R] $=$ present $\wedge$ $R \rightarrow Elderly = E$
heartRate[E]	inspect	workStatus[C_P] $=$ working $\wedge$ $E \rightarrow Caregiver = C_P$ $\wedge$ presence[C_P,R] $=$ present $\wedge$ $R \rightarrow Elderly = E$
appearance[R]	inspect	$R \rightarrow Elderly \rightarrow Caregiver = C_P$ $\wedge$ healthiness$[R \rightarrow Elderly]$ $=$ alarming

Table 4.7: *HMI Application* and *Mobile Application Policies* in the AAL system $HMI_{v1.0}$ is fully automatic and, hence, more restrictive. $HMI_{v2.0}$ supports user interaction. Additional constraints in $HMI_{v1.0}$ are defined between $[\ ]^*$.

either defines a relation between *Assets* (like $\texttt{C} = \texttt{C}_\texttt{P}$ *or* $\texttt{R} \rightarrow \texttt{Elderly} = \texttt{E}$) or a relation between *Parameter States* (like $\texttt{presence[E, R]} = \texttt{present}$ *or* $\texttt{healthiness}[\texttt{R} \rightarrow \texttt{Elderly}] = \texttt{alarming}$). Note that, in contrast to the *State Definitions*, *Device Parameters* or their *Values* cannot be operands in *Application Policies*. This implies that policy definitions are fully infrastructure agnostic. This has many advantages. Application designers can focus on business logic and do not need to have knowledge about the underlying infrastructure when defining *Application Policies*. Also, modifications to the underlying IoT infrastructure do not have an impact on the *Application Policies*. Finally, *Application Policies* can easily be understood by end users. Note that $\text{HMI}_{v1.0}$ and $\text{HMI}_{v2.0}$ have slightly different *Application Policies*. Additional conditions must be fulfilled in $\text{HMI}_{v1.0}$ to inspect the appearance in a residence, and to modify the heat and clarity. The additional conditions are defined between starred brackets (i.e., [ ]*). For instance, the fully automated $\text{HMI}_{v1.0}$ will only control the temperature of the residence if it is hot or cold, whereas the residence owner (i.e., the elderly $\texttt{E}_\texttt{P}$) can always modify the residence's heat in the second version (i.e., $\text{HMI}_{v2.0}$).

Application Development The IoT environment modeling and *Application Policy* specification aim at facilitating application development. As demonstrator, both the *HMI Applications* and the *Mobile Application* are developed in Android relying on the software framework proposed in Section 3.4.1. The proposed structured modeling approach guides the component layer implementation and virtual device layer configuration.

The subset of *Assets* and *Asset Parameters* from the overall IoT environment that need to be accessible and, hence, instantiated in an *Application Instance* can easily be extracted from the *Application Policies* depicted in Table 4.2. The relations between the subset of *Assets* in the AAL system can be derived from Figure 4.3. Note, for instance, that caregivers C do not need to be defined in the *HMI Applications*. In contrast, the *Mobile Application Policies* rely on the caregivers (C), residences (R) and elderly (E). Similarly, it can occur that only a subset of *Asset Parameters* need to be defined. For instance, defining the residence's heat is only relevant in the *HMI Applications*. Similarly, the subset of *Device Types* that are required in an *Application Instance* can be extracted from the *Application Policy* and the table that maps *Asset Parameters* and operations (i.e., inspect *and* control) to *Devices* (i.e., sensors *and* actuators) and *Device Parameters*. It becomes clear that temperature sensors, thermostats and fall detectors only need to be virtualized in both *HMI Applications*. On the contrary, heart rate sensors and cameras need to be imported in all applications. This implies that the uniform interfaces must be imported, together with a set of plugins or at least a reference to a plugin store for a particular *Device Type*. The former strategy, however, restricts the technologies that can be rolled out in the IoT infrastructure. To increase openness and flexibility, the plugins can be inserted at a later stage (i.e., at deployment time or even at

runtime). Note also that the required links between the *Assets* and IoT *Devices* can be extracted from the *Type*$_{Asset}$-*Type*$_{Device}$ bindings. For instance, in the component layer, the elderly E is linked to a heart rate and positioning sensor in all applications, whereas the elderly E is also linked to a fall detector in both *HMI Applications*.

4.4 Management and Operations

The stages defined in Section 4.3 focused on the modeling of meaningful *Assets* and *Devices* within a particular domain, and on the design and development of feasible applications within that domain. This section describes the steps that are required to roll out and manage specific IoT ecosystems, consisting of *Assets*, IoT infrastructure and *Application Instances*. We first show how *Assets* and *Devices* are managed. Thereafter, we focus on the life cycle of two different application instances that are rolled out in the IoT ecosystem.

To demonstrate the scalability of our approach, the sample AAL ecosystem consists of numerous instances, namely 220 *Assets*, 3320 infrastructural elements (physical *Devices*) and 120 *Application Instances*. We assume a commercial health provider that employs 20 caregivers C_y with $y \in [1, 20]$, and rolls out AAL IoT infrastructure and applications. We assume 100 elderly E_x with $x \in [1, 100]$ and 100 residences $R_{x'}$ with $x' \in [1, 100]$. Elderly E_x lives in residence $R_{x'}$ for $x = x'$. Each caregiver C_y has responsibility over five elderly E_x with $y = \lceil \frac{x}{5} \rceil$. Furthermore, each elderly E_x maintains a HMI$_{v2.0}$ *Application Instance* HMI^x with $x \in [1, 100]$, and each caregiver C_y controls a *Mobile Application Instance* Mob^y with $y \in [1, 20]$.

30 IoT *Devices* are rolled out in each residence R_x, namely 10 lamps $La_i^{R_x}$ and 10 light sensors $Ls_i^{R_x}$ with $i \in [1, 10]$, 6 webcams $Ca_j^{R_x}$ with $j \in [1, 6]$, and 2 temperature sensors $Tm_k^{R_x}$ with $k \in [1, 2]$ and 1 thermostat Ts^{R_x}. Furthermore, each elderly E has three wearables, namely a heart rate sensor Hr^{E_x}, a fall detector Fd^{E_x} and a position sensor Ps^{E_x}. Each caregiver C_y is equipped with a position sensor Ps^{C_y}, possibly built into the mobile device. Note that the number of IoT *Devices* of each type that are coupled to *Assets* is fixed for sake of clarity. For some devices types, the cardinality deviates from the initial specification in Section 4.3 in which the default cardinality (i.e., 1) is applied for each binding. Increasing the cardinality allows to show the scalability of the proposed approach. Modifying the cardinality only has a minor impact on *State Definitions* in Table 4.6. For instance, the clarity in a residence can be calculated by taking the average of the light intensities of the 10 sensors that are deployed, and is no longer determined by the light intensity of one light sensor. An analogue reasoning applies to the *State Definition* of the heat in a residence. Note that taking cardinality ranges instead of cardinality numbers would also lead to small modifications of the *State Definitions*.

4.4.1 IoT Ecosystem Management

The *IoT Ecosystem Manager* [82] is a software tool that keeps track of *Assets*, IoT *Devices* and *Application Instances* in the IoT ecosystem. Although a centralized commercial health provider currently runs the *IoT Ecosystem Manager*, and thus, has a complete overview of *Assets*, *Devices* and *Application Instances*, it can be extended for distributed asset and infrastructure management. The tool aims at easing multiple IoT management tasks. First, it supports the IoT manager with inventorying $Type_{Asset}$ and their *Parameters*, and relations thereof. Each *Asset* must belong to one of the formerly defined $Type_{Asset}$, and static *Parameters* (e.g., *address* or *name*) can be initialized. The IoT environment modeling restricts the relations between *Assets*. For instance, each elderly E_x can only be coupled to one residence $R_{x'}$, and vice versa. Second, it keeps a catalog of supported device technologies including a pointer to the corresponding plugin to invoke methods from the virtual device layer. Third, it keeps an inventory of the IoT devices that are rolled out in the physical world, together with their unique identifier, technology instance, access point information and credentials. The $Type_{Device}$ restricts the device types that can be integrated in the IoT ecosystem. Fourth, the tool supports linking IoT infrastructural elements to *Assets*. The modeling phase restricts the *Device Types* that can be linked to *Asset Types*. For instance, a light sensor can only be coupled to a residence, not to an elderly or caregiver, due to constraints imposed by the IoT modeling phase. Finally, the *IoT Ecosystem Manager* keeps track of installed *Application Instances*, together with their principals.

Multiple queries can be performed on the inventory that can be applied to take intelligent decisions in later phases.

Querying Asset Types The following query (Q_1) returns the *Asset Types* that are required in *Application Instance* App_X when the *Application Policies* P_{App} are applied.

$$Q_1{:}setT_A^X \leftarrow needed\_asset\_types(App_X, P_{App})$$

Note that *Asset Types* map to templates that are typically loaded during the whole lifetime of the *Application Instance* App_X, and hence, need to be packaged with the *Application* at install time. It can easily be derived from the *Application Policies* that the elderly and residence *Asset Types* are needed in both *HMI* and *Mobile Application Instances* whereas the caregiver template is only required in *Mobile Application Instance*.

Querying Assets The second query (Q_2) returns the set of *Assets* that need to be accessible with respect to a particular *Application Instance*.

$$Q_2{:}setD_A^X \leftarrow needed\_assets(App_X, P_{App}[, cond])$$

The *Assets* can be derived from the *Application Policies*. For instance, one elderly E and one residence R need to be accessible in each *HMI Application*. R refers to the residence in which the *HMI Application* is deployed. The elderly E refers to the inhabitant of residence R. Similarly, one caregiver C, five residences R and five elderly E need to be accessible by each *Mobile Application Instance*. A condition $cond$ can be defined as an additional parameter of the query. It refers to a (sub)set of assets that need to be accessible when that condition $cond$ is fulfilled. For instance, the query $needed\_assets(Mobile_{Carol}, P_{App}, presence(Carol, R_i))$ returns the set of *Assets* that need to be accessible in the *Mobile Application* run by a principal $Carol$ if $Carol$ is present in a certain residence R_i. The set contains the principal $Carol$, the residence R_i and the elderly E_i living in that residence if the caregiver $Carol$ is responsible for elderly E_i. Otherwise, R_i and its inhabitant E_i are not returned.

Querying Device Types Query 3 (Q_3) returns the *Device Types* that are required in *Application Instance* App_X when the *Application Policies* P_{App} are applied.

$$Q_3{:}setT_D^X \leftarrow needed\_device\_types(App_X, P_{App})$$

The set can be derived from the needed *Asset Types*, the relevant *Parameters* together with their operation mode (i.e., inspect and/or control), and the table that maps *Asset Parameter* monitoring to *Device Types*. Accessing those *Device Types* is required to enforce the *Application Policies* P_{App}. The query result determines the plugins that should at least be loaded in the *Application* App_X when *Application Policies* P_{App} applies.

Querying Devices Similarly to Q_2, this last query (Q_4) returns the set of *Devices* that need to be accessible with respect to a particular*Application Instance*, and can be derived from IoT ecosystem management information optionally under a given condition $cond$.

$$Q_4{:}setD_D^X \leftarrow needed\_devices(App_X, P_{App}[, cond])$$

4.4.2 Application Instantiation

During *Application* instantiation, the code is packaged for a particular *Application Instance*. Besides a common code base that is shared by all *Application Instances* of the same type, a set of *Asset Types*, *Device Types* and *Device* plugins need to be installed that may be different for each *Application Instance*. The set of required

Asset and *Device Types* can be derived from Q_1 and Q_3 respectively. One option is to package all device plugins that correspond to a particular device type with a given *Application Instance*. However, this may lead to unnecessary packaging of plugins that are never used. An alternative is to execute Q_4 on the *Application Instance* without $cond$ argument. Q_4 returns the *Devices* that are currently rolled out, and that might be accessible by the *Application Instance* during its lifetime. Each *Device* is labeled with a *technology* attribute which refers to the plugin that needs to be installed. The drawback of this approach is that *Application Instances* must foresee a mechanism to load new plugins to deal with changes in the IoT infrastructure over time. Our current Android prototypes applies the first option. However, the footprint can be reduced considerably if a lot of technologies are supported but hardly used in practice.

4.4.3 Permission Handling

Infrastructure owners need to give permission to *Application Instances* running on behalf of principals to monitor and/or actuate physical devices. Multiple permission handling strategies can be applied to achieve this goal. The *HMI Application* is managed by elderly E_P and needs access to multiple *Devices* that are owned by that principal. By relying on the mobile platform permission handling mechanism, namely asking informed consent the first time resources need to be accessed (e.g. accessing Bluetooth or location tracking), coarse grained permission handling can be added. Alternatively, consent can be given at application installation time, and each time IoT device modifications occur in the physical world. The *Mobile Application* is run by a caregiver and typically needs access to multiple sensors and actuators owned by elderly E. Hence, asking consent to caregivers is not appropriate. Instead, elderly can rely on a web interface in which they can give consent to *Application Instances* running on behalf of certain caregivers to access a (sub)set of IoT *Devices*. The rights are delegated to the *Application Instances* of the aforementioned caregivers.

Note that in the setting presented before, 3220 IoT *Devices* are rolled out in the physical world. Each *HMI Application Instance* will ask permission to at most 32 IoT *Devices*, namely 3 attached to the elderly (i.e., 1 heart rate sensor, 1 position sensor and 1 fall detector) and 29 installed in the residence (i.e., 2 temperature sensors and 1 thermostat, 10 lamps and 10 light sensors, and 6 cameras). Each *Mobile Application Instance* in our setting requests access to 141 IoT *Devices*, namely the caregiver's position sensor and 27 *Devices* per elderly under his/her supervision. The *Devices* are 1 heart rate sensor, 1 fall detector and 26 *Devices* installed in the residence of the elderly (i.e., 10 lamps and 10 light sensors, and 6 cameras). The sets can be derived by query Q_4.

4.4.4 Device Loading and Addressing Devices

Device loading is the process of retrieving the access and authentication data required to successfully connect to an IoT sensor or actuator, and subsequently perform operations on that *Device*. The data typically consists of technology and communication type info, addresses and credentials. Some *Devices* are directly accessible. Others are accessible only indirectly by applications via a gateway of via the cloud. This implies that credentials can be (re)used to get access to multiple sensors and actuators in some settings. This simplifies manageability at the cost of security. The data can be gathered either from an external server, a local gateway, user input or a combination thereof. Generally, device loading strategies can be classified according to two categories, namely pre-loading and on-demand loading. Pre-loading implies that access and authentication data is retrieved in the application before the sensor or actuator is actually accessed by that application. This can be done at start-up or at regular time intervals, which facilitates manageability at the cost of security. Malware getting access to local application data can possibly steal credentials that are never used by that application. On-demand loading only retrieves access and authentication data when the application wants to connect to a particular device. This often imposes stronger connectivity requirements between credential issuers and end-user applications but leads to improved security by applying the principle-of-least-privilege. Credential information can eventually be erased automatically when no longer needed by the application. Pre-loading can be supported by applying query Q_4 with two parameters, namely the *Application Instance* App_X and the *Application Policies* P_{App}. This query returns all devices that will possibly be connected to App_X in the future, given the current IoT environment (i.e., assets, infrastructure and relations between them). On-demand loading can be supported by applying the same query Q_4 with a third parameter, namely a condition $cond$ defining the current context. Note that the set of devices returned by the second query is a subset of the devices returned by the first one.

Addressing devices can only occur after device loading. User input and *Asset Parameter States* may trigger sensing and actuation. The triggers as well as required implications on device operations can be derived from the *Application Policies*. Although operations on an IoT device can be invoked as soon as it is loaded, it is recommended to sense only if the predefined conditions are fulfilled, both from privacy perspective (i.e., only retrieve sensor values when necessary) and performance perspective (i.e., minimizing battery drainage).

4.5 Discussion

Our methodology is driven by the advanced reconfigurability needs of companies developing sustainable IoT software applications in one of more verticals. Although the AAL domain is used to demonstrate the feasibility, the design tactics can equally be applied to other verticals like companies building advanced fleet management software and smart factory ecosystems. Different customers within one vertical have to manage distinct *Assets*, may prefer slightly deviating *Application Policies*, and/or select alternative infrastructural elements over time. Cost-efficient reconfigurability is a key concern.

Devices can be coupled to other *Assets* explicitly. For instance, a heart rate sensor can be transferred to another individual. Recoupling can also occur implicitly by redefining relations between *Assets*. For instance, connecting a trailer to given truck automatically re-couples all sensors attached to that trailer to the truck's on-board *Application Instance*. Adding *Assets* to an IoT ecosystem or removing them is straightforward but restricted by the environmental model. For instance, in our proposed AAL model, an elderly should always be coupled to a residence. Also, technology-agnostic policies can be defined per *Application Instance*. These policies can easily be tuned to embrace specific customer demands without huge engineering efforts. Many existing approaches group IoT devices and define actions that must be taken when sensed values of devices within a certain group exceed pre-configured values. Our policy language allows to express more complex conditions in terms of *Asset States*. The *Asset States* are sensor/actuator-agnostic although often mapped obviously to an IoT *Device* sensing or actuation operation. The `heat` *State* of a `residence` *Asset*, for example, can be inspected by the `temperature` *Parameter* of one or more `TemperatureSensors`. However, inspecting *Asset Parameters* can be mapped to more complex sensing operations in which multiple *Device Types* are involved. Inspecting the `healthiness` of an elderly relies on heart rate sensor and fall detector values. Similarly, inspecting the residence's `clarity` can be done by fusing (i.e., averaging or weighting) values returned by multiple light sensors. This implies that policies might change if the environmental model changes. Although this occurs rarely, it can be triggered by specific customer demands. Section 4.4 already mentioned the impact of cardinality changes on the *Application Policies*. In the aforementioned examples, we always applied cardinalities greater than zero. However, the 0..N cardinality range is often applied in practice to support lacking or broken IoT devices. This implies that no sensor values can be propagated to the *Asset Layer*. *Assets* are often extended with an `undefined` *State* to tackle sensor absence or failure.

Scalability is another key concern. Selective *Device* loading, sensing and actuation is an important strategy that may also positively impact security and privacy. Table 4.8 and 4.9 can be generated automatically by the *IoT Ecosystem Manager* for various

Application Policies, and shows the subset of *Assets* and *Devices* that need to be loaded in each *Application Instance* if certain *Asset States* are fulfilled. The first rows in each table shows the total number of *Assets* (i.e., 220) and *Devices* (i.e., 3220), respectively, that are managed in the sample AAL IoT ecosystem. We see that the number of *Assets* and *Devices* that need to be loaded in both versions of the *HMI Application* never exceeds 2 and 32, respectively, which is for both less than one percent. The maximum number of *Assets* and *Devices* that need to be loaded by the *Mobile Application* is 11 (i.e., 5 percent) and 141 (i.e., 4.09 percent) respectively. These numbers depend on the number of elderly that are allocated to each caregiver.

	Assets				
	C	R	E	Total	%
AAL IoT ecosystem	**20**	**100**	**100**	**220**	**100**
HMI$_{v1.0}$	**0**	**1**	**1**	**2**	**0.91**
presence[E_P, R] = absent ∧ healthiness[E_P] = normal	0	1	1	2	**0.91**
presence[E_P, R] = absent ∧ heat[R]= normal	0	1	1	2	**0.91**
presence[E_P, R] = absent ∧ heat[R]= cold‖hot	0	1	1	2	**0.91**
presence[E_P, R] = present ∧ healthiness[E_P] = normal	0	1	1	2	**0.91**
presence[E_P, R] = present ∧ healthiness[E_P] = alarming	0	1	1	2	**0.91**
HMI$_{v2.0}$	**0**	**1**	**1**	**2**	**0.91**
presence[E_P, R] = absent	0	1	1	2	**0.91**
presence[E_P, R] = present	0	1	1	2	**0.91**
Mobile	**1**	**5**	**5**	**11**	**5.00**
presence[C_P,R]=absent ∧ ∀i healthiness[E_i]=normal	1	0	5	6	**2.73**
presence[C_P,R]=absent ∧ ∃!i healthiness[E_i]=alarming	1	1	5	7	**3.18**
presence[C_P,R]=present ∧ ∀i healthiness[E_i]=normal	1	0	5	6	**2.73**
presence[C_P,R]=present ∧ ∃!i healthiness[E_i]=alarming	1	1	5	7	**3.18**

Table 4.8: Overview of the number of *Assets* that should be loaded minimally in *HMI* and *Mobile Application Instances* under certain conditions to adhere to the *Application Policies* defined in Table 4.7 in the sample AAL IoT ecosystem

	Infrastructure									
	La	Ls	Ca	Tm	Ts	Hr	Fd	Ps	Total	%
AAL IoT ecosystem	**1000**	**1000**	**600**	**200**	**100**	**100**	**100**	**120**	**3220**	**100**
HMI$_{v1.0}$	**10**	**10**	**6**	**2**	**1**	**1**	**1**	**1**	**32**	**0.99**
presence[E_P, R] = absent $\wedge$ healthiness[E_P] = normal	0	10	0	2	1	0	0	1	14	**0.43**
presence[E_P, R] = absent $\wedge$ heat[R]= normal	0	10	6	2	0	0	0	1	19	**0.59**
presence[E_P, R] = absent $\wedge$ heat[R]= cold$\|\|$hot	0	10	6	2	1	0	0	1	20	**0.62**
presence[E_P, R] = present $\wedge$ healthiness[E_P] = normal	10	10	0	2	1	1	1	1	26	**0.81**
presence[E_P, R] = present $\wedge$ healthiness[E_P] = alarming	10	10	6	2	1	1	1	1	32	**0.99**
HMI$_{v2.0}$	**10**	**10**	**6**	**2**	**1**	**1**	**1**	**1**	**32**	**0.99**
presence[E_P, R] = absent	10	10	6	2	1	0	0	1	30	**0.93**
presence[E_P, R] = present	10	10	6	2	1	1	1	1	32	**0.99**
Mobile	**50**	**50**	**30**	**0**	**0**	**5**	**5**	**1**	**141**	**4.38**
presence[C_P,R]=absent $\wedge$ $\forall i$ healthiness[E_i]=normal	0	0	0	0	0	5	5	1	11	**0.34**
presence[C_P,R]=absent $\wedge$ $\exists! i$ healthiness[E_i]=alarming	0	0	6	0	0	5	5	1	17	**0.53**
presence[C_P,R]=present $\wedge$ $\forall i$ healthiness[E_i]=normal	10	10	0	0	0	5	5	1	31	**0.96**
presence[C_P,R]=present $\wedge$ $\exists! i$ healthiness[E_i]=alarming	10	10	6	0	0	5	5	1	37	**1.15**

Table 4.9: Overview of the number of IoT devices that should be loaded minimally in the *HMI* and *Mobile Application Instances* under certain conditions to adhere to the *Application Policies* defined in Table 4.7 in the sample AAL IoT ecosystem

The current implementation relies on a centralized server for *Asset* and *Device* management. This is, however, no longer a feasible solution in open edge oriented IoT ecosystems. Distributed *Asset* and infrastructure management leads to improved security, privacy and availability properties. Local gateways manage access and authentication information of locally installed sensors and actuators which are be obtained by end-user applications after authentication and authorization. Environmental and infrastructural changes are supported in the current prototype. On applications start, *Asset* and infrastructure updates are loaded in the application. Besides implicit updates, both the *Mobile Application* and *HMI Application* interfaces contain a button to instantly reload the more recent view on the IoT environment. A request can be sent from the *HMI Application* to the management server to

retrieve the most recent updates. Alternatively, updates are pushed immediately to apps. Finally, we assume that the devices hosting the *Mobile Application* and *HMI Application* are subject to Mobile Device Management (MDM) which allows the commercial health provider to control the integrity of the applications. This ensures that the specified *Application Policies* are enforced correctly.

4.6 Conclusion

In this chapter, we focused on design guidelines for application centric development of IoT ecosystems. Keeping in mind the architecture presented in Chapter 3, a distinction is made between the design and operational perspectives of IoT ecosystems. The design perspective refers to the modeling concepts, the operational perspective to an instance of the model in specific settings. A structured approach helps us to build a complete model of the ecosystem that is technology agnostic and considers scalability and reconfigurability. Based on the design, applications can be developed at a high level manner using policies. Moreover, the approach facilitates the redefinition of behavior of already existing applications and modifications in the underlying infrastructure. Finally, the structured design also simplifies management of the infrastructure and applications. Thus, not only the task of developers, but also that of maintenance personnel is simplified by designing the ecosystem in a correct and structured manner.

Chapter 5

Creating Advanced IoT Applications

The content from this chapter is previously published in:

- I. Bohé et al. "Towards low-effort development of advanced IoT applications". In: *Proceedings of the 8th International Workshop on Middleware and Applications for the Internet of Things (M4IoT)*. Québec, Canada - Online, 2021, pp. 1–7
- I. Bohé et al. "A Logic Programming Approach to Incorporate Access Control in the Internet of Things". Accepted at IFIP IoT 2022. Amsterdam, the Netherlands

Despite the improvements brought by Internet of Things (IoT) frameworks and modeling approaches, as the ones proposed in Chapter 3 and Chapter 4, respectively, not all needs in IoT environments can be covered. Handling runtime dynamics, such as device connectivity and dynamically fulfilling Quality of Service (QoS) requirements, such as latency and energy usage, is usually not supported. Because not every application has the same functional requirements, simply adjusting the set of supported devices is not sufficient. It is necessary to look at how different functionalities (i.e., access control, connectivity management and automation) can be included or adapted easily. On top of that, seamless collaboration between these different functionalities is of great importance for the proper functioning of the applications. Logic programming languages are a great way to incorporate such reasoning capabilities in the middleware.

Furthermore, it is clear that data is playing an important role in all kinds of systems, including IoT systems [52]. Data can provide a lot of information about the system and help to achieve different goals, i.e., listing available assets and determining the root cause of a problem in the system. This, of course, on the condition that the data can be easily retrieved from the system.

Logic programming languages have already been proven to be useful to retrieve information (i.e., facts) from databases. Based on specified rules, data can be deducted together with the available data in the database [35]. Moreover, the querying abilities of logic programming languages make it possible to retrieve the desired information without the need to define the type of information that can be requested beforehand. In other words, when designing the system, the developer must not have knowledge about the possible queries.

Unfortunately, the many examples of data breaches [16, 34] and the loss of privacy represent a permanent threat. Several access control strategies have been developed to help in securing the data. Although access control may seem conceptually straightforward, its integration is often complex and error-prone. Over the years, research on access control that harnesses logic is substantial: it has been used to formally verify security properties; to explain, express and enforce access control policies. While knowledge bases may house huge amounts of data and knowledge, research on the use of access control within knowledge representation and reasoning systems is very limited. As former research shows, logic easily lends itself in expressing and enforcing access control policies. However, no structural approach is available to enforce access control inside logic programs.

Contributions The contributions in this chapter are two fold.

First, we propose a reusable **IoT reasoning middleware** that supports application developers in building dynamic IoT applications. It can be used on top of existing IoT frameworks providing generic access to IoT devices, such as the one described in Chapter 3. It comprises an event-based architecture running a logic reasoner in the background. It hosts a number of IoT modules specially devised to handle different functionalities, such as the handling of contextual changes, managing and enforcing access control and supporting a full featured automation engine.

The logic reasoner is implemented using Prolog [81] and is the basis to add more advanced intelligence to the IoT applications. It allows applications to automatically infer knowledge and provides querying capabilities to gain insights in the IoT system. Tasks such as root cause analysis can thus be performed based on real-time data extracted from the system. Retrieving information from the systems state, such as a list of all active devices in a certain room is trivial and requires no additional programming.

To demonstrate its feasibility, the reasoning middleware has been integrated in a server and mobile application. For the demonstrator, the middleware was built on top of the JavaScript port of the SMIoT framework described in Chapter 3. Nevertheless, other frameworks could be used as well.

Second, we presents a solution that evaluates access control policies during resolution in logic programs, taking special care for impure predicates, (e.g., actions in the IoT space). It provides a high expressiveness and fine-grained control of the program and makes it a widely applicable approach. A *deny as soon as possible* strategy is used, but decisions are postponed until they can be decided with certainty. Moreover, as enforcement occurs during inference, the approach easily extends to the dynamic case such as a reactive system (i.e., one that responds to external inputs). In this approach, access rules are defined as part of the program logic. In other words, the rules can take advantage of the program's knowledge base. Hence, expressing access control strategies, such as resource based, role based and relationship based access control, is straightforward.

To validate and demonstrate the approach, an implementation is provided as a Prolog meta-interpreter, named ACoP. It can easily be integrated in existing Prolog programs with minimal effort. Overhead is limited to defining the access rules, also in Prolog.

The remainder of this chapter is structured as followed. In Section 5.1 we describe the complete construction and operation of the reasoning middleware. We also look at how this can be implemented. In Section 5.2 incorporation of access control in the middleware and more generally in logic programming languages is described. Here, too, we go into more detail about the implementation. Both sections end with a discussion.

5.1 Reasoning Middleware

Although many middleware frameworks, such as the ones described in Section 2.3, make the applications 'smart' by adding automation functionality, almost none of them take care of the dynamic nature of IoT ecosystems. The reasoning middleware proposed in this section supports application developers in coping with the dynamic nature of IoT systems and is designed with device availability and reliability in mind. It allows to specify policies based on, for instance, connectivity and reliability, and enforces them on-the-fly.

Automation in IoT systems is widely available. Reasoning in these systems, however, is often static, limited to very basic event-condition-action rule handling and tightly

coupled to the framework (e.g., Home Assistant). To provide more advanced reasoning capabilities, recent work suggests the use of more complex symbolic reasoners [17, 21, 22, 65, 84]. Calegari et al. propose LPAAS, a logic programming REST-based service for IoT Systems [21]. Machado et al. propose hybrid reasoning based on compositional rules selecting the reasoner based on the context[65]. dos Reis et al. present a semantic model and an IoT middleware service for data stream reasoning[84]. The stream of events is enriched with semantic meaning.

Most of the reasoners above start from the use of ontologies to provide higher level reasoning. Although our reasoning middleware may be extended to take advantage of these ontologies, it is seen as a possible extension and not focused on in this work.

The logic reasoner is not only involved in the extraction of knowledge. It is also the backbone of our middleware, controlling event flow, dynamic processing and reasoning. In other words, while most solutions are imperative programs that interact with a reasoner for symbolic reasoning, our middleware is mainly a logic program, that interfaces with standard (imperative) libraries when appropriate.

More importantly, taking advantage of the reasoning engine, our middleware provides a number of building blocks, called IoT modules, that greatly simplify IoT application development: an advanced automation module, a connection manager, a module for access control and a module to manage assets.

5.1.1 Architecture of the Reasoning Middleware

The architecture consists of an *event-based logic reasoning engine*, hosting multiple *IoT modules*. Figure 5.1 presents the architecture of the reasoning middleware. Adhering to the principle of separation of concerns, each *module* implements a particular functionality. Immutable *event* messages are used as a messaging mechanism between modules and for communication with the external environment. Using an event bus, each module subscribes for specific incoming event messages, process them and may create new event messages as a result. The middleware is developed in Prolog and the IoT modules are implemented as actual Prolog modules. Each module provides an interface so that communication with it can take place. Modules have access to a knowledge base containing general information such as the environment state, and module specific rules and policies.

The configuration of the middleware defines the modules to be loaded. To communicate with the application or the underlying IoT framework, an application and device module are available. By default, new event messages are submitted to the event bus, and, based on the configurations of the middleware, IoT modules subscribe for event messages on that bus. The event bus allows multiple modules to listen for the same events. For instance, both the automation module and

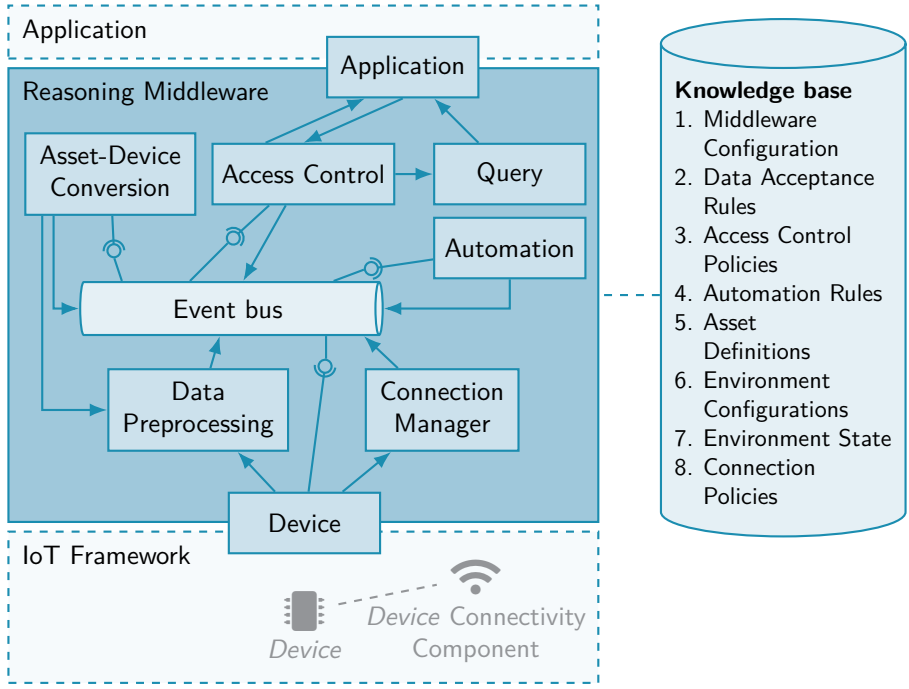


Figure 5.1: Structure of the reasoning middleware

asset-device conversion module (explained in Section 5.1.1.2) may be interested in event messages containing data from an IoT device (i.e., update events). While subscribing, each module passes a filter to the bus specifying the event messages they want to receive.

The middleware builds upon the concept of assets, which represent items of interest in the physical world (e.g., a room, a production line and a person). By using the concept of assets, it is possible to develop the application independent from the underlying IoT infrastructure, as described in Chapter 3.

To provide more control and flexibility, the application designer may choose to configure a more hybrid architecture, combing the event-based with a flow-based approach. For instance, to limit the event messages on the event bus (i.e., access control module) or to split related functionality into reusable sub-modules. Through this mechanism, IoT modules are easily added, replaced or adapted by modifying the configuration of the middleware.

As shown in Figure 5.2, the middleware distinguishes itself from other IoT frameworks

in that it can be used on a multitude of platforms, going from IoT devices, mobile platforms up to even cloud platforms. Although applications on each of these platform have their own requirements, the framework provides features that are useful for all of them.

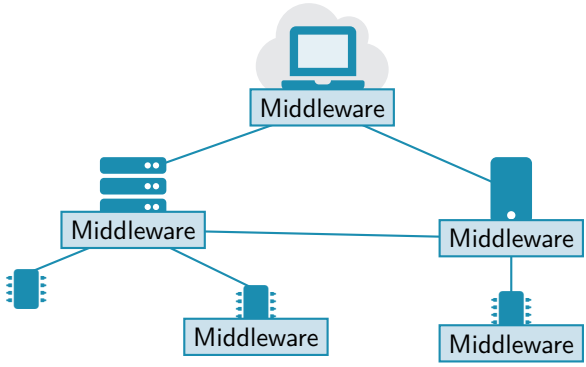


Figure 5.2: A multi-platform IoT reasoning middleware for device, mobile and cloud platforms

Moreover, due to the high number of IoT devices, fog computing, as described in Chapter 2 is getting traction. This moves processing and intelligence from the cloud towards the edge. Thus, support for effective reasoning and decision making becomes required in all parts of the IoT ecosystem.

The main building blocks (i.e., events and IoT modules), will now be presented in more detail.

5.1.1.1 Events

Events denote changes in the system. In the event-driven architecture, event messages (henceforth called *events*) are used to notify these changes. In the middleware, subscriptions are set up to register for event streams. For instance, a module may subscribe to retrieve events asynchronously from a specific IoT device, such as measurements or changes in the connectivity of the device. Likewise, an application on top of the middleware creates a subscription to the middleware to receive changes related to a certain asset or device managed by the middleware. Note that the events associated with an asset (e.g., clarity change in a room) are most-likely triggered based on events from its underlying devices (e.g., value change in a light sensor). To control the application, next to these change based events, the application also uses events to trigger certain functionalities in the middleware (i.e., action and query events).

Several types of events are defined and described below. Events are defined by their properties and consist of the following:

- `type`, either update, action, query or query-result (explained below).
- `id`, to uniquely identify the event.
- `creation-time`.
- `creator` of the event, being either the device that generated the event, the user that sent the request, other application components or the middleware itself.
- `origin-event` *(optional)* , specifying the `id` of the event that triggered the event (e.g., the `id` of the subscription request in case of an update event).
- `subject` *(optional)*, describing the entity to which the event information is related. For instance, when new sensor data is received from a sensor, the subject identifies the sensor. In the case of an action-event, the subject indicates the entity on which the action must be executed.

Below, the different events are discussed in more detail, namely **update events**, **action events** and **query events**.

Update events As IoT environments are highly dynamic, it is important to inform the application of any change in the environment. Not only the monitoring of sensor data is of interest, also the connectivity and reachability status of a device is useful information. The type of information being updated is specified in an additional `update-property` field. This simplifies filtering on these particular types of update events. The currently supported property types are *parameter*, *connectivity* and *reachability*. Upon a change of a sensor value, a *parameter* update event is generated. The event is triggered when an IoT device presents a new value for one or more of its parameters. In many IoT environments, the *reachability* of IoT devices changes frequently. *Reachability* events are triggered to inform the application when devices come in-range or go out-of-range. Likewise, the connectivity state of IoT devices may change when devices connect or disconnect. *Connectivity* update events allow the application to properly address such changes.

Parameter, *connectivity* and *reachability* related updates are discussed below:

Parameter Upon a change of device or asset parameters (described in Chapter 4), a parameter-update event is generated. The event is triggered when an IoT device presents a new value for one or more of its parameters. For instance, after a subscription to sensor `temp_sensor_ID1`, the sensor submits a temperature value of 24.8°C and humidity value of 45.4%. Besides variable parameters, static parameters only change infrequently, due to explicit modifications in the physical environment. Examples can be a change in the name or polling time of a sensor. An example parameter-update event can be seen in Listing 5.1

```
{
  type: 'update',
  id: '2cd3ae3f-100d-4cbd-...',
  creation-time: 2020/01/01 16:40:00,
  origin-event: 'f198ae08-be4d-4...',
  creator: 'temp_sensor_ID1',
  subject: 'temp_sensor_ID1',
  update-property: 'parameter',
  data:
  [
  {
    parameter: 'temp_1',
    value: 24.8
  },
  {
    parameter: 'hum_1',
    value: 45.4
  }
  ]
}
```

Listing 5.1: Example parameter update event in the reasoning middleware

Connectivity In many IoT environments, the connectivity of IoT devices changes frequently. Connectivity typed update events allow the application to properly address connectivity changes. The IoT framework is responsible for monitoring the IoT devices and submits connectivity update events to the middleware. A device can either be connected or disconnected from the application.

Reachability Similar to the connectivity-update events, the device reachability state may change often. These events are triggered when devices come in-range or go out-of-range. Based on these events decisions are made to manage device connections.

Action Events Action events are events created to trigger specific functionality. They can be initiated both by the application and the modules defined in the middleware. To support the use of a heterogeneous set of devices, actions should be performed on assets instead of specific devices. For instance, when a user clicks a button the application creates an action event to turn the light on in the dining room, and submits it to the middleware. The asset-device conversion module (see Section 5.1.1.2) listens to events for the dining room asset and transforms them into new action events for the specific lamp configured for this asset. Subsequently, the new action event is submitted to the underlying IoT framework. An example of both action events can be seen in Listing 5.2

Query & Query-Result Events As mentioned before, the middleware uses a Prolog reasoner. This reasoner takes care of the channeling of messages, filtering of messages and rule handling. A strong property of Prolog, however, is its inductive

```
{
  type: 'action',
  event-id: '0a80f883-556a-4489-...',
  creation-time: 2020/01/01 16:40:00,
  creator: 'john',
  subject: 'dining_room',
  data:
    {
     parameter: 'lightning',
     value: 1
    }
}

{
  type: 'action',
  event-id: '5ddb9c75-c674-43bf-...',
  origin-id: '0a80f883-556a-4489...',
  creation-time: 2020/01/01 16:40:01,
  creator: 'asset-device-conversion',
  subject: 'lamp_ID2',
  data:
    {
     parameter: 'on-off-status',
     value: 1
    }

}
```

Listing 5.2: Example action events in the reasoning middleware

reasoning. It allows us to query the knowledge base (including the policies and the current state of the engine) and infer new conclusions. This does not require any additional coding effort from the developer and allows a flexible and database-like interaction with the IoT devices. Applications may use queries to obtain information from the IoT network. Such queries may be very basic, retrieving the value of a certain sensor, but also quite complex. As an example, Listing 5.3 shows a Prolog query request to obtain the production line in which machines are in alarm, and the corresponding machines in alarm.

```
?- prod_line(Line),
    location(Machine, Line),
    parameter_value(Machine, alarm, on).
```

Listing 5.3: Prolog query to obtain production lines and machines in alarm

Integrating the Prolog query into a query event gives the event described in Listing 5.4.

```
{
  type: 'query',
  event-id: '5ddb9c75-c674-43bf-...',
  creation-time: 2020/01/01 16:40:01,
  creator: 'app',
  query: 'prod_line(Line),
    location(Machine, Line),
    parameter_value(Machine, alarm, on).',
  return:['Line', 'Machine']
}
```

Listing 5.4: Example query event to obtain production lines and machines in alarm

The `return` property defines which information should be returned. In this case, all production lines (i.e., `Line`) and machines (i.e., `Machine`) in alarm are requested. Since multiple solutions are possible, they are collected and returned in a single query-result event of which an example can be seen in Listing 5.5

```
{
  type: 'query-result',
  event-id: '54a451b5-0f79-41bc-...'
  origin-event: '5ddb9c75-c674-43bf-...',
  creation-time: 2020/01/01 16:40:02,
  creator: 'engine',
  query-result: [ ['line_1','machine_1'], ['line_1','machine_2'],
  ↪  ['line_3','machine_7']]
}
```

Listing 5.5: Example query-result event with production lines and machines in alarm

5.1.1.2 IoT Modules

IoT modules, implemented in Prolog, are the workhorses of the middleware. Each module either filters existing events, or creates new events as a result of its functionality. Filtering happens, for instance, by the access control module which only forwards user requests if they are allowed according to the predefined policies. The automation module, on the other hand, may create an action event to switch on the heating.

Events are asynchronous and come in unadvertised. However, some events require a certain response to be returned to the client application. To prevent undefined behavior, the middleware contains logic to detect cases where no response is generated (i.e., a timeout-event is raised if a query event does not result in a response event within a reasonable amount of time).

The core of the middleware is implemented in Prolog. It provides the routing of events to the correct modules, implements the main IoT modules and keeps a knowledge base on which reasoning can be done. The IoT modules are defined as Prolog modules with a common interface. The interface with the core is specified through a set of predefined predicates that each module provides, namely `init/0` to initialize the module (where `init` is the name of the predicate and `0` the number of variables) and `handle/1` to allow the middleware to send events to the module for processing.

For several operations such as automation and policies, a logic rule-based approach is preferable. Nevertheless, the middleware easily adapts to support script based modules. These modules are better suited for more computationally intensive operations, such as aggregation of data or analytic processing.

To help IoT application developers in building applications, the middleware provides a number of basic modules, each responsible for tackling specific challenges:

- connection manager
- access control
- data preprocessing
- asset-device conversion
- automation

Each of these modules will now be discussed.

Connection Manager The *Connection Manager* is responsible for the creation and tear-down of communication channels with IoT devices. Connection policies define rules on the type of channels that are set up with devices, potentially depending on other environmental factors. For instance, the interval of polling based subscriptions may be increased in case of a low battery level of the device.

When a subscription for parameter updates is requested, the connection manager takes care of the communication channel but also makes a subscription for state change events. When connection policies state to automatically recover a connection, appropriate actions are taken as soon as the device comes in range. On the other hand, connection policies may specify to only search for devices when certain constraints are satisfied (e.g., only try to connect to certain devices when connected to a specific network). The connection manager module monitors device connectivity through the reachability and connectivity update events.

To determine the most suitable action to take, rules, which are stored in the knowledge base, specify the constraints and requirements to manage connections.

The connection manager executes these rules and enforces related connection policies.

In addition to the more general connection control, the module also offers a functionality called *automated source selection*. Several IoT solutions provide multiple channels to collect data from a device. Sometimes, data can be collected both directly from the device, and via a cloud platform. Applications may benefit when both channels can be used. If a device is in range, the data is retrieved directly, and when no direct connection is available or the application needs data store in the cloud, the application connects to the cloud, despite an increase in latency or lower accuracy. This module allows to switch flawlessly between both, without intervention of the application developer or the user.

Access Control In the *Access Control* module policies define who has access to what information in the knowledge base and under what conditions. The module supports both positive (`allow`) and negative (`deny`) assertions.

The access control module can in this way filter action events originating from the application. The module consults the access control policies to determine if an action may be executed. In case access is granted, it is forwarded to the event bus. Otherwise, the module blocks the event. In the same way update, queries and subscription requests can be filtered.

This approach allows for fine grained access control. Access can be limited to specific parameter actions (e.g., monitor, read, write) on a device or asset, based on contextual information, or even depending the current value of other device parameters.

A full elaboration on how this is implemented in Prolog can be found in Section 5.2.

Data Preprocessing Parameter update events enter the system frequently. The pace at which these updates come in, is not always under the control of the middleware. When the size of the stream is too large, it could lead to unnecessary or excessive processing. Depending on the data acceptance rules (e.g., to limit processing in case of a low battery) and the requirements of the applications, events are filtered to limit the number of events exposed to the middleware. For instance, firing an update event may only be worthwhile when the value of the data changed sufficiently or when it crossed a certain value.

The following types of filters are available:

- *pass*: all events are forwarded.
- *value change*: the value must change.
- *absolute difference*: the value change must be more than a fixed value.

- *relative difference*: the value change must be more than a value relative to the range of the parameter.
- *time difference*: the time difference between two subsequent events must be more than a fixed value.
- *time delay*: an update is delayed for a predefined amount of time.

The latter is a special type of filter, requiring a timer to wait for a predefined time interval. If no new value arrives in the meanwhile, the value is accepted. In Table 5.1 the conditions are formalized. By default, the *value change* filter is used. Filters can be created to support more complex situations. For instance, the relative difference filter can be combined with the delay filter, to prevent short bursts of changes to be presented to the other modules or the application.

Filter Type	Condition
pass	$true$
value change	$x_{new} \neq x_{old}$
absolute difference	$\lvert x_{new} - x_{old} \rvert \geq x_{\Delta}$
relative difference	$\lvert x_{new} - x_{old} \rvert \geq p \lvert x_{max} - x_{min} \rvert$
time difference	$\lvert t_{new} - t_{old} \rvert \geq t_{\Delta}$
time delay	no new value for a certain time t_{Δ}

Table 5.1: Formalized conditions of filter types in the data preprocessing module

Asset-Device Conversion The *Asset-Device Conversion* module is in charge of the conversion of device related events into asset related events, and vice versa. When modeling the IoT ecosystem based on the principles described in Chapter 4, the asset definition module is used to translate device data into asset data. By defining domain, device and state definitions and the binding between them, incoming events are easily translated. The most basic integration of an asset (e.g., living room) simply converts events coming from its underlying devices (e.g., light and temperature sensor) into new events for the new resource and corresponding parameter definitions. Action events performed on the asset parameters are converted into action events on specific parameters of a device and forwarded to the device.

More advanced cases may consider to transform sensor data into a different representation, e.g., from a temperature of 40°C to the state value `hot`, or include specialized functionality that may be provided by the asset (e.g., to create a certain mood in the room).

It is advised for other modules to define their functionality based on these assets. This ensures that it is not necessary to update the policies and rules that depend on the underlying devices in case a device is replaced. In fact, every device in the

system can be converted into a 'basic' asset, and higher level assets are then defined based on those 'basic' assets. Replacing a device only requires to remap the new device with the existing asset.

Automation As one of the most basic functionalities in applications, this module supports automation. Basic Event-Condition-Action (ECA) rules are easily defined using Prolog logic. Although in general, automation rules are triggered by events coming from the underlying framework (i.e., update events), this module takes advantage of the event-based architecture, and can listen for any event exposed to the module, including user actions and events created by internal modules.

In automation, some actions should be triggered at specific moments in time. Therefore, the module includes functions to register for timer events (at absolute dates/times or by interval). Automation rules sometimes imply that a device is monitored. For example, switching on the light in the room when it gets dark implies that you need to monitor the clarity status of the room. In this case (based on the asset state definition) the light sensor must be monitored. The automation module takes care of this and will request a subscription for the asset or device parameter to be sensed. Usually, this module outputs action events to perform specific actions either in the middleware or on the peripherals.

5.1.2 Implementation of the Reasoning Middleware

The source code of the middleware with basic integrations of the IoT modules is available at `https://github.com/ilse-bohe/iot-reasoning-middleware`

The middleware has been integrated in a combination of Javascript and Prolog, and build on top of the JavaScript framework proposed in Chapter 3. This allows for an easy integration of the demonstrator in both server (e.g., using Node.js) and mobile applications. The middleware's logic reasoning capabilities are provided using Tau-Prolog [103], a lightweight, open source Prolog interpreter developed in JavaScript.

The main Prolog program in the middleware *(main.pl)* is responsible for the routing of events between the different modules in the reasoning engine. Connections define how the main program needs to route messages. A custom interface *(connector.js)* is developed to support communication between the Prolog and outer environment. This allows events to be sent asynchronously from the application or device layer to the reasoning engine and vice versa.

Performance Measurements. The middleware is tested on a basic manufacturing use case consisting of multiple *production lines* with *machines*. A production line has a *state* which can be set to start or stop all machines in the line. It also has an *alarm* state which is active when at least one of the machines in the production

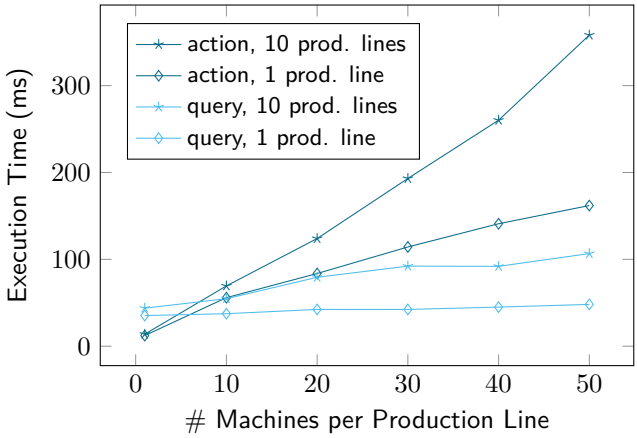


Figure 5.3: Execution time for handling events in the reasoning middleware

line is in alarm. For the performance tests we measure the time needed to process an action and a query event sent from the application to the middleware.

The *action event* starts the production line. The middleware translates the action into starting all machines in that line. In Tau-Prolog the order of facts in the knowledge base is important. For the tests with 10 production lines, we measured the time to activate the last production line found in the knowledge base, giving us the worst case execution time.

The *query event* is the event introduced in Section 5.1.1.1, and requests the production lines and machines in alarm. The last machine in the last production line is in alarm, giving us the worst case execution time for querying the lines and machines in alarm.

Figure 5.3 shows the average time (in ms) to execute the events with respect to the number of machines per production line. As can be seen, for action events, the time increases linear with the number of assets. For an environment with one production line it ranges from about 12 ms up to 162 ms for 1 machine and 50 machines, respectively. In case of ten production lines this ranges from 14 ms to 358 ms respectively. Adding additional assets to the knowledge base (no production lines nor machines) has no direct impact on the execution time.

The time difference for answering a query event is less affected by the number of machines in the production line. Only 1 event needs to be created for answering a query event, independent of the number of machines in alarm, while for the action event, a new event is created for each machine in the production line (i.e., up to 50 events).

The demonstrator has been developed using Tau-Prolog. This made it easy to create a cross-platform demo for both a server and mobile application. The Tau-Prolog reasoner can be used in small, non-time-critical, settings. For performance, support or timing reasons, more established reasoners such as SWI-Prolog [111] can be preferred. Tests with the reasoning engine using SWI-Prolog shows a clear performance gain. As an example, executing the action event in an environment with 10.000 production lines with a single machine only takes 15 ms, and takes 68 ms with 10 production lines and 10.000 machines each. This is due to several optimizations integrated in SWI-Prolog.

5.1.3 Discussion

The main goal of the middleware introduced in this paper, is to support application developers in building responsive, easy to maintain, and smart IoT applications. Several modules have been proposed and specified, each taking care of certain functionality useful in such environments. As discussed in related work, many reasoning applications for IoT try to tackle the use of ontologies to enhance reasoning capabilities. Although ontologies may indeed have interesting applications, this paper demonstrates that even without adding these additional complexities, including a logic reasoner in the middleware already adds important benefits to IoT applications. An example is the querying capability, allowing application developers to query the IoT system as if it were a database. While often very simple queries may be sufficient, more complex ones may, for instance, support the integration of root cause analysis in the IoT application.

Last but not least, the event based reasoning system was integrated using Prolog as a backbone. While often Prolog is used as a service only for very specific reasoning tasks, this papers shows that integrating a full featured IoT middleware with complex reasoning in Prolog is a valuable alternative. It shows it is a viable solution for many IoT applications, especially with the current drive towards edge intelligence.

Future work. The use of Prolog also provides a number of other advantages. When defining rules and policies, it is important that they are not in conflict with one another. Although conflicts can be avoided by carefully drawing them up, a *conflict detection module* as discussed in [5] could be integrated. Furthermore, despite being very flexible, integrating fine-grained access control on query events remains a complex endeavor. Extending the access control module with automated verification of access control rules, and support for fine-grained filtering in the case of query events is left for future work.

Currently, the custom configuration of modules, and their policies are written in Prolog. Application users usually have no experience with programming languages,

let alone Prolog. Providing a basic, graphical interface for creating queries and updating policies is desirable.

5.2 Access Control in the Reasoning Middleware

One of the important functionalities within the reasoning middleware is access control. In this section, we will take a closer look at how this access control can be added to the middleware. Moreover, we do not only focus on how access control can be added to the middleware, but how access control can be injected into logic programming languages, without taking into account the surrounding reasoning middleware.

A straightforward approach would be to verify access control policies and remove predicates that do not comply with those policies during consultation of a program. This mimics standard access control to resources as a whole (e.g., a file). In logic programs, however, access control can be much more versatile. Not only access to data or entities can be controlled, but also access to knowledge (i.e., the reasoning itself). Resulting in complex access control logic.

Intuitively, when access is denied, the knowledge should appear as nonexistent, and the user only has a limited view on the knowledge base. In other words, queries requiring inaccessible knowledge for its reasoning, do not return results. Otherwise, they must produce the same results as if no access control were used. In that sense, impure predicates require special care. Impure predicates result in side effects, when the predicate is resolved [100]. For instance, consider the `open(Lock)` predicate that opens the smart door lock, `Lock`. It is impossible to revert the side effects upon backtracking. In the light of access control, it is therefore very important that side effects only occur when allowed.

For many years, logic programming has been used to support access control [2, 59, 87]. Also more recent work takes advantage of formal logic to realize and verify access control models. There are several established access control models, ranging from easy to implement strategies, such as consulting an access control matrix, over Rule Based Access Control (RBAC), to more complex strategies such as Organizational Based Access Control (OrBAC) [28] and Relationship Based Access Control (ReBAC) [42]. Huynh et. al defined an alternative strategy that uses priority, modality and specificity to handle conflicts [54]. The multi-layered access control model was implemented in both ProB and Alloy. ProB is a model checking tool for the B programming language, helping the developer by detecting errors in B specifications [62]. Similarly, Alloy is a language for describing structural properties [56]. Both languages are thus suited for writing complex access rules policies, free of conflicts. The work of Kolovski et. al., provide a formalization of eXtensible Access Control Markup Language (XACML), using description logic

which is the basis for the Web Ontology Language (OWL) [59]. Now, XACML is a widely used and standardized access-control policy language.

In the related work described above, the use of logic programming is limited to either the specification, the design and/or verification of access control policies. The logic programs are merely used as a tool or in the backend of a bigger non-logic-based system. On the contrary, the proposed solution can be used inside logic programs to enforce access control. Provided translation, however, rules written in B, Alloy or XACML can be used by ACoP.

Sartoli et al. use Answer Set Programming (ASP) to implement adaptive access control policies, allowing access control on incomplete policies and imperfect data [92]. This approach is interesting as it can handle exceptional cases and supports dynamic environments where former believes may conflict with new observations. While the policies are specified and handled in ASP, the focus is also in providing support as backend solution towards external systems.

Bruckner et al. present a policy system that allows to compile access control policies in the application logic [20]. An automatically created domain specific language is therefore cross-compiled into the host language. Although the system puts no restrictions on the host language, it is unclear if it transfers to logic programming languages as well.

To the best of our knowledge, ACoP is the first to provide a solution to apply access control to logic programs. Hence, the focus of this section is on how access control can be enforced on the reasoning of logic programs. It not only allows to control access to data or entities, but also to control access to knowledge (e.g., rules).

Support for logic programming languages. Several logic programming languages exist. The proposed solution is validated by a Prolog implementation, and can be integrated into existing Prolog programs. Nevertheless, the approach may be extended to other logic programming languages as well. Examples are Datalog [68], a subset of Prolog, or the more recent Logica [46], a modern logic programming language for data manipulation. However, while in Prolog the meta-interpreter and policies can be fully written in the language itself, writing a meta-interpreter for Datalog and Logica may require more effort and the possibilities to define access control rules will be more restricted.

5.2.1 General Approach of the Access Control Module

Access control is the act of ensuring that a user only has access to what he/she is entitled to. It is usually defined in three levels, using an access control policy, a security model and a security mechanism [87]. The *policy* expresses the rules according to which control must be regulated. The *security model* provides a

formal model of the policy and its working, and the *security mechanism* defines the low level functionality that implements the controls as formally stated in the model. Generally, access control policies are defined in tailored languages such as XACML [36]. They allow to express generic assertions about subjects and the right to perform certain operations. Logic programs, however, naturally support logic-based formulations of access control policies, providing clean foundations and a high expressiveness. In fact, it merges both access control policy and security model, into a single formal specification of the policy. In the remainder, we make no distinction between both, and will use the access control policy to denote both.

In this work, access control policies are defined at the predicate level, by specifying whether access to the predicate is allowed or denied. To ensure completeness (i.e., in case no authorization is specified), a default policy is used. Whether an open (i.e., default access) or a closed policy (i.e., default access denied) is used, is configured at design time by the policy administrator.

Figure 5.4 shows the structure of a target logic program, protected by the ACoP system. Queries sent to the ACoP system are resolved by the access control module implementing the security mechanism. The burden of adding access control to a logic program is very limited. Introducing access control to a target program requires no changes to the program itself. It only requires the definition of the access control policies and the configuration of the the access control module.

In the following sections, the policies and the access control module will be defined.

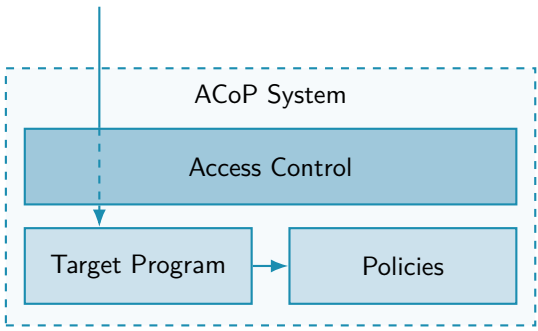


Figure 5.4: Structure of the ACoP access control system

5.2.1.1 Access Control Strategy and Policies

The applied access control strategy in ACoP is the following: *deny access as soon as possible*. Therefore, access must already be verified before the predicate is being resolved, i.e., preliminary access control. If, based on the defined access control policies, it determines that access is denied, resolution will stop. If it cannot

yet determine whether access is denied, an attempt will be made to resolve the predicate. Once the predicate is resolved, access is verified again with the now resolved predicate.

Inaccessible data appears as nonexistent. Thus, the user only has a limited view of the entire knowledge base. Queries to inaccessible data do not return any answers, while queries for accessible data should produce the same results as when no access control is used. Note that this may result in a change of semantics: When no results are retrieved, it may either indicate that no answers to the query exist, or that the user has insufficient rights to access the information. This is to maintain the privacy of the users. It can be compared with login systems, failed login attempts should not indicate whether a user exists in the system or not.

In general, access control policies use a combination of an *Object*, being the resource to which access is requested, and a *Condition*, defining the constraints that need to hold before access is granted. Based on the type of conditions that can be specified, different access control models exist (e.g., attribute-based, role-based, rule-based, discretionary or mandatory access control). For instance, conditions may relate to the subject, the current context and the allowed operations.

In ACoP, objects are represented by predicates, with no restrictions on how conditions are defined. In other words, any access control model can be supported. The object of an access policy is either allowed or denied, depending on customised conditions. Basically, allow and deny access policies, are defined using the syntax shown in Listing 5.6

```
allow(Pred(...)) :- <conditions>.
deny(Pred(...)) :- <conditions>.
```

Listing 5.6: Access policy syntax

Positive and Negative Policies The permission of an access policy is either positive (`allow`) or negative (`deny`), granting and refusing access to the predicate respectively. `Pred` is the target predicate (defined or used in the Program) to which the permission applies, i.e., the access rule's object. The predicate may contain a number of atoms as arguments to constrain the applicability of the permissions, others may be left open (i.e., remain variable).

Optional `conditions` define under what circumstances the permission applies. These conditions may contain custom logic, or refer to predicates defined by the target program. Sometimes, a permission is not based on the validity of a predicate in a target program, but on whether access to that predicate is granted. Therefore, the predicate `access/1` is introduced. This predicate allows to verify if access is granted to a predicate in the target program.

Multiple permission rules may be defined on the same predicate, both allow or deny, and with different conditions or arguments. The access control module will correctly resolve the potentially conflicting policies, based on the configured access control strategy.

By supporting both allow and deny policies, ACoP allows for more fine-grained rules, in contrast to whether only one type of permission can be specified. In Listing 5.7, the specification of some example access control policies is given for a manufacturing environment. The predicate `current_user/1` requests the identifier of the entity issuing the request to access the object of the policy. The policies are the following. In general, a machine `M` is accessible to the manager of the production line in which the machine is located (policy 1). Starting a machine `M` is permitted when access to the machine itself is allowed (policy 2). However, starting a machine during night time is prohibited (policy 3). Hence, policy 3 further restricts policy 2.

```
% policy 1
allow(machine(M)) :- current_user(U), line_manager(U,P), location(M,P).
% policy 2
allow(start_machine(M)) :- access(machine(X)).
% policy 3
deny(start_machine(M)) :- night_time.
```

Listing 5.7: Example access control policies in a manufacturing environment

Completeness For *completeness*, it is required to resolve authorization when no permissions are defined. Therefore, ACoP can be configured for either an *Open* or a *Closed* policy. In an open policy strategy, access is granted by default, while in a closed strategy, access is denied. In traditional access control systems, closed policies are custom as a fail-safe alternative when no permission is defined. However, in logic programs, it could make sense to use an open policy. For instance, in a reactive system controlling a robot, all reasoning is by default allowed, except for the predicates that are used to control the robot.

Conflict Resolution To ensure consistency, proper conflict resolution is required. The meaning and resolution of permissions depends on the strategy in use. In a closed policy, one should define allow policies to provide access to predicates. In other words, accessible predicates must be 'allow listed'. To support a more fine grained allow listing, deny policies may overrule accessible predicates. As shown in Figure 5.5, a deny policy may partially overrule one or more allow policies at once.

The opposite reasoning applies for an open policy. Deny policies restrict access to predicates (deny listing). Analogous to the closed case, allow policies may overrule the deny policies, and make access less stringent.

This also defines how conflicts are resolved. When access is granted by default, this is also what takes precedence in case of a conflict. Contrarily, when access is denied, denial takes precedence.

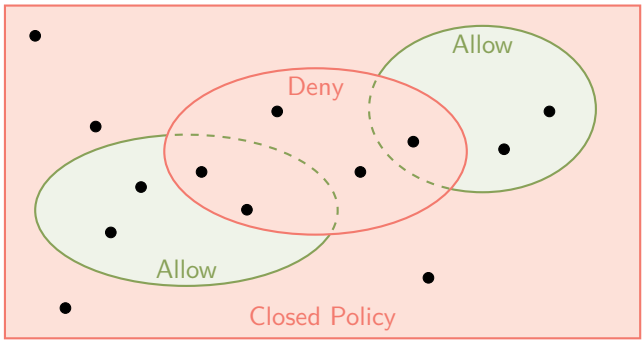


Figure 5.5: Venndiagram depicting allowed or denied predicates (•) for a closed policy

Controlled Reasoning In traditional systems, access control only applies to resources. In contrast, in logic programs, access control can be extended towards its logic rules. When no explicit permissions were found for a certain predicate, ACoP can be configured to infer permissions based on logic rules defining the predicate. In the following, this will be denoted as *body resolution*. This is achieved by scanning the knowledge base for clauses that define the predicate `P`. Permissions for `P` are inferred from the predicates defining `P` (i.e., the body). Permissions for the defining predicates may also be derived from their definition, making access control a recursive process.

Compound Statements Access on a compound statement depends on the accessibility of the predicates in the statement. Therefore, a conjunction of predicates is allowed when each predicate is accessible, while in a disjunction at least one of the predicates must be accessible. This also reflects what would happen in the 'absence' of certain knowledge.

Impure Logic A straightforward approach to integrate access control into logic programming would be to check for each resolved predicate used during inference, whether it is allowed to be accessed, and only proceed if it is. Otherwise, it fails and proceeds by backtracking. This approach would work in pure logic, but fails as soon as *impure predicates* are involved. The problem with impure predicates is that during resolution of the predicate, side effects can take place which cannot be undone during backtracking. Since access control is particularly relevant for applications where logic programs interact with external processes, e.g., sending instructions to a robot, controlling an actuator in a house, etc., it is important to cover this case. Applying access control should be transparent and handle a query as if the knowledge were absent. Access to an impure predicate must therefore be checked before side effects can take place. Hence the preliminary access control and the default strategy to deny access as soon as possible.

5.2.1.2 Terminology and Working Example

Before elaborating the process step by step, the terms *subsumption*, *unification* and *resolution* that are often used in the context of logic programming are explained in more detail. A working example in the field of smart manufacturing, used in Section 5.2.1.3, is presented. Note that in a predicate definition an upper case letter denotes a variable and a lower case letter denotes an instantiated value.

Subsumption A predicate A subsumes a predicate B if the predicate A can be made equivalent to B by only instantiating variables in A. For example `location(M,P)` subsumes `location(m1, P)` as the former can be made equivalent to the latter by only instantiating variable `M` to `m1`. The predicate `location(M,p1)` does not subsume `location(m1,P)` because, in order to make the predicates equivalent, also variables in the second predicate must be instantiated.

Unification A predicate A is unifiable with a predicate B if A can be made equivalent to B by instantiating variables in A and/or B. Similarly a predicate A is unified to predicate B if all variables are instantiated to make A equivalent to B. The predicate `location(M,p1)` is unifiable with `location(m1,P)` by instantiating the variable `M` to `m1` and variable `P` to `p1`. After unification both predicates are equal (i.e., `location(m1,p1)`).

Resolution During resolution of a predicate, a logic program recursively searches for terms in the knowledge base that unify with the predicate. After resolution the predicate is resolved. If no unifications can be found, resolution fails. For impure predicates, this is also the moment that side effects occur.

Working Example Listing 5.8 presents the working example. It consists of machines located in production lines controlled by a line manager. The impure predicates in this example are `start_machine/1`, which sends a request to start a machine, and `request_state/2` to request the current state (on/off) of a machine.

5.2.1.3 ACoP Mechanism

To integrate the above access control strategy, ACoP defines a security mechanism, able to enforce permissions on predicates. The rules are applied dynamically during logic inference of a query. It intervenes the normal execution by verifying access during resolution.

Figure 5.6 visualizes the logic used to resolve a single predicate based on its access control policies. The same scheme is used for both the open and closed policy strategy, and with or without body resolution. Each step for resolving a predicate `P` while enforcing access control is explained below. In order to elucidate the procedure, some steps are demonstrated using the example in Listing 5.8.

1. `public_predicate(P)`. The first step filters the impure and private built-in predicates from public facts and rules present in the knowledge base. For impure and private predicates, examining clauses (step 2) will be unsuccessful,

```
% FACTS
current_user(alice).
production_line(l1).        line_manager(alice, l1).
production_line(l2).        line_manager(bob, l2).
machine(m1).                location(m1, l1).
machine(m2).                location(m2, l1).
machine(m3).                location(m3, l2).

%RULES
start_production_line(P) :- production_line(P), location(M,P),
↪  start_machine(M).
machine_state(M,S) :- machine(M), request_state(M, S).

% ACCESS CONTROL POLICY 1
allow(location(_,_)).
% ACCESS CONTROL POLICY 2
allow(machine(M)) :- current_user(U), line_manager(U,P), location(M,P).
% ACCESS CONTROL POLICY 3
allow(start_machine(M)) :- access(machine(M)).
% ACCESS CONTROL POLICY 4
allow(machine_state(M,_)) :- current_user(U), line_manager(U,L),
↪  location(M,L).
```

Listing 5.8: Working example to demonstrate the ACoP mechanism

furthermore, body resolution is not meaningful. Thus, for these predicates, control can be passed to step 5. For public facts or user defined rules, control is passed to step 2.

Example The predicate `start_machine(m1)` is an impure predicate and will be forwarded directly to step 5. The predicates `machine(M)`, `machine_state(M,S)` and `start_production_line(P)` can be mapped to public facts or rules in the knowledge base, and are forwarded to step 2.

2. `clause(P, Body)`. The second step searches for clauses (i.e., facts and rules) in the knowledge base matching the predicate `P`. When a clause is found, `P` is unified with the head of that clause (denoted as `P'`). In case of rules, `Body` is unified with the body of the rule. For facts, `Body` is unified with the atom `true`. Hence, `Body` is not yet resolved and potential side effects do not take place. Alternative clauses are handled on backtracking. If no clause can be found, resolution stops.

   Example Searching for clauses that match `machine(M)`, results in `P` being unified to `machine(m1)` (i.e., `P'`), and after backtracking `machine(m2)` and `machine(m3)`. In all cases `Body` is unified to `true`. The predicate `machine_state(M,S)` is unified to `machine_state(M,S)`, hence the predicate does not change. `Body`, however, is unified with the compound term `(machine(M), request_state(M,S))`.

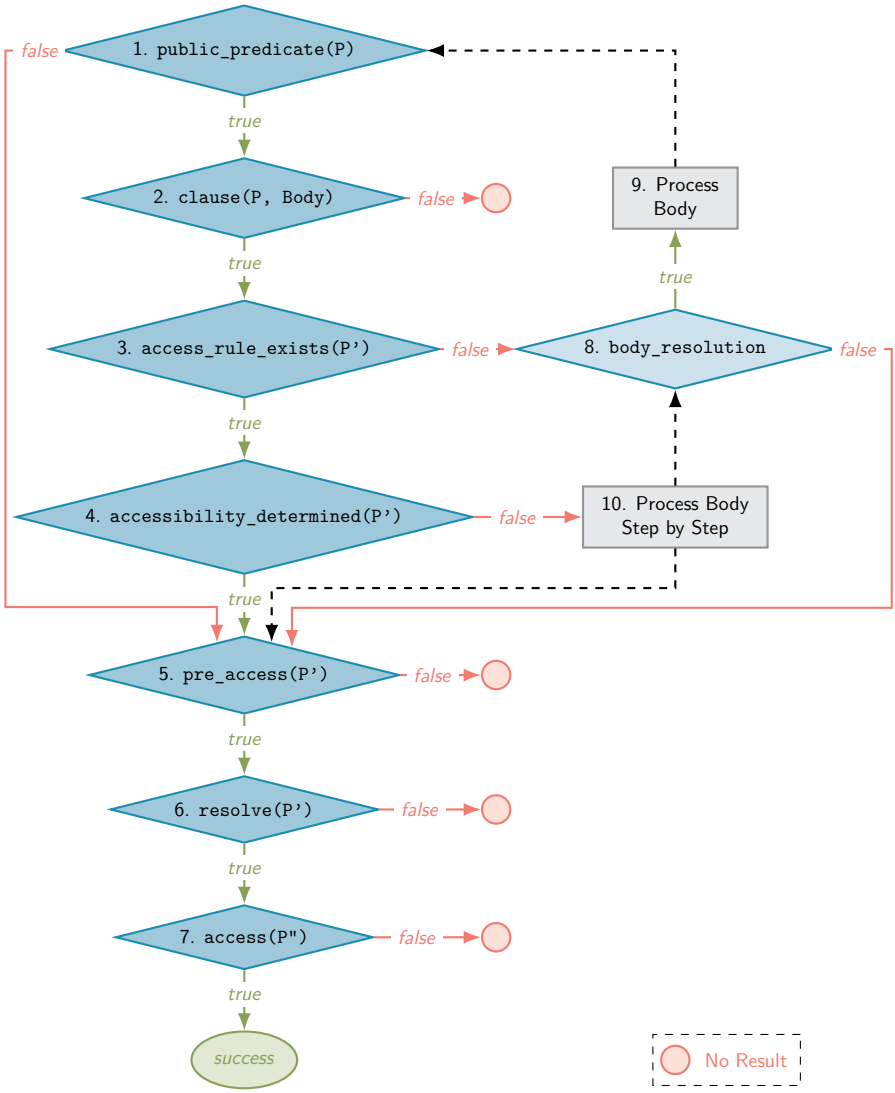


Figure 5.6: Simplified flowchart indicating the steps taken to execute access control on a predicate in the ACoP system

3. `access_rule_exists(P')`. The third step checks whether or not an access rule exists for which the target predicate is unifiable with predicate `P'`. This depends on the predicate name and the arguments of the predicate under evaluation (i.e., `P'`). A matching access control policy exists if the predicate under evaluation

can be unified to the predicate in the policy. If at least one match for predicate `P'` can be found, further action is taken in step 4. If no access rules match the predicate, access cannot be decided by the provided access rules and control is passed to step 8. The predicate will then either be resolved based on body resolution or determined by the default policy.

Example Policy 2 of the working example matches predicate `machine(m1)`, as the policy's target predicate, `machine(M)` can be made equivalent to the predicate `machine(m1)`. For `start_production_line(l1)`, however, no matching access rule can be found. In that case, the predicate, together with its matching `Body`, are forwarded to step 8.

4. `accessibility_determined(P')`. This step checks whether or not access to `P'` can already be determined based on the defined access rules. It checks whether the target predicate of each matching access rule (determined in step 3) subsumes the predicate `P'`. In addition, ACoP verifies that no variables are present in both the head and body of the access rule. Those variable terms must be instantiated before access can be properly determined. In case accessibility is determined, step 5 grants or denies access to the predicate. If accessibility is not yet determined (i.e., an access rule matches but does not yet subsume `P'`), the body of the rule will be examined until access is determined in step 10.

   Example Accessibility for `machine(m1)` can be determined, as the target predicate of policy 2 (i.e., `machine(M)`) can subsume `machine(M)` and all arguments are sufficiently instantiated to determine access. Access to query all machine states (i.e., `machine_state(M, S)`) cannot be determined yet. Although the target predicate of policy 4 (i.e., `machine_state(M,_)`) subsumes `machine_state(M, S)`, the variable `M` must be instantiated before access can be determined.

5. `pre_access(P')`. This step preliminary verifies access to the predicate before resolution. This check only fails if it is sure that access to the predicate is denied. Otherwise, the predicate is handed to step 6. Note that the default access control strategy, either open or closed, is taken into account if no access rules match. In case of pure logic, preliminary access control is only useful to stop prematurely, omitting this step would not affect the obtained results. In impure logic, however, this step prevents impure predicates to be executed if not allowed, as the occurred side effects can not be rolled back on backtracking.

6. `resolve(P')`. In this step, the predicate `P'` is resolved. In case of a rule or fact, this means that `Body` defined in step 2 is resolved. In case of an impure or private predicate, the predicate itself is resolved and possible side effects take place. As in execution without access control, when resolving the predicate fails, resolution fails. Otherwise, `P'` is resolved (denoted as `P"`) and is further handled in step 7.

7. `access(P")`. Similar to step 5, permission to access `P"` is verified. This second iteration is required since additional arguments in the predicate might be instantiated, and certain policies may become applicable. Note that the default access control strategy is also taken into account here. If access is still allowed, the resolution of the predicate ends successfully, else it fails.

8. `body_resolution`. When no explicit permissions are defined for the predicate, this step consults the configuration and checks whether `body_resolution` is active. If that's the case, it proceeds to step 9. Otherwise the predicate is passed to step 5, that takes the default strategy into account to decide upon access to the predicate.

9. *Process Body*. In this block, access to the predicate `P'` is decided based on the body of the clause found in step 2. It does this with body resolution, by repeating the entire process for each predicate in `Body`, taking into account access control in compound statements as described in Section 5.2.1.1.

   Example The predicate `start_production_line(l1)` with body `(production_line(l1), location(M,l1), start_machine(M))` ends up in this step. The entire process is thus repeated for the compound term `(production_line(l1), location(M,l1), start_machine(M))`. As a result all machines in production line will start, with the condition that the user has the authority to do so.

10. *Process Body Step by Step*. When a matching access rule exists for the predicate (step 3), but access cannot yet be determined (step 4), the terms in `Body` are resolved step by step. Processing the body step by step causes variables to be instantiated leading to one of the events below, causing the processing to stop.

    (a) *Access to the predicate `P'` becomes determined*. Enough terms in `Body` are resolved such that variables in `P'` become instantiated and allow to determine the accessibility to the predicate based on the defined access rules. In other words, there exists an access rule's predicate that subsumes predicate `P'`. Accessibility will then be decided in step 5.

    (b) *The access rules no longer apply*. It is possible that by resolving terms in `Body`, the arguments are instantiated such that there are no more access rules for which the target predicate matches with predicate `P'`. Access to predicate `P'` can then still be decided using body resolution (step 8 and 9).

    (c) *The body is entirely executed*. It is possible that after resolving the entire body, the matching access rules are still not subsumable, and will never be. Access must then be decided using body resolution if applicable (step 8 and 9).

    Special care must be taken when `Body` contains impure predicates, because side effects cannot be reversed. Therefore, an additional access control check is

performed on the original predicate before resolving impure predicates during the step by step processing of the body. Impure predicates are thus only executed if access is granted.

It is important to keep observing the original predicate to check when one of the above events occurs, as well as to keep track of the terms that have already been resolved. As a term in the body of a rule might be defined by a rule itself, processing the body step by step is a recursive process. After processing the body, already resolved terms must be taken into account such that already executed impure predicates, and corresponding side effects, are not executed twice.

Example The predicate `machine_state(M,S)` with body `(machine(M), request_state(M, S))` is handled here. To begin, the first term of the compound body (i.e., `machine(M)`) is resolved, resulting in the variable `M` being instantiated to `m1`. Consequently the original predicate is instantiated to `machine_state(m1,S)`. Now it is necessary to check again whether access to the predicate can be determined (step 4). As the predicate is sufficiently instantiated, and policy 4 subsumes the predicate, access can be decided. The predicate is forwarded to step 5, where the access is determined. As the current user is line manager of the production line where machine `m1` is part of, access is granted and the predicate can be resolved (step 6). The state of the machine is requested and returned to the user.

To generalise the case of access control from a single predicate to compound statements (used in both user queries and logic rules), ACoP applies the rules discussed for compound statements in Section 5.2.1.1. However, in a conjunction, the resolution of predicates that come later may instantiate variables. As a result, certain access rules may become applicable later. Therefore, an additional access control check is performed on the predicates earlier in the chain to ensure that access is still granted.

5.2.2 A Prolog Implementation of ACoP

To validate the security mechanism discussed in Section 5.2.1, an implementation is available for SWI-Prolog. Access control is enforced by a Prolog meta-interpreter that can be plugged in and configured in any Prolog program. The implementation can be found at https://github.com/ilse-bohe/ACoP. In this section, we will take a closer look at some of the implementation details.

The meta-interpreter takes advantage of query expansion to replace normal query resolution with inference that includes the ACoP's access control logic. This allows an almost plug-and-play use of access control in an existing program. Access control

is transparent to the user and queries send to the reasoner automatically resolve with access control in place.

In the previous section, the steps required to implement the security mechanism have been discussed. The implementation of the most important constructs in Prolog are now discussed in more detail.

Public Predicates As described in Section 5.2.1.3, the first step is to separate public predicates from private and impure predicates. This is done using the `predicate_property/2` predicate which provides the properties of a given predicate. In the proposed implementation, predicates with the *built_in* or *foreign* property are defined as private, resp. impure predicates. While the former specifies built-in predicates for which no body can be retrieved, the latter defines predicates that have its implementation defined in the C-language. The execution of such predicates often result in side effects (i.e., impure predicates).

The Access Predicate The implementation to determine access builds upon the basic predicate `subsumes_chk/2` which checks if a predicate can be subsumed by another given predicate.

To determine access to a predicate, two additional rules are defined. The `match_allow/1` and `match_deny/1` predicates, as shown in Listing 5.9, verify whether there is a definition for a positive, resp. negative permission that matches the predicate `P`. A policy matches a predicate only if the predicate in the policy (`Pol`) is more generic or equivalent to `P` (i.e., using `subsumes_chk/2`). The `access` predicate for both the open and closed policy strategy is presented.

```
% Matching allow, resp. deny policies.
match_allow(P) :- copy_term(P,Pol), allow(Pol), subsumes_chk(Pol,P).
match_deny(P) :- copy_term(P,Pol), deny(Pol), subsumes_chk(Pol,P).
% Open policy
access(P) :- (match_allow(P); \+match_deny(P)), !.
% Closed policy
access(P) :- (match_allow(P), \+match_deny(P)), !.
```

Listing 5.9: Prolog code to check access, based on currently known data

In the *open policy* case, it looks for a matching allow or the absence of a matching deny rule. Hence, access is allowed if there is *a matching allow or no matching deny*. It fails only if there is no matching allow and a matching deny policy.

In the *closed policy* case, reasoning is slightly different, and requires *a matching allow policy and no deny* to be successful. If there is no allow rule or there is a deny rule that applies to the predicate `P`, access is denied. When a predicate matches

multiple policies, backtracking is not desired, therefore, the cut-operator (i.e., `i`), which stops resolution, prevents alternative resolutions for granting access.

Processing Body Step by Step When processing the body of a rule step by step, as described in Section 5.2.1 the current state is tracked. The state keeps track of the terms that have already been resolved and the terms that have not yet been resolved, in order to prevent duplicate resolution of impure predicates. Therefore the predicate `state` is introduced and defined as follows:

```
state(Predicate, Resolved, ToResolve).
```

`Predicate` is the head of the rule and also the predicate under evaluation, `Resolved` is a list of the resolved terms and `ToResolve` is a list of the terms that are not yet resolved. As a term in the body of a rule might be defined by a rule itself, processing the body step by step is a recursive process and the `Resolved`-list can also contain states of terms. The original predicate for which access must be defined is being monitored after every step, either until access can be decided, the existing access rules are no longer applicable, or the body is completely resolved. The final state is handled depending on how the processing ended. If access is determined and allowed, all terms in `ToResolve` are being resolved. If the existing rules are no longer relevant or the body is entirely processed, body resolution can still be used to decide access. In case body resolution is applicable, it is first checked whether access to the previously resolved terms is allowed. If access is allowed, all terms in `ToResolve` are processed taking into account the access policy.

5.2.3 Application in Multiple Access Control Strategies

ACoP can easily be used to enable various access control strategies. Several examples of how possible strategies can be implemented can be found in this section.

Identity Based Access Control One of the most basic access control strategies is Identity Based Access Control (IBAC). Access to a resource is determined based on the identity of the individual trying to access the resource. An IBAC strategy for accessing files can be achieved using ACoP using the rules, described in Listing 5.10, in a closed policy. For example can access to *file1.txt* be granted for both *Alice* and *Bob*.

In IBAC, an Access Control List (ACL) is often used to bundle all identifiers together. An access control policy supportnig the use of an ACL is also proposed in Listing 5.10 assuming that `acl(L)` unifies `L` with a list of the identifiers that may access he resource.

Role Based Access Control RBAC was first introduced by Ferraiolo et. al. in 1992 [39]. It is since a widely used strategy in large companies and as the name

```
% semantics
allow(file(<filename>) :- current_user(<user_identifier>).

% example access policies
allow(file(file1.txt) :- current_user(alice).
allow(file(file1.txt) :- current_user(bob).

% example access policy using ACL
allow(file(file1.txt) :- current_user(U), acl(L), member(U,L).
```

Listing 5.10: Example access control policies for IBAC

states based on roles assigned to users of the system. An example for the role based access control strategy, is a blogpost website, where dependent on the role, a user can take several actions. The limited set of possible roles and actions that can be taken on the blogpost website can be found in Table 5.2.

	Visitor	Subscriber	Contributor	Editor	Admin
Add User					✓
Remove Users					✓
Moderate Comment				✓	✓
Publish Posts			✓	✓	✓
Edit and Delete Posts			✓	✓	✓
Comment on Posts		✓	✓	✓	✓
Read Posts	✓	✓	✓	✓	✓

Table 5.2: Roles and possible actions for a blogpost website

To enable RBAC using ACoP, the users of the system must be defined together with their role. The closed policy used for the blogpost website is described in Listing 5.11

```
% defining user roles
user_role(alice, admin).
user_role(bob, subscriber).
user_role(_, visitor).

% defining access policies on actions
allow(add_user(_)) :- current_user(U), user_role(U,admin).
allow(publish_post(_)) :- current_user(U),user_role(U,admin).
allow(post_comment(_,_)) :- current_user(U),
↪  (user_role(U,admin);user_role(U,subscriber)).
allow(read(P)).
```

Listing 5.11: Example access control policies for RBAC

Relationship Based Access Control ReBAC was first introduced by Gates in 2007 [42]. Access control policies to resources are defined based on relationships between users, and are mainly used in the context of social networking systems. Figure 5.7 gives an example of a social networking system with friend relations. Personal information and photo resources are not available to everyone. The accessibility depends on the relationship between the owner of the data and the user requesting access.

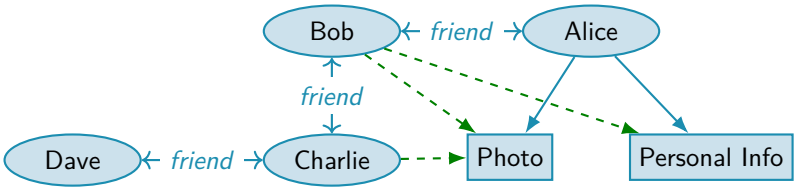


Figure 5.7: Social network system with friend relations

The access control policy is the following. A user has access to its own personal information and photos. A user has access to another users personal information, if they have a friend relation. A user has access to another users photos, if the person is a friend of the owner (i.e., has a friend relationship), or if they have a friend relation with a friend of the owner. To enable the ReBAC policy, using the ACoP system, users must be defined together with the resources they own as well as the relationships between the users. The closed policy can then be described as shown in Listing 5.12

Note that the `relation/3` predicate could also be simplified and defined using a `friend/2` predicate.

```
allow(personal_info(I)) :-  owner(O,I),(current_user(O) ;(current_user(U),
↪  friend(U,O, friend))).
allow(photo(P)) :- owner(O,P), (current_user(O) ;(current_user(U),
↪  (relation(U,O, friend); (relation(U,F,friend),relation(F,O, friend))))).
```

Listing 5.12: Example access control policies for ReBAC

5.2.4 Tests and Evaluation

Figure 5.8 shows the execution time for the query `?-machine(M)`, in function of the machines per production line. The query requests all the machines the requesting user has access to. The tests are performed on a smart manufactory similar to the one described in Listing 5.7 and consists of 3 managers each controlling 5 production lines.

Adding access control clearly has implications to the performance of the logic program. The current implementation is built to support different scenarios, but does not yet include major optimizations. Nevertheless, it is clear that the way access control policies are defined only has a linear impact on the performance of the program.

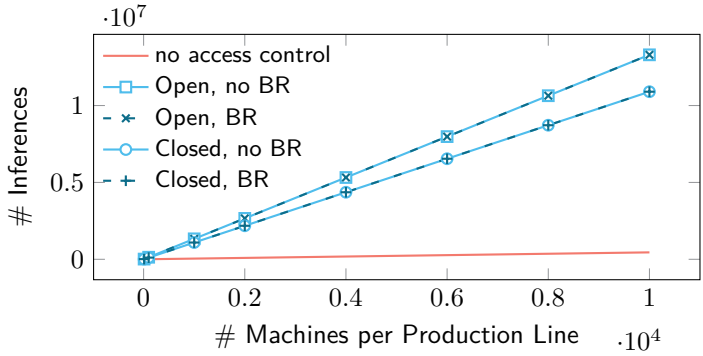


Figure 5.8: Numbber of inferences needed to request all machines for a variable number of machines per production line in ACoP

Figure 5.9 shows the number of inferences for different queries in the previously presented manufactory setting with ten machines per production line.

Several conclusions can be made based on the four performed queries.

- For queries on predicates for which access rules exist (i.e., query 1, 2 and 3), there is no difference in a setup with or without body resolution. Since matching access rules can be found, body resolution must not be performed. Thus, the execution time is independent from whether body resolution is enabled or not.
- The more specific a query is, the smaller the overhead caused by access control. This is because the number of variables to instantiate is lower (i.e., query 1 versus query 3).
- The number of inferences for an open policy are different than for a closed policy. This is both a result of how access control is handled and how the policies are defined. For the open policy, ACoP can already stop resolution if at least one matching allow rule can be found. For the closed policy, however, not only an allow rule must be found but all deny rules must be verified to be sure access is allowed. Access control could therefore be determined more quickly in the case of an open policy, which is the case for query 1. For this example, however, the open policy rules are very basic and equal to the closed policy rules complemented with a deny rule without conditions on `machine/1`, `start_machine/1` and

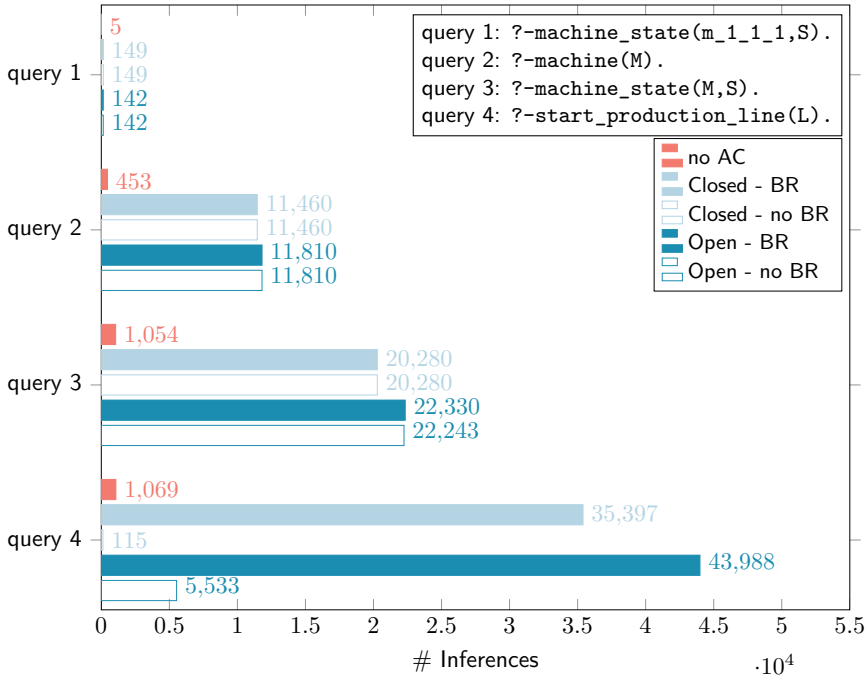


Figure 5.9: Amount of inferences needed to answer different queries, using different ACoP setups in a manufacturing environment, having ten machines per production line

`machine_state/2`. This results in an increased overhead for the open policy and becomes more apparent when more steps in the ACoP process have to be taken (i.e., queries 2, 3 and 4).

- The number of inferences for query 4 depends on the body resolution setting. There is no matching access control rule for this query. In case body resolution is disabled, ACoP can quickly decide whether or not to resolve the predicate, with little impact on the query. When body resolution is enabled, access must be controlled for each predicate in the body of the rule, which quickly increases the number of steps. Note that for this query the closed policy without body resolution is the only case for which access control is denied, resulting in a lower number of inferences than when access control is disabled.

5.2.5 Discussion

A number of pitfalls can occur based on the defined access control policies and arre discussed below.

Filtering on *outputs* When an access control policy on an impure predicate filters on 'output' arguments (i.e., arguments that only get instantiated after resolution), it implies that the predicate may be resolved before it is denied access. Although this may sound trivial, special care must be taken when it is required to prevent execution in advance. A warning during consultation time could inform the developer of such cases, to make adjustments to the policies, if necessary.

The access predicate The `access` predicate may be used in the body of a rule to verify whether access to another predicate is allowed. This may possibly lead to infinite loops. Hence, special care must be taken when this predicate is used. Especially when body resolution is active.

Insufficient instantiation Often, access control policies filter on the values of arguments. In complex cases, one of the arguments of the predicate under control may be used to compare with some value. Although this seems intuitively correct, the query results in an error. Since access control is also verified before resolving the predicate, the arguments of the predicate may still be variable (both for pure and impure predicates). Using variables, while non-variables are expected may result in unexpected behavior. Since it cannot be derived from the predicates whether arguments are allowed to be variable, it is not possible to verify this automatically. An example is shown in Listing 5.13.

```
age(X,A) :- info(X,birthdate,date(Y)), calculate_age(Y,A).
allow(age(_,A)) :- A>18.
?- age(X,Y).
```

Listing 5.13: Example code resulting in an insufficient instantiation error

To handle this, either the arguments used in the filter must be instantiated properly, or the developer needs to take additional measures when defining the rules. For instance, `nonvar/1` can be used to check whether a term is already instantiated, before performing arithmetic operations. Another solution to solve such problems is by giving the possibility to ignore the argument during preliminary access control. For instance, by annotating such arguments.

Support and conflict resolution As discussed in Section 5.2.1.1, conflict resolution is determined by the default policy. In a closed policy, deny rules take precedence, whereas in an open policy, allow rules take precedence, even in

the presence of body resolution. This makes it possible to write access rules that will not be considered, but give the developer an unjustified feeling of control. Adding support to track and warn users of aforementioned cases could prevent the misleading feeling of security.

Data privacy Although access control may help in preventing access to specific information, it does not prevent that rules may still leak information. The common example is when a clerk is using a program to enter the salary of an employee. A rule states that the salary cannot be higher than the salary of the director. Although access to the salary of the director may be denied, the clerk can derive the salary of the director by checking multiple salaries and see if it is allowed to be entered or not. Thus, additional measures may be required if privacy is at stake.

5.3 Conclusion

The work in this chapter can clearly be separated in two parts.

In the first part a logic reasoning IoT middleware is presented. The module based architecture makes it possible to integrate functionalities such as access control, automation, preprocessing and connection management to IoT applications. Using an event bus, communication between modules is handled seamlessly. User defined policies, which are easy to set up, define the internal working of the modules. A demonstrator in JavaScript, using Tau-Prolog to handle the logic, ensures that the middleware can be integrated in cloud, gateway and mobile applications, and even inside IoT devices.

The second part presents an access control mechanism that enforces access control in existing logic programs, called ACoP. A *deny as soon as possible* strategy is applied and various configurations make it possible to adapt the system to the needs of the use case. The presented solution can be integrated as module in the previously defined reasoning middleware to add access control capabilities to IoT applications. However, it can also be used in other logic programs to enforce access control, not only on the knowledge base, but also to the logic program itself. ACoP takes into account the use of *impure predicates*, which trigger side effects that cannot be reversed, that are often used in an IoT setting. The methodology is discussed in detail by means of an example. For validation, ACoP is implemented in SWI-Prolog and applied to several access control strategies, namely IBAC, RBAC and ReBAC. Finally, the different configurations of the system are evaluated.

Chapter 6

Supporting Software Integrators in Building IoT Applications

In this chapter we elaborate on the valorisation potential from a software integrator perspective. The task of such companies is, upon request of customers, to connect multiple software subsystems and make them work as a whole. In the context of this PhD, the focus is on software integrators in the IoT space. Such companies want to develop robust applications in a short time and in a cost-efficient manner. The research conducted in this PhD can be a first step towards building reusable architectures, hence, reducing development time and cost.

The valorisation perspectives are twofold. First, the results can be bundled into design and development guidelines. Secondly, software support can be provided by means of software tools and templates.

By using the proposed guidelines and tools, companies can counteract vendor lock-in. This will lead to more attractive and sustainable systems, which can be a clear advantage over their competitors. Next generation systems can be developed that, compared to the static first generation systems, are dynamically adaptable to the needs of the user and environment.

6.1 Design and Development Guidelines

A clear guide, either digital (i.e., online modules) or physical (i.e., book), can help software companies to develop reusable IoT applications. Based on the Software Development Life Cycle (SDLC), a step-by-step guide explains which tasks must be performed in each stage and how.

A clear distinction must be made between the necessary stakeholders in the process, namely customers, requirement analysts and environment designers. We assume that software integrators select commercial-of-the-shelf (COTS) devices and do not perform any hardware design or development. Note that multiple roles can be taken by the same entity.

The number of tasks when developing Internet of Things (IoT) applications is big, but can be clearly separated and assigned to the stakeholders to provide a clear guide. The SDLC is used as starting point to define these tasks.

To provide more tangible information, case studies of different companies should be included. An example case study can be found in Appendix A. In the scope of a two year Research and Development Project funded by Vlaio, the proposed work was applied to an audiovisual use case in collaboration with the Belgian company APEX[1]. APEX is a manufacturer of professional audio and audiovisual equipment. Among others, their solution is applied in retail, museums, bars, restaurants and funeral homes.

6.2 Software Support

A second type of valorisation is oriented towards software support tools and frameworks. Referring to the Chapters 3, 4 and 5, several software tools/frameworks can be developed to promote application-driven development within software integrator companies.

Architectural Middleware Frameworks A first major contribution are the architectural frameworks of which a prototype is described in Chapter 3. These frameworks should not be limited to the previously proposed Android and JavaScript implementations but can also be extended to other languages and platforms. Open-sourcing of such frameworks can only benefit the community. Note that it is not the goal to compete with existing domain specific frameworks such as Home Assistant[2] and openHAB[3]. The goal is to give large IoT integrator companies a starting point from which they can implement and tune an architecture depending on their own requirements.

Device Catalog A second contribution to speed up the development process is the composition of the device catalog described in Section 3.2. By building such catalogs, either internally or publicly, it is possible to quickly determine the most suitable devices for various IoT projects. It is very important that a catalog is kept up to date, the public creation of such catalogs can aid in this.

[1] https://www.apex-audio.be/
[2] https://www.home-assistant.io/
[3] https://www.openhab.org/

Modeling and Management Tool Modeling the IoT environment can be supported by a tool that makes it possible to create and manage the entire model based on the meta models described in Chapter 4. Relying on that model, assets and devices can be inventoried and coupled to each other. In combination with the device catalog, this tool can provide the best choices in devices. Based on the physical connected devices, the tool can provide feedback about requirements that have or have not been met.

Reasoning Middleware The next valorization contribution is a fully elaborated reasoning middleware. In combination with the virtualization frameworks, developers can develop advanced IoT applications. Adding functionalities such as automation, access control and querying capabilities can be done with a limited coding effort. The asset-based definition of rules and policies makes it a lot easier to express desired functionalities, especially for people with a non-technical background.

Policy Designing Tool While it is a lot easier to define applications using asset-based rules and policies, using logic programming languages, such as Prolog, comes with a learning curve. It is not always easy for people without appropriate programming skills to define these rules in a correct way. Even for people who are familiar with programming, checking whether there are any rule conflicts is by far not trivial. Graphical tools, which make it possible to visualize rules in a user-friendly way and also take constraints into account, can largely facilitate the development and configuration of IoT ecosystems.

If all the tools described above seamlessly interact, not only development and configuration, but also maintenance can be simplified a lot. It takes effort and investment to develop such tools and learning to operate them. With efficient use, however, companies can certainly recoup this investment in the time saved.

6.3 Business Model

When drawing up a business model for a company, it is not only necessary to take into account how turnover can be achieved. It must also be clearly identified who the potential customers are, which product or service is exactly being offered and how this is achieved. Gassmann et. al. [41] proposed a visual definition of what a business model should define, the business model navigator, which can be seen in Figure 6.1.

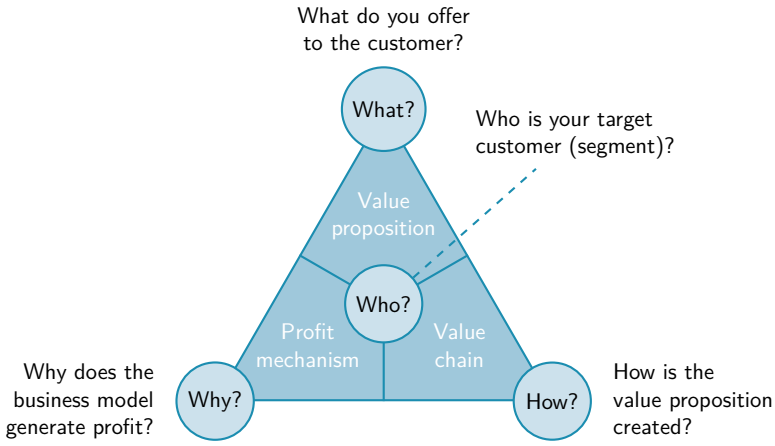


Figure 6.1: Business model navigator [41]

The customer – *who* – can be found centrally in the model. They are the most important part of the model, if a company has no customers, it won't be able to gain profit. If the product or service offered is not adapted to the needs of the customer, the company won't sell. It is thus important for the company to define who it wants to (or don't want to) target with the product or service. Each other aspect can be positioned relative to the central customer. The value proposition – *what* – defines the product(s) and/or service(s) the company provides to the customer and how it is adapted to the customers needs. In order to make the value proposition possible, the product(s) and/or services must be developed. The needed resources and capacities, and the development process compose the value chain – *how*. Last, the profit mechanism – *why* – defines cost structures and the revenue stream. The question *"why does this model generate profit"* is answered, and viability is checked.

We will now briefly define each of these four parts for a possible business model. We define a business model for a software integrator company, providing accessible software development for the private sector and Small and Medium Enterprises

(SMEs). We define for whom (i.e private sector and SME), what the business offers, how it can be realized and how it can become a profitable business. Note that this is not the only possible valorisation direction for the work conducted during this PhD.

Who? The services and software are developed for the private sector and SMEs. The goal is to develop affordable IoT applications in various fields based on the requirements of the customer. Examples can be a personalized home automation application, a monitoring application for an estate property or a company branded application for access control and monitoring on the work floor.

What? We offer customized IoT applications for iOS and Android in various domains. We provide integrations of several COTS IoT devices but also provide the ability to integrate external customized devices on request. Application can be customized in different ways, not only the user interface but also functionalities can be adapted or added to the needs of the customer. Customers can thus not only increase their productivity but also strengthen their own identity or the identity of the company.

How? In order to be able to create customized IoT applications, the software framework must be further developed and an iOS port of the framework must be made. A preliminary set of IoT integrations must be added, so that a set of possible integrations can already be offered to potential customers. This platform then needs to be properly maintained to support recent technologies. It therefore remains a continuous investment to keep the software up to date and integrate new device technologies. After that, application development according to the customer's needs can be started. This development can of course take place in sprints, so that a basic application can already be delivered in the short term. Further elaborations and addition of functions can then be added with each update.

Why? Revenue can be created by working with a subscription based model. By combining a fixed cost with a variable cost depending on the number of integrations and functionalities it is an attractive model for both the customer and the business. Different tiers (e.g., basic, standard and premium) can offer pricing solutions based on the customers needs and the company enjoys a continuous revenue stream. For example, a basic subscription can support the integration of ten IoT systems. A standard subscription can then provide the integration up to 20 IoT systems together with access control and automation functionalities.

The above version of the business model is a starting point. In order to draw up a fully-fledged business model, requirements of potential customers can be evaluated. Also the revenue model needs further development so that a viable business can be created.

Chapter 7

Conclusion

This PhD trajectory investigated how software companies can be aided during the design and development of maintainable Internet of Things (IoT) applications, taking into account both the dynamic nature of the IoT ecosystems and the constantly evolving IoT landscape. With the Software Development Life Cycle (SDLC) as a guidance, modeling guidelines, architectures and tools were presented to achieve this goal.

This chapter evaluates the obtained results by looking back at the research questions defined in section 1.2.

Research Question 1 What sensor/actuator abstractions are appropriate towards application developers? What are the functionalities and tasks that middleware must provide to support these abstractions?

Research Question 2 Can application programmers reason in terms of assets (i.e., objects at application level) instead of sensors? Can an architecture hide the underlying sensor complexity completely towards application developers?

Research Question 3 Which are feasible architectural decisions that can lead to increased flexibility with respect to sensor selection/replacements, and hence, contribute to vendor lock-in avoidance?

Research Question 4 Can the software architecture also support the realization of other non-functional concerns like dynamics and security?

7.1 Obtained Results

The obtained results could clearly be separated in three major chapters. The chapter on the IoT architecture & middleware (Chapter 3) and the chapter on the modeling

of IoT ecosystems (Chapter 4) each lead to one major contribution. The chapter on reasoning for IoT ecosystems (Chapter 5) leads to two major contributions.

Contribution 1 - Application Centric Architecture This first part presented an architecture supporting IoT application developers in building maintainable IoT applications. With application centric development in mind, two abstraction layers were defined that allow to decouple the application from the underlying IoT infrastructure. *Virtual IoT Devices* provide a uniform and intuitive interface for device access and shields complex low-level and non-functional aspects like communication mechanisms and security protocols from the application developer. Intuitive Application Programming Interfaces (APIs) in an additional *Asset Layer* imply that the developer can focus on implementing the business logic. The architectural insights are incorporated in both an Android and JavaScript framework and validated through the design and development of a care home ecosystem consisting of various *Assets*.

This part contributes to *Research Question 1*, *Research Question 2* and *Research Question 3*. Appropriate device abstractions are done at device type level. For example, all lamps, independent from device technology, will provide the same interface to upper levels. However, not all lamps have the same functionalities. Informing the users about degradation of the service, based on the specific technology used, gives them the opportunity to change the technology in case a specific function is needed. In addition, the way in which data is formatted is not the same for every technology. If a specific standard format is chosen, each deviating format needs to be converted so that integration of different technologies can proceed without problems. In order to provide these abstractions, a decoupling layer must be provided. In this layer the devices are virtualized, hence the name *Virtual Device Layer*. On top of the *Virtual Device Layer*, an *Asset Layer* can provide reasoning based on items of interest (i.e., assets) in the physical environment. In this way, the entire sensor complexity can be hidden from application developers.

Contribution 2 - Ecosystem Modeling Guidelines The second part presented an application centric approach to design, develop and operate advanced reconfigurable IoT ecosystems. Separation-of-duties, loose coupling between the business logic and the IoT infrastructure, and selective binding of edge devices to applications are key tactics. Technology-agnostic application policy definition is a central building block which means that correct application behavior is expressed in terms of asset methods and states, and relations between them. Policies are then mapped to operations on IoT infrastructural elements. We showed that our approach facilitates the design of new applications within the same IoT ecosystem, redefinition of behavior of already existing applications and modifications in the underlying infrastructure.

This part maps to *Research Question 2* and *Research Question 3*. By first modeling the ecosystem based on *Assets* in the environment, you can quickly move on to application development. This often makes development more intuitive. Application requirement can be defined by the requesting party on the basis of these *Assets*. Therefore, implementation of these requirements does not require an additional translation step. Linking devices to the applications can happen independently from application development. Replacing a device therefore has no impact on the application. Only configurations have to be adjusted, thus avoiding vendor lock-in.

Contribution 3 - Reasoning Middleware In the first part of Chapter 5, a logic reasoning based IoT middleware is presented. It is available for integration in either cloud, mobile and gateway applications, and even inside IoT devices. The flexible event-based architecture, hosting different modules, supports application developers in building and maintaining smart and reactive IoT applications. Including contextual information in rules and policies is possible. This reasoning middleware can be combined with the redefined IoT middleware to build advanced IoT applications, with additional functionalities such as automation and access control. The integrated reasoner brings advanced intelligence to IoT applications and enforces access control and connection policies. A demonstrator of the middleware is integrated using a JavaScript based back-end. Advanced Android, iOS and NodeJS applications can be built in at a fast pace.

Contribution 4 - Access Control in Logic Programming The second part of Chapter 5 presented a mechanism that enforces access control in existing logic programs (called ACoP). Access control is fine grained at predicate level, supporting multiple established access control strategies. The solution takes into account the use of impure predicates and applies a *deny as soon as possible* strategy to prevent prohibited side effects from taking place. The presented Prolog meta-interpreter shows that the integration does not entail excessive overhead. This strategy can easily be incorporated in the previously described reasoning middleware and in this way add access control capabilities to IoT applications.

Answer to Research Question 4 can be covered by the last two aforementioned contributions. The reasoning middleware can be extended with appropriate modules. In this way both functional and non-functional requirements can be added to applications. The reasoning middleware already hosts a connection manager that takes care of the dynamic nature of IoT ecosystems. Devices coming in or getting out of range are seamlessly connected to or disconnected from the application. The strategy to integrate access control into logic programming languages makes it possible to add privacy protection to applications. Application developers are shielded from a large implementation effort, as it only entails defining the access control policies.

7.2 Future Research Possibilities

During validation of the presented architecture and middleware, prototypes of moderate size were built consisting up to 100 devices and 10 stakeholders. Developing larger ecosystems with a higher order of devices and stakeholders may reveal new insights and raise additional scalability requirements. The focus on the distributed nature of the architecture might be enlarged, and adjustments may then lead to a more robust and even more widely applicable architecture.

When performing the work during the doctoral trajectory, security was not a major concern. However, security becomes increasingly important in open IoT environments embracing multiple stakeholders and devices from varying vendors. Moreover, this is a very complex challenge as in IoT – like in any other open digital ecosystem – security is as weak as its weakest link [93]. Further research in this area may lift the proposed architecture to a next level. When we no longer assume that the incorporated devices are secure, mitigating countermeasures must be taken. To tackle these challenges, a security wall can shield the applications from malicious IoT devices.

Appendix A

Case Study - Reusable Multimedia Platform in Collaboration with APEX

In collaboration with the Belgian based company APEX[1] the work in this dissertation is valorized and applied to an audiovisual use case . APEX is a manufacturer of professional audio and audiovisual equipment. Among others their solution is used in retail, museums, bars, restaurants and funeral homes.

We go through the SDLC to develop several applications in the audiovisual setting using the presented tools and guidelines. First the different applications and their requirements are determined. Thereafter, the ecosystem is modeled according to the needs of the applications, taking portability into account. By making only minor adjustments, the ecosystem can be ported to a different setting. On the basis of the predefined requirements and the design, applications can be developed completely device agnostic. Communication with IoT devices is no concern of the application developer. Finally, device selection can take place, and the one time development to integrate the different devices must be done.

A.1 Application Analysis and Requirements

Two applications, each in a different setting, are defined. One application in the funeral home setting, the other in a restaurant setting. The requirements of each application are elaborated an split up in different versions.

[1] https://www.apex-audio.be/

A.1.1 Smart Application for Funeral Home Ceremonies

The first application is used to configure funeral ceremonies and make sure that during the ceremony everything works as planned. Ceremonies can "PowerPoint" wise be designed with visuals (images and videos), sounds and lightning. Each funeral home consists of areas. The different ceremony rooms are separate areas, but also in a single room different areas can be specified. Examples are, the platform in front, the seating area and the side walls. An example ceremony room is shown in Figure A.1

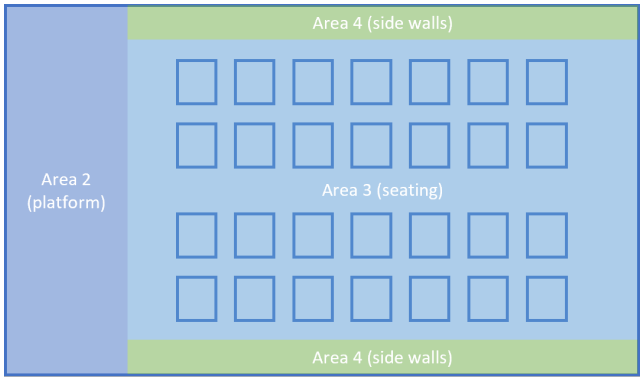


Figure A.1: Example ceremony room

Settings (i.e., visuals, sound and lightning) can be configured per area. In a first version of the application (*v1*) these settings are configured in a configuration file. In a second version of the application (*v2*) a Graphical User Interface (GUI) can be used to configure the settings.

A.1.2 Smart Application for Bars and Restaurants

In a bar and restaurant, different areas can be determined up to the level of different tables, see Figure A.2.

In each area, music and lightning can be configured. Predefined configurations (e.g. *summer lunch*, *romantic dinner*, *evening lounge*) can be used. On top of that, lightning brightness is adapted based on the overall clarity in the room.

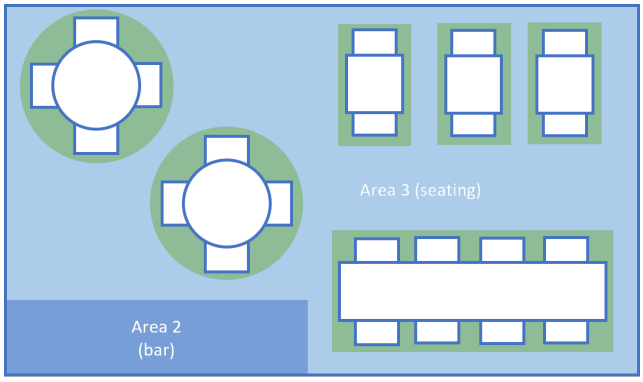


Figure A.2: Example restaurant area

A.2 Design

Based on the requirements of both applications the assets and the environment model are designed. Figure A.3 shows the environment model.

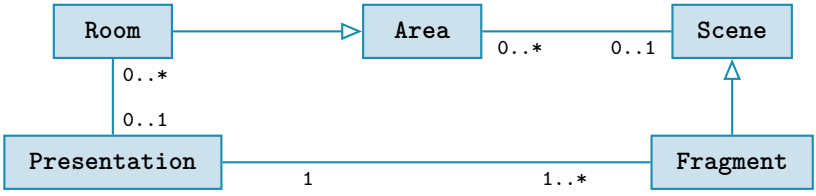


Figure A.3: $Model_{Environment}$ for the APEX use case

The environment consists of **areas**. An area is a part of physical space and is not necessarily delimited by physical elements. Each area can have child areas and thus an area can be part of multiple other areas. For example can the *bar* area be part of both the *dance floor* and the *seating area*. A **scene** can be set to one or more areas. A scene is a set of predefined configuration. An area can only have one scene at a time or none at all. A **room** is a specific type of area, to which a **presentation** can be coupled. A presentation is a sequence of **fragments**. Fragments are scenes with a specific duration. The configurations of a fragment are thus only applied for a specific amount of time. Fragments can be compound and can thus exist of multiple other fragments.

Different fragments of the presentation can be set to areas or activated based on different triggers:

- time trigger: on specific time
- sequence trigger: after another fragment is terminated
- external trigger: someone pushed the next button

Each scene can contain the following elements with their corresponding parameters between square brakets.

- a list of visuals [visuals]
  - video-files
  - image files, with a time to display the image (general or specific per image)
- a list of sound files, with optional time to play [music]
- lightning settings [brightness, color]
- volume [volume]

Bibliography

[2] M. Abadi. "Logic in access control". In: *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.* IEEE. 2003, pp. 228–233.

[5] A. Al Farooq et al. "Iotc 2: A formal method approach for detecting conflicts in large scale iot systems". In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM).* IEEE. 2019, pp. 442–447.

[7] C. Bayılmış et al. "A survey on communication protocols and performance evaluations for Internet of Things". In: *Digital Communications and Networks* (2022). ISSN: 2352-8648.

[8] A. Bhawiyuga et al. "Architectural design of IoT-cloud computing integration platform". In: *Telkomnika (Telecommunication Computing Electronics and Control)* 17.3 (2019), pp. 1399–1408.

[10] I. Bohé et al. "A Crowdsensing Solution for Tracking Bicycle Path Conditions". In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT).* IEEE. New Orleans, LA, USA - Online, 2020, pp. 1–6.

[11] I. Bohé et al. "A Logic Programming Approach to Incorporate Access Control in the Internet of Things". Accepted at IFIP IoT 2022. Amsterdam, the Netherlands.

[12] I. Bohé et al. "An extensible approach for integrating health and activity wearables in mobile IoT apps". In: *2019 IEEE international congress on Internet of Things (ICIoT).* IEEE. Milan, Italy, 2019, pp. 69–75.

[13] I. Bohé et al. "SMIoT: a software architecture for maintainable internet-of-things applications". In: *International Journal of Cloud Computing* 9.1 (2020), pp. 75–94.

[14] I. Bohé et al. "Towards low-effort development of advanced IoT applications". In: *Proceedings of the 8th International Workshop on Middleware and Applications for the Internet of Things (M4IoT).* Québec, Canada - Online, 2021, pp. 1–7.

[15] I. Bohé et al. "Untangling the Physical-Digital Knot When Designing Advanced IoT Ecosystems". In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things (M4IoT).* UC Davis, CA, USA, 2019, pp. 1–6. ISBN: 9781450370288.

[16] R. Böhme et al. "A fundamental approach to cyber risk analysis". In: *Variance* 12.2 (2019), pp. 161–185.

[17] P. Bonte et al. "Subset Reasoning for Event-Based Systems". In: *IEEE Access* 7 (2019), pp. 107533–107549.

[18] A. Botta et al. "Integration of cloud computing and internet of things: a survey". In: *Future generation computer systems* 56 (2016), pp. 684–700.

[19] P. Bourhis et al. "JSON: Data model and query languages". In: *Information Systems* 89 (2020), p. 101478. ISSN: 0306-4379.

[20] F. Bruckner. et al. "A Framework for Creating Policy-agnostic Programming Languages". In: *Proceedings of the 9th International Conference on Data Science, Technology and Applications - DATA,* INSTICC. SciTePress, 2020, pp. 31–42. ISBN: 978-989-758-440-4.

[21] R. Calegari et al. "Logic Programming as a Service (LPaaS): Intelligence for the IoT." In: *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*. IEEE, 2017, pp. 72–77.

[22] G. Chen et al. "Modeling and reasoning of IoT architecture in semantic ontology dimension". In: *Computer Communications* 153 (2020), pp. 580–594. ISSN: 0140-3664.

[23] S. Cheruvu et al. "Connectivity technologies for IoT". In: *Demystifying internet of things security: Successful IoT Device/Edge and Platform Security Deployment*. Springer, 2020, pp. 347–411. ISBN: 978-1-4842-2896-8.

[27] S. K. Datta et al. "oneM2M architecture based IoT framework for mobile crowd sensing in smart cities". In: *2016 European Conference on Networks and Communications (EuCNC)*. 2016, pp. 168–173.

[28] S. De Capitani di Vimercati. "Access Control Policies, Models, and Mechanisms". In: *Encyclopedia of Cryptography and Security*. Ed. by H. C. A. van Tilborg et al. Springer US, 2011, pp. 13–14. ISBN: 978-1-4419-5906-5.

[29] H. Demirkan. "A Smart Healthcare Systems Framework". In: *IT Professional* 15.5 (2013), pp. 38–45. ISSN: 1520-9202.

[30] P. Desai et al. "Semantic Gateway as a Service Architecture for IoT Interoperability". In: *2015 IEEE International Conference on Mobile Services*, pp. 313–319.

[32] J. Ding et al. "IoT connectivity technologies and applications: A survey". In: *IEEE Access* 8 (2020), pp. 67646–67673.

[33] J. Dizdarević et al. "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration". In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–29.

[34] B. Edwards et al. "Hype and heavy tails: A closer look at data breaches". In: *Journal of Cybersecurity* 2.1 (2016), pp. 3–14.

[35] R. Elmasri et al. *Fundamentals of database systems*. Pearson, 2017.

[37] Z. M. Fadlullah et al. "Toward intelligent machine-to-machine communications in smart grid". In: *IEEE Communications Magazine* 49.4 (2011), pp. 60–65. ISSN: 0163-6804.

[38] G. Feng et al. "Floor pressure imaging for fall detection with fiber-optic sensors". In: *IEEE Pervasive Computing* 15.2 (2016), pp. 40–47.

[39] D. Ferraiolo et al. "Natl Institute of Standards and Tech., Dept. of Commerce, Maryland, Role-Based Access Control". In: *Proceedings of 15th Annual Conference on National Computer Security*. National Institute of Standards and Technology, 1992, pp. 554–563.

[40] A. Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4 (2015), pp. 2347–2376. ISSN: 1553-877X.

[41] O. Gassmann et al. *The business model navigator: 55 models that will revolutionise your business*. Pearson UK, 2014.

[42] C. Gates. "Access control requirements for web 2.0 security and privacy". In: *IEEE Web* 2 (2007), pp. 12–15.

[43] C. B. Gemirter et al. "A Comparative Evaluation of AMQP, MQTT and HTTP Protocols Using Real-Time Public Smart City Data". In: *2021 6th International Conference on Computer Science and Engineering (UBMK)*. IEEE. 2021, pp. 542–547.

[47] P. Gupta et al. "IoT based smart healthcare kit". In: *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*. IEEE. 2016, pp. 237–242.

[48] R. Herrero. *Fundamentals of IoT Communication Technologies*. Springer, 2022.

[49] R. Herrero. "Thread Architecture". In: *Fundamentals of IoT Communication Technologies*. Springer, 2022, pp. 213–225.

[51] P. Hough et al. "The Accuracy of Wrist-worn Heart Rate Monitors across a Range of Exercise Intensities". In: *Journal of Physical Activity Research* 2.2 (2017), pp. 112–116.

[53] S.-L. Hsieh et al. "A wrist-worn fall detection system using accelerometers and gyroscopes". In: *Proceedings of the 11th IEEE International Conference on Networking, Sensing and Control*. IEEE. 2014, pp. 518–523.

[54] N. Huynh et al. "SGAC: A patient-centered access control method". In: *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. 2016, pp. 1–12.

[56] D. Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 256–290. ISSN: 1049-331X.

[57] J. M. Kang et al. "A wrist-worn integrated health monitoring instrument with a tele-reporting device for telemedicine and telecare". In: *IEEE Transactions on Instrumentation and Measurement* 55.5 (2006), pp. 1655–1661.

[58] W. Kassab et al. "A-Z survey of Internet of Things: Architectures, protocols, applications, recent advances, future directions and recommendations". In: *Journal of Network and Computer Applications* 163 (2020), p. 102663. ISSN: 1084-8045.

[59] V. Kolovski et al. "Analyzing web access control policies". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 677–686. ISBN: 9781595936547.

[60] V. Kumar et al. "Development of Electronic Floor Mat for Fall Detection and Elderly Care". In: *Asian Journal of Scientific Research* 11 (2018), pp. 344–356.

[61] R. Lea et al. "City Hub: A Cloud-Based IoT Platform for Smart Cities". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. 2014, pp. 799–804.

[62] M. Leuschel et al. "ProB: A Model Checker for B". In: *FME 2003: Formal Methods*. Ed. by K. Araki et al. Springer Berlin Heidelberg, 2003, pp. 855–874. ISBN: 978-3-540-45236-2.

[63] I. I. Lysogor et al. "Survey of data exchange formats for heterogeneous LPWAN-satellite IoT networks". In: *2018 Moscow workshop on electronic and networking technologies (MWENT)*. IEEE. 2018, pp. 1–5.

[65] R. Machado et al. "An IoT Architecture to Provide Hybrid Context Reasoning". In: *Internet of Things. A Confluence of Many Disciplines*. Springer International Publishing, 2020, pp. 86–102. ISBN: 978-3-030-43605-6.

[66] L. Magnoni. "Modern messaging for distributed sytems". In: *Journal of Physics: Conference Series*. Vol. 608. 1. IOP Publishing. 2015, p. 012038.

[67] S. Mahanthappa et al. "Data formats and its research challenges in iot: A survey". In: *Evolutionary Computing and Mobile Sustainable Networks* (2021), pp. 503–515.

[68] D. Maier et al. *Computing with Logic: Logic Programming with Prolog*. Benjamin-Cummings Publishing Co., Inc., 1988. ISBN: 0805366814.

[70] E. Al-Masri et al. "Investigating messaging protocols for the Internet of Things". In: *IEEE Access* 8 (2020), pp. 94880–94911.

[71] N. Matthys et al. "μPnP-Mesh: the plug-and-play mesh network for the internet of things". In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE. 2015, pp. 311–315.

[72] R. Maurya et al. "Application of Restful APIs in IOT: A Review". In: *International Journal for Research in Applied Science & Engineering Technology* 9.2 (2021), pp. 145–151. ISSN: 2321-9653.

[73] K. Mekki et al. "A comparative study of LPWAN technologies for large-scale IoT deployment". In: *ICT express* 5.1 (2019), pp. 1–7.

[75] S. Mumtaz et al. "Massive Internet of Things for Industrial Applications: Addressing Wireless IIoT Connectivity Challenges and Ecosystem Fragmentation". In: *IEEE Industrial Electronics Magazine* 11.1 (2017), pp. 28–33. ISSN: 1932-4529.

[76] G. Naik et al. "A Brief Comparative Analysis on Application Layer Protocols of Internet of Things: MQTT, CoAP, AMQP and HTTP". In: *International Journal of Computer Science and Mobile Computing* 9 (2020), pp. 135–141.

[77] N. Naik. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In: *2017 IEEE international systems engineering symposium (ISSE)*. IEEE. 2017, pp. 1–7.

[78] A. Ngu et al. "IoT middleware: A survey on issues and enabling technologies". In: *IEEE Internet of Things Journal* 4.1 (2016), pp. 1–20.

[82] H. Raes. "Verzekeren van Quality-of-Service in IoT toepassingen". MA thesis. KU Leuven. Faculteit Industriële Ingenieurswetenschappen, 2019.

[83] J. M. Raja et al. "Apple watch, wearables, and heart rhythm: where do we stand?" In: *Annals of translational medicine* 7.17 (2019).

[84] R. dos Reis et al. "A Soft Real-Time Stream Reasoning Service for the Internet of Things". In: *13th IEEE International Conference on Semantic Computing, ICSC 2019*. IEEE. IEEE, 2019, pp. 166–169.

[85] R. Roman et al. "On the features and challenges of security and privacy in distributed internet of things". In: *Computer Networks* 57.10 (2013), pp. 2266–2279. ISSN: 1389-1286.

[86] K. Sairam et al. "Bluetooth in wireless communication". In: *IEEE Communications Magazine* 40.6 (2002), pp. 90–96.

[87] P. Samarati et al. "Access Control: Policies, Models, and Mechanisms". In: *Foundations of Security Analysis and Design*. Ed. by R. Focardi et al. Springer Berlin Heidelberg, 2001, pp. 137–196. ISBN: 978-3-540-45608-7.

[89] R. Sanchez-Iborra et al. "State of the Art in LP-WAN Solutions for Industrial IoT Services". In: *Sensors* 16.5 (2016). ISSN: 1424-8220.

[90] J. Santiago et al. "Fall detection system for the elderly". In: *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2017, pp. 1–4.

[91] S. Al-Sarawi et al. "Internet of Things (IoT) communication protocols". In: *2017 8th International conference on information technology (ICIT)*. IEEE. 2017, pp. 685–690.

[92] S. Sartoli et al. "Modeling adaptive access control policies using answer set programming". In: *Journal of Information Security and Applications* 44 (2019), pp. 49–63. ISSN: 2214-2126.

[93] B. Schneier et al. *Beyond fear: Thinking sensibly about security in an uncertain world*. Vol. 10. Springer, 2003.

[94] A. Shamsundar. "Modeling and performance evaluation of the Thread protocol". MA thesis. Delft University of Technology, 2017.

[95] J. Sidna et al. "Analysis and evaluation of communication Protocols for IoT Applications". In: *Proceedings of the 13th international conference on intelligent systems: theories and applications*. 2020, pp. 1–6.

[97] M. Sikimić et al. "An overview of wireless technologies for IoT network". In: *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE. 2020, pp. 1–6.

[99] M. Soliman et al. "Smart Home: Integrating Internet of Things with Web Services and Cloud Computing". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. 2013, pp. 317–320.

[100] L. Sterling et al. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[102] J. Swetina et al. "Toward a standardized common M2M service layer platform: Introduction to oneM2M". In: *IEEE Wireless Communications* 21.3 (2014), pp. 20–26. ISSN: 1536-1284.

[104] D. Thangavel et al. "Performance evaluation of MQTT and CoAP via a common middleware". In: *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*. IEEE. 2014, pp. 1–6.

[108] I. Unwala et al. "Thread: An IoT protocol". In: *2018 IEEE Green Technologies Conference (GreenTech)*. IEEE. 2018, pp. 161–167.

[109] X. Wang et al. "An Efficient Named-Data-Networking-Based IoT Cloud Framework". In: *IEEE Internet of Things Journal* 7.4 (2020), pp. 3453–3461.

[111] J. Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.

[112] M. Willocx et al. "Developing Maintainable Application-Centric IoT Ecosystems". In: *2018 IEEE International Congress on Internet of Things (ICIoT)*. San Fransisco, CA, USA, July 2018, pp. 25–32.

[113] M. Willocx et al. "QoS-by-Design in reconfigurable IoT ecosystems". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE. Limerick, Ireland, 2019, pp. 628–632.

[114] S. S. Wong et al. "Smart applications to track and record physical activity: implications for obesity treatment". In: *Smart Homecare Technology and TeleHealth* 2014.1 (2014), pp. 77–91.

[115] B. Wukkadada et al. "Comparison with HTTP and MQTT in Internet of Things (IoT)". In: *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. IEEE. 2018, pp. 249–253.

[116] F. Yang et al. "μPnP: Plug and Play Peripherals for the Internet of Things". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. ACM, 2015, pp. 1–14. ISBN: 978-1-4503-3238-5.

[117] J. Yin et al. "A survey on Bluetooth 5.0 and mesh: New milestones of IoT". In: *ACM Transactions on Sensor Networks (TOSN)* 15.3 (2019), pp. 1–29.

[118] A. Yousefpour et al. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Journal of Systems Architecture* 98 (2019), pp. 289–330.

[119] S. Zeadally et al. "25 years of Bluetooth technology". In: *Future Internet* 11.9 (2019), p. 194.

[121] R. Zgheib et al. "Engineering IoT Healthcare Applications: Towards a Semantic Data Driven Sustainable Architecture". In: *eHealth 360°*. Ed. by K. Giokas et al. Springer International Publishing, 2017, pp. 407–418. ISBN: 978-3-319-49655-9.

Technical Reports

[3] *Activiteitenverslag 2021-2022*. Tech. rep. Bpost Group, 2021.

[6] *AN1142: Mesh Network Performance Comparison*. Tech. rep. Silicon Laboratories, Inc., 2018.

[26] *Concise binary object representation (cbor)*. Tech. rep. Internet Engineering Task Force, 2013.

[45] *Going Hybrid in Industrial IoT - How a Holistic Data Placement Strategy Solves the Edge versus Cloud Decision*. Tech. rep. Siemens Advanta, 2022.

[81] *Prolog ISO/IEC 13211-1:1995*. Tech. rep. International Organization for Standardization, 1995.

[105] *The Internet of Things: Mapping the value beyond the hype*. Tech. rep. McKinsey Global Institute, 2015.

[107] *Thread Network Fundamentals*. Tech. rep. Thread Group, 2020.

[110] *Wi-Fi 6 and Matter: The Real Plug and Play Smart Home*. Tech. rep. Qorvo US, Inc.

Online Resources

[1] *4 Google smart home updates that Matter*. May 19, 2021. URL: https://blog.google/products/google-nest/four-google-smart-home-updates-matter/ (visited on 05/11/2022).

[4] *Add support for Matter in your smart home app*. 2021. URL: https://developer.apple.com/videos/play/wwdc2021/10298/ (visited on 05/12/2022).

[9] *Blynk IoT platform: for businesses and developers*. 2015. URL: https://blynk.io/ (visited on 11/30/2020).

[24] *Colruyt opent 24/7 stadssupermarkt zonder personeel*. URL: https://www.tijd.be/ondernemen/retail/colruyt-opent-24-7-stadssupermarkt-zonder-personeel/10344863.html (visited on 05/09/2022).

[25] *Complete Philips Hue range to be compatible with new smart home connectivity standard Matter*. May 12, 2021. URL: https://www.signify.com/global/our-company/news/press-releases/2021/20210512-complete-philips-hue-range-to-be-compatible-with-new-smart-home-connectivity-standard-matter (visited on 05/12/2022).

[31] *DeviceHive Open Source IoT Data Platform*. 2013. URL: http://devicehive.com (visited on 11/30/2020).

[36] *eXtensible Access Control Markup Language (XACML) version 3.0*. 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (visited on 09/05/2022).

[44] *General Data Protection Regulation (EU) no 2016/679*. 2016. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 09/05/2022).

[46] *Google Logica*. 2021. URL: https://opensource.google/projects/logica (visited on 05/05/2022).

[50] *Home Assistant - Open source home automation that puts local control and privacy first*. 2020. URL: https://www.home-assistant.io/ (visited on 11/30/2020).

[52] *How IoT Impacts Data and Analytics*. 2018. URL: https://www.gartner.com/smarterwithgartner/how-iot-impacts-data-and-analytics/ (visited on 09/01/2021).

[55] *IoT Analytics - ThingSpeak Internet of Things*. 2017. URL: https://thingspeak.com/ (visited on 11/30/2020).

[64] *macchina.io - Control and manage your devices with a secure, private connection*. 2015. URL: https://macchina.io (visited on 11/30/2020).

[69] *Manifesto for agile software development*. 2001. URL: http://agilemanifesto.org/ (visited on 09/06/2022).

[74] *MQTT*. URL: https://www.mqtt.org (visited on 09/05/2022).

[79] *openHAB - a vendor and technology agnostic open source automation software for your home*. 2015. URL: https://www.openhab.org/ (visited on 11/30/2020).

[80] *Predix Platform | Industrial Cloud Based Platform*. 2018. URL: https://www.ge.com/digital/iiot-platform (visited on 11/30/2020).

[88] *Samsung SmartThings Integrates Matter Into Ecosystem, Bringing Matter Device Control to Multiple Samsung Products*. Oct. 27, 2021. URL: https://news.samsung.com/global/samsung-smartthings-integrates-matter-into-ecosystem-bringing-matter-device-control-to-multiple-samsung-products (visited on 05/12/2022).

[96] *Siemens MindSphere - Connecting the things that run the world*. 2020. URL: https://siemens.mindsphere.io/ (visited on 11/30/2020).

[98] *SMALT: The World's First Interactive Centerpiece*. URL: https://www.indiegogo.com/projects/smalt-the-world-s-first-interactive-centerpiece#/ (visited on 05/05/2022).

[101] *Support for Matter over Thread coming to Echo and eero devices*. Nov. 3, 2021. URL: https://developer.amazon.com/en-US/blogs/alexa/device-makers/2021/11/support-for-matter-over-thread-coming-to-echo-and-eero-devices.html (visited on 05/11/2022).

[103] *Tau Prolog - An open source Prolog interpreter in JavaScript*. 2020. URL: http://tau-prolog.org/ (visited on 11/30/2020).

[106] *This Smart Salt Shaker Wants to Change the Way You Season Food*. URL: https://time.com/4773835/smalt-salt-shaker-bluetooth/ (visited on 05/05/2022).

[120] *Zetta - An API-First Internet of Things (IoT) Platform*. 2016. URL: https://www.zettajs.org/ (visited on 11/30/2020).

List of Publications

Main publications

1. I. Bohé, M. Willocx, and V. Naessens. "An extensible approach for integrating health and activity wearables in mobile IoT apps". In: *2019 IEEE international congress on Internet of Things (ICIoT)*. IEEE. Milan, Italy, 2019, pp. 69–75
2. I. Bohé, M. Willocx, and V. Naessens. "Untangling the Physical-Digital Knot When Designing Advanced IoT Ecosystems". In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things (M4IoT)*. UC Davis, CA, USA, 2019, pp. 1–6. ISBN: 9781450370288
3. I. Bohé, M. Willocx, and V. Naessens. "SMIoT: a software architecture for maintainable internet-of-things applications". In: *International Journal of Cloud Computing* 9.1 (2020), pp. 75–94
4. I. Bohé, J. Gardeyn, L. Cuypers, J. Lapon, M. Willocx, and V. Naessens. "A Crowdsensing Solution for Tracking Bicycle Path Conditions". In: *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE. New Orleans, LA, USA - Online, 2020, pp. 1–6
5. I. Bohé, M. Willocx, J. Lapon, and V. Naessens. "Towards low-effort development of advanced IoT applications". In: *Proceedings of the 8th International Workshop on Middleware and Applications for the Internet of Things (M4IoT)*. Québec, Canada - Online, 2021, pp. 1–7
6. I. Bohé, M. Willocx, J. Lapon, and V. Naessens. "A Logic Programming Approach to Incorporate Access Control in the Internet of Things". Accepted at IFIP IoT 2022. Amsterdam, the Netherlands

Papers with contributions as co-author

1. M. Willocx, I. Bohé, J. Vossaert, and V. Naessens. "Developing Maintainable Application-Centric IoT Ecosystems". In: *2018 IEEE International Congress on Internet of Things (ICIoT)*. San Fransisco, CA, USA, July 2018, pp. 25–32
2. M. Willocx, I. Bohé, and V. Naessens. "QoS-by-Design in reconfigurable IoT ecosystems". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE. Limerick, Ireland, 2019, pp. 628–632

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
DISTRINET@GENT
Gebroeders de Smetstraat 1
B-9000 GENT
ilse.bohe@kuleuven.be
https://iiw.kuleuven.be/onderzoek/distrinet