UCLouvain

epl

École polytechnique de Louvain

Sensors Integration in Feature-Based Context-Oriented Programming

Author: **Béatrice Lambert**
Supervisors: **Axel Legay, Kim Mens, Benoît Duhoux**
Readers: **Axel Legay, Kim Mens, Benoît Duhoux, François de Keersmaeker**
Academic year 2023–2024
Master [120] in Computer Science

Abstract

Mobile and ubiquitous have expanded the range of context-aware applications. while the *Feature-based context-oriented programming* (FBCOP) paradigm provides support to develop such applications, it is not able to perceive the application surrounding contexts. This master thesis aims to extend the RubyCOP framework that implements the FBCOP paradigm, to propose a sensor layer that identifies and activates the application surrounding contexts. We used ontology modeling to design this sensor layer in order to allow the developer to provide the logic which is essential for the application context identification. The sensor layer we propose also includes a validation process that avoids the need for unreasonable context switching. The proposed sensor layer implementation has been validated with real sensors as part of a Smart Home application.

Acknowledgments

First of all, I would like to thank my supervisors Axel Legay, Kim Mens and Benoît Duhoux for their availability and their guidance throughout this year, as well as for providing this master thesis topic. A special mention goes to our long meetings, which served as checkpoints and motivators throughout the year. I would also like to thank François de Keersmaeker for agreeing to be part of my jury and this master thesis' readers.

I would also like to express my gratitude to my parents, friends and my boyfriend for supporting me and believing in me throughout this project. Thanks to the "Réaumur" group for brightening up my master's years and taking part in my brainstorming sessions.

From a technical point of view, I would like to thank Deepl and Chatgpt for their help in improving the readability of my master thesis.

Contents

1 Introduction **1**
1.1 Problem and motivation 1
1.2 Objectives 2
1.3 Contributions 3
1.4 Running example 3
1.5 Roadmap 4

2 Background material **5**
2.1 Context-oriented programming 5
2.2 Feature-based context-oriented programming 6
2.3 RubyCOP 8
2.4 Ontology and context-awareness 9

3 Related work **10**
3.1 A context-oriented software architecture 10
3.2 OpenHAB 11

4 Problem statement **12**
4.1 Sensor and data types 12
4.2 Sensor interaction with RubyCOP 13

5 Solution **15**
5.1 Architecture 15
5.1.1 Sensor data retrieval 16
5.1.2 Logical part 17
5.2 Implementation 18
5.2.1 Sensor data retrieval 18
5.2.2 Logic part 23
5.2.3 Simulator 30

6 Validation through the case study **32**
6.1 Smart Home design and implementation 33
6.1.1 FBCOP design 33
6.1.2 Sensor layer 39
6.2 Testing the sensor layer 45
6.2.1 Simulator 46
6.2.2 Physical sensors 47
6.2.3 Code quality 47
6.3 Results 48
6.3.1 Strengths 48
6.3.2 Weaknesses 49
6.4 Threats to validity 50

7 Conclusion **51**
7.1 Conclusion 51
7.2 Future work 52
7.2.1 Extend data type used 52
7.2.2 Extend the possible actions of the rules 52
7.2.3 Enable sensor data validation 53

Chapter 1

Introduction

Feature-based context-oriented programming (FBCOP) is a paradigm that has emerged in the field of context-aware application development [Duh22]. It offers the developer a way to build a dynamic application that adapts its behavior to its environment at runtime. RubyCOP is a framework that implements such a paradigm by providing support to the developer to build such applications. In this framework, the developer must define the theoretical contexts to which the application will adapt its behavior.

This chapter begins by outlining the problem and motivation behind this master thesis, section 1.1. In section 1.2, we define the objectives of this work in the form of research questions. Following that, section 1.3, presents the contributions of this master thesis. In section 1.4 we introduce the chosen running example that will follow the reader throughout this work. Finally, in section 1.5, we provide an overview of each chapter of this document with its specific goals.

1.1 Problem and motivation

Currently, the framework does not provide support to detect the application's surrounding contexts. These can only be updated manually or through the application itself. The framework enables the development of an application whose context can be updated through user interaction, for example using a form to find out the user's age or visual disabilities such as color blindness, to adapt the application's behavior to the user. The number of application contexts that can be updated to represent the surrounding context of the application is therefore limited to those detectable by user intervention.

This master thesis aims to extend RubyCOP with a sensor layer that provides support to the developer to be able to perceive and identify each application context depending on its surrounding environment. This sensor extension must be able to work with sensors by collecting their raw data and must be able to reason about them to automatically update application contexts according to the developer's requirements. Our goal is to improve the dynamic change of contexts of the framework by enabling the use of sensors. This way the only limit will be the developer's creativity (and the budget for sensors).

1.2 Objectives

In this master thesis, we set ourselves several objectives that represent our expectations for the implementation of this RubyCOP sensor layer. We have defined these objectives in the form of four research questions.

RQ1. *How can we design a sensory layer in the FBCOP framework?*

Developing a new layer of a framework may require the introduction of new components that we want to interact with the existing components of the current framework. In the case of developing a sensor layer for the RubyCOP framework, we need to decide how we will define the sensors and the logic part of the layer. This logic part will reason about sensor data to be able to perceive and update application contexts using current RubyCOP components.

RQ2. *How can we design and implement an architecture to enable interactions between physical sensors and the FBCOP framework?*

Interacting with sensors involves several challenges such as sensor identification but also data communication. We want our sensor layer to be able to receive the data collected by the sensor but also identify the sensor itself to be able to reason over the collected data.

RQ3. *How to work with continuous, discrete and event-driven physical sensor data in this extension of the framework FBCOP?*

The strength of sensors depends on their precision and versatility. However, it is important to bear in mind that the data from these sensors can take several forms. We want our sensor layer to be able to work with sensors that provide continuous, discrete and event-driven data.

RQ4. *Does the extension answer to realistic problems?*

We do not want to simply develop an extension to the framework. We want the solution we propose to solve realistic problems to add value to the framework.

1.3 Contributions

This master thesis describes our proposed implementation of the RubyCOP framework's sensor layer. It shows how we managed to design a layer that can interact with both physical sensors and RubyCOP's components.

We were able to validate the support provided by this sensor layer through the implementation of our running example and the use of real sensors. These sensors were chosen to collect the data needed to identify the contexts of our application, but also to demonstrate the fact that the sensor layer can handle continuous, discrete and event-driven data.

The sensor layer implementation can be found on the repository: `https://bitbucket.org/benoitduhoux/rubycop_lambert/src/sensor-layer-ontology/`

1.4 Running example

We have defined a case study that will be used throughout this thesis to illustrate our thinking and the choices we made while developing the sensor layer. We naturally chose an application that simulates a Smart Home as a running example. It is an FBCOP application that dynamically adapts the application's behavior to the surrounding context based on physical assumptions made for this Smart Home application within the scope of FBCOP.

We assume that a house is equipped with solar panels and a circuit breaker for the lighting circuit. When the circuit breaker is open, there is no electricity on the lighting circuit making it is impossible to turn on the house lights. On the contrary, when the breaker is closed, electricity flows through the circuit and the inhabitant can turn the house lights on and off at will, exactly as he would have done in a conventional home. Additionally, it has home automation tools to manage radiators, opening and closing windows as well as Wi-Fi management. Furthermore, it enables the launch of major appliance programs. In this case study we will limit ourselves to a dishwasher and a washing machine as major appliances. Finally, the house is inhabited by several people.

The aim of this application can be divided into two aspects. The first aspect is ecological since the aim is to optimize the use of the green energy generated by the solar panels. The second goal is to enhance the comfort of the house's inhabitants by monitoring and regulating the house's temperature.

1.5 Roadmap

In the remainder of this master thesis, we begin with chapter 2 where we will introduce the background material which is useful to understand this document. Then in chapter 3, we present related work on sensor layer development, such as a sensor layer architecture for the FBCOP paradigm, or the use of sensors in context-aware applications. Next, in chapter 4, we describe the challenges of this master's thesis in detail.

Now that we introduced everything we need to know and we described the challenges of this work, we present our solution in chapter 5 where we describe the architecture and the implementation of our sensor layer. Then, in chapter 6, we depicted the validation of our sensor layer implementation through the running example as well as the analysis of the sensor layer strengths and limitations.

The document ends with a conclusion, chapter 7, where we will also talk about the future work we have identified to enhance our proposed sensor layer.

Chapter 2

Background material

This chapter outlines all the theoretical information used in this master thesis. We begin by explaining what context-oriented programming is and its purpose, in section 2.1. Then we explain what feature-based context-oriented programming is, and its similarities and differences with context-oriented programming in section 2.2. Next, in section 2.3, we present the RubyCOP framework. Lastly, in section 2.4, we present what is an ontology in the scope of context-aware systems.

2.1 Context-oriented programming

Context-oriented programming (COP) is a programming paradigm that offers a new approach to application development, particularly useful for context-aware systems [HCN08]. These systems emerged with the development of mobile and ubiquitous computing. Mobile computing represents a high level of mobility of computing services. Indeed, our computers are part of our daily lives and tend to be taken for granted. On the other hand, ubiquitous computing has not only a high level of mobility but also a high level of embeddedness [LY02]. Context-aware systems adapt their behavior based on their environment without requiring user interaction, unlike traditional computing, which needs user input through an input device [LLH11]. COP is designed to support the developer in the development of context-sensitive applications by enabling context-dependent behavioral variations.

To develop such applications, understanding the fundamental concepts of the COP paradigm is essential [SGP12]. One first fundamental concept is the notion of *context* which represents the surrounding environment. Context can represent outside states of the program system such as the weather or internal states of the system like battery percentage. It can also represent user preferences, age or even potential disabilities. The range of possible contexts is vast and diverse. From a

practical point of view, the concept of context encompasses any information that can be accessed computationally. In fact, as in context-aware systems, contexts can be detected using sensors, among other things. We can use sensor combinations to define more complex and realistic contexts. Another important point is the multiplicity of possible contexts. Indeed, we can combine partial contexts to represent a more complete and realistic environment in a flexible way. The second concept defined by both Salvaneschi et al. [SGP12] and Hirschfeld et al. [HCN08] is *behavioral variation*. This concept represents the variation part of the paradigm. Behavioral variations are partial behavior definitions that can take the form of partial definitions of modules or even complete class definitions. These fragmentary implementations can modify or even add new functionalities to the general behavior of the application. As for contexts, these *behavioral variations* can be dynamically combined. All these are organized by *layers* that aggregate behavioral variations that are concerned by the same context.

Another important aspect of COP is its *proceed* mechanism. In this paradigm, it is possible to have several layers active at the same time and these lead to an adaptation of the application's behavior. These layers can refine common functionalities or even replace the default behavior. These are partial implementations and have to be dynamically combined to provide the complete application behavior. The *proceed* mechanism allows layers to be combined similarly to the *super* keyword in object-oriented (OO) languages [HCN08]. It establishes the link between the different layers while the super keyword establishes the link between the objects and their parents. The behavior of supercalls in OO languages remains constant throughout the lifetime of the application. The behavior of the *proceed* mechanism depends on the arrangement of the sensor layers which can vary throughout the lifetime of the application since it depends on the contexts perceived by the application.

A COP application adapts its behavior at runtime according to its current environment. The critical aspect is the runtime adaptation, which allows source code variation based on the actual context without recompilation. To enable runtime adaptation, we need to define the *de/activation* concepts. It is the part that links predefined contexts and their associated layers. It can enable or disable layers depending on the actual context at runtime.

2.2 Feature-based context-oriented programming

Feature-based context-oriented programming (FBCOP) is a paradigm that takes advantage of COP and feature modeling (FM) [Duh22]. This paradigm also offers

support to developers in implementing context-aware applications, but unlike COP, it makes an explicit distinction between contexts and features. This distinction was inspired by Hartmann et al. [HT08]. They proposed to divide a system into a context variability model and a feature model where the context variability model constrains the feature model.

FBCOP models the contexts and features in a tree-like structure which are respectively named the *context model* and the *feature model*. This structure takes into account the hierarchy between the nodes as in a basic tree structure, but they also have four hierarchical relationships possible: *mandatory*, *optional*,*or* and *alternative* [Duh22]. Figure 2.1 illustrates those hierarchical relations. The *mandatory* constraint between a node and its child, *FeatureModel* and *HeavyDevicesController* in the example, means that if the parent is present then the child must be present in the system. They are always present together. The *optional* constraint, *FeatureModel* and *RadiatorsController* in the example, means that the child may be present if the parent is present in the system. The *or* constraint, *Lights* and *MainLightsController*, *SmallLightsController* in the example, means that at least one child is present if the parent is. The last hierarchical constraint is the *alternative*, *HeavyDevicesController* and *HeavyDevicesOn*, *HeavyDevicesOff*, which means that if the parent is present in the system there is exactly one child present. When we add an *alternative* constraint, we can define a default child. In addition to those hierarchical constraints, the model also has two cross-tree constraints: *exclusion* and *requirement*. They can be defined between nodes that may not have a hierarchical relation. To ensure the validity of these models, all of the model constraints must be satisfied.

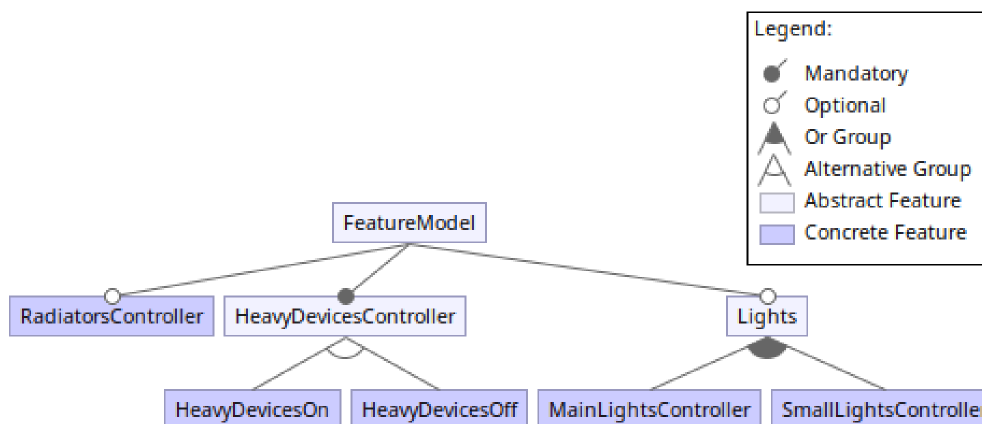

Figure 2.1: Feature model inspired from the case study Smart Home that depicts hierarchical relations.

The context model and feature model are connected through a *Context-Feature mapping model* designed to identify which features should be activated when contexts are activated [DMD22].

2.3 RubyCOP

RubyCOP is an application programming framework that adheres to the underlying principles of FBCOP [DMD22]. It provides support to the developer to implement FBCOP applications. The developer only needs to define the *context model*, *feature model* and the *context-feature mapping model* using the framework and RubyCOP will manage the runtime behavior adaptations based on the contexts present in the define *context model*.

It also supports developers in feature development by offering specific language constructs, such as *set_prologue* and *set_epilogue*. The *set_prologue* construct defines the methods that must be executed when the feature is activated, while the *set_epilogue* construct defines the methods to be executed before the feature deactivation. Since FBCOP is built on top of COP, the *proceed* mechanism is also part of RubyCOP. The whole process of feature selection, activation, and model validity checking is abstracted by the framework since it is based on the three models defined by the developers.

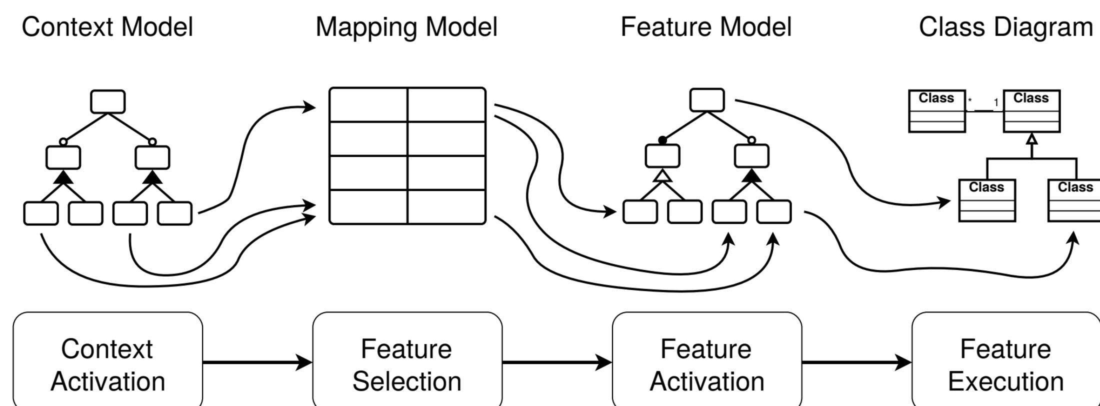

Figure 2.2: Overview of the FBCOP's system architecture [DMD19]

Figure 2.2 represents the overview of the FBCOP's system architecture [DMD19]. The control flow of this approach starts with the *context model* and more precisely with the *Context Activation* component. This component is responsible for de/activating contexts following a request. When it receives a request to de/activate

a set of contexts, it attempts to make the change in the context model while checking that the model's constraints are respected. If this is not the case, there is a rollback function that undoes the changes made to the model. If the *context model* is still valid after the modifications, then the process may continue. The second component, *Feature Selection*, is responsible for selecting the features to de/activate depending on the context changes. This selection is made by looking at the *mapping model* defined by the developer. Then it is the *Feature Activation* component's turn to take the lead. Its process is similar to the *Context Activation* component, except that here we are looking to de/activate features instead of contexts. At this stage, a rollback is also possible if the modifications made to the *feature model* render it invalid to its constraints. The last step is managed by the *Feature Execution* component. It is responsible for un/deploying the features selected in the previous step.

2.4 Ontology and context-awareness

Ontology is a formal representation that is used to represent concepts and their relationships [CC12]. Using an ontology to detect contexts from raw data is a good idea according to several points [KM03]. First of all, we can adapt ontology modeling to the desired domain, but also to the desired level of complexity. Then, it is also useful in the scope of context-aware applications since it is flexible but also expandable. Adding new contexts in the ontology is fully supported. Finally, it can still be interpreted by humans.

Korpipää et al. [KM03] proposed some properties that would be interesting while defining such contexts. One important property is *context* is a symbolic value and often a verbal description. The other is the *value* that corresponds to a numerical value. Then ontological contexts may also have *confidence* property to manage the uncertainty of the context source. The context source may also be defined as a property.

Another advantage of using ontology modeling in context-aware systems is that it facilitates any inference thanks to its design. It does not require a central inference mechanism. Everything can be managed by the contexts properties as explained with the *confidence* property. The ontology itself is capable of performing the context recognition process.

Chapter 3

Related work

This chapter presents two main related works of this master thesis. The first one, in section 3.1, presents a proposed FBCOP architecture with the sensor layer. The second one, in section 3.2, presents a software that provides support to manage home automation devices.

3.1 A context-oriented software architecture

In many context-oriented programming languages, context discovery is often overlooked since they set their focus on behavioral variation logic. Mens et al. have proposed a context-oriented software architecture that includes sensor-based context detection [MCD16]. The layered architecture proposed is composed of four layers where the first three layers constitutes the *adaptation framework*.

The *interaction layer* is the one responsible for gathering information from the system's context which can be about the physical environment (e.g., the outside temperature), the user (e.g., user age) or even about inner components of the device (e.g., battery charge level). These pieces of information can be gathered in two different ways. The first one defined in the paper is the *User Input* where the information is gathered from the user. It is often used to gather user preference pieces of information. The second one is the use of *Sensors*. Contextual information is gathered through physical devices.

The *discovery layer* aims to extract the representative context by interpreting and reasoning over the information collected by the *interaction layer*. The *Interpretation* component is responsible for filtering the received information with the filters provided by the programmer. The *Reasoning* component is the one that determines

the set of contexts relevant to the application and thus adapts the application's behavior by selecting, activating and executing the corresponding feature variants.

The *handling layer* is the one in charge of activating the contexts selected by the *discovery layer*. This layer is also responsible for verifying if the dependencies that exist between the contexts and their features are respected to adapt the system's behavior consistently. The layer is divided into two sublayers. The first one, *context handling* is responsible for deciding which contexts get activated. The second one, *feature handling* is responsible for managing the selection, activation and execution of features.

The fourth layer, the *application layer*, lies outside the scope of the *adaptation framework*. It contains the implementation components provided by the application programmer. These include *listeners* that intercept data from the user or sensors, *filters* responsible for filtering out the data received by the *discovery layer*, *context declaration* where the developer specifies details such as the context's name and its dependent sensors, and *feature handling* which contains the context-feature mapping, feature declarations and other important components to build context-aware applications.

3.2 OpenHAB

OpenHAB[1] is an open-source automation software for home automation. It is a scalable software that combines various technologies with home automation systems to provide uniform user interfaces [AP19]. The client can use OpenHAB to provide rules to coordinate the smart home devices. OpenHAB aims to be easy to use but also to be able to run everywhere thanks to its compatibility with several OS (Linux, Windows and iOS) and main devices (computer or even Raspberry Pi). Its strength is also its interoperability [AP19].

This tool is a good example of how we can adapt a smart device's behavior according to the logic provided by the client. The client first needs to define its devices. The devices are represented as *things* and their states are called *items*. Then he can add the logic to monitor those smart devices thanks to the automation rules provided by OpenHAB. This work helped us to identify the important points we wanted for our sensor layer and how to propose them by isolating the logic. This master thesis aims to not be limited to home automation applications thanks to the FBCOP paradigm.

[1] `https://www.openhab.org/`

Chapter 4

Problem statement

This chapter brings together all the challenges aimed to be overcome when adding a sensory layer to the RubyCOP framework. On the one hand, there is a useful framework for context-aware applications, but it still requires user intervention to update the application's context and fully leverage its capabilities. On the other hand, we have sensors, each with their characteristics and requirements. This master's thesis aims to bridge the gap between physical or virtual sensors and the RubyCOP framework, enabling the representation of coherent and dynamic contexts.

The first section 4.1 addresses the challenges we face due to the sensors themselves and their possible data types. It gathers the challenges encountered when working with physical sensors, as well as the methods for retrieving logical information from them. The second section 4.2 states the challenges related to sensor interaction with the framework from data collection to context change requests.

4.1 Sensor and data types

As we want to use sensors to be able to perceive contexts dynamically, we need to be aware of the different types of sensors likely to be used by developers of context-aware applications and their specificities. This understanding is crucial to make our sensory layer useful to as many applications as possible. This way, the developers will not be limited in their choice of sensors.

There exist several kinds of sensors. We divide them according to their behavior and their data types. We chose those criteria since these are the ones that will have an impact on the sensory layer's development. According to Ricquebourg et

al., we can identify four possible sensor modes for measuring and communicating data [RMM+].

- *Uninterrupted* mode: when sensors that measure and send their data following a frequency that can stay constant or dynamically modified.
- *By event* mode: when sensors send data only when an event occurs. For these sensors, the meaningful information includes not only the potential data but also the event that triggered the sensor.
- *On request* mode: when sensors provide data in response to a request. That request is done by an external entity which waits until it receives the sensor response.
- *Hybrid mode*: this mode is a combination of the three modes described above.

This master thesis's first challenge is to develop an extension that is compatible with as many types of sensors as possible.

4.2 Sensor interaction with RubyCOP

Now that we identified the characteristics of the sensors we are likely to interact with, we still need to identify the challenges that lie ahead. As we are developing a sensory layer for the RubyCOP framework, it is crucial to contain all relevant logic within this layer. Therefore, we must limit our expectations of sensors to their core functions: measuring and sending data. The responsibility for sensor identification and any additional processing must reside within the sensory layer. These challenges range from handling the raw data provided by the sensors to ensuring seamless interaction with the framework.

Sensor data acquisition

One of the first challenges when working with sensors is sensor data collection. There are two aspects to this challenge. The first one is about the choice of a communication mechanism that integrates well with both the sensors and the framework without incurring excessive additional communication costs.

The second challenge involves the identification of sensors by the framework, which is a mandatory step for interpreting and reasoning about the sensor data. It is essential to determine which specific sensor provided the data in order to analyze and utilize it effectively.

Another challenge about sensor data acquisition concerns the various possible sensor modes we have defined earlier in section 4.1. We need to define an architecture for our sensory layer that works equally well with sensors that operate *on request*, *by event* or in *Uninterrupted* mode and by consequence therefore also by hybrid mode.

Data management

Once the data collection is done, some challenges remain. Indeed, once we have collected the sensor data, we need to extract logical information from it. This allows us to select the most appropriate corresponding context. Moreover, this process must be aware that it is working with raw data. The data must be filtered first to keep only valid data. An example of filtering may consist of ignoring or adapting the logic behavior when a data item is encountered that does not belong to the value domain of the corresponding sensor.

Context update

Once the logical part is done and we have identified the context perceived by our sensors, we must notify the framework. This allows the framework to adapt its behavior in consequence. The challenge here lies in notifying the framework. Indeed, context changes should not be as frequent as sensor measurements and therefore their semantic evaluation. This would risk flooding the framework with notifications of context changes.

Chapter 5

Solution

In this chapter, we introduce the architecture, in section 5.1, and implementation, in section 5.2 of our sensory layer. Currently, the FBCOP paradigm requires developer interaction to de/activate contexts. The objective of the sensor extension proposed is to augment the FBCOP control flow by identifying and activating the appropriate set of contexts that represent the application's actual environment. This is accomplished by processing sensor raw data and applying the logic defined by the developer. This approach enables the development of fully autonomous applications that instantaneously adapt their behavior based on their environment. The sensor layer we propose is positioned upstream of the *Context Activation* component within the FBCOP system architecture.

5.1 Architecture

In this section, we will describe the architecture and the concepts of our sensory layer. This architecture was inspired by ontology modeling, see section 2.4 and by the proposition of Mens et al. depicted in section 3.1. We will divide the architecture into two parts. The first one, see subsection 5.1.1, presents how our sensor layer retrieves data from sensors while the second part, see subsection 5.1.2, is responsible for reasoning over the collected data.

If we connect these two parts with the architecture proposed by Mens et al. [MCD16], we can say that the first part represents the *interaction layer* since it is responsible for collecting the sensor's data. The other part represents the *discovery layer* and interacts with the first component of the *handling layer*: the *Context activation* component.

Ontologies are used to represent objects and the relationships between them. In our case, the objects are on one side the sensors and on the other side we have the contexts defined in the *Context model* of the FBCOP's application.

Figure 5.1 represents an overview of the architecture of the proposed sensor layer. Each of these components, as well as their interactions and objectives, are described in subsection 5.1.1 and subsection 5.1.2. The physical sensors responsible for measuring the data needed by the sensor layer to perceive the appropriate contexts are located outside the application, represented by the blue zone. In the application, we find the architecture of the sensor layer and the FBCOP components with which the sensor layer interacts.

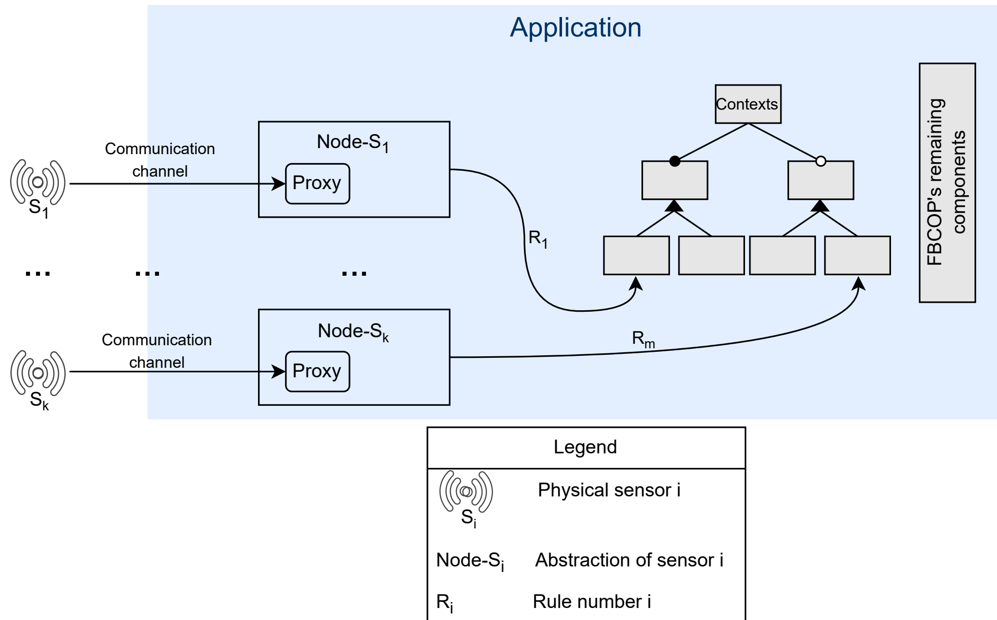

Figure 5.1: Sensor layer architecture overview. Gray components come from the RubyCOP framework.

5.1.1 Sensor data retrieval

The first part of our sensor layer architecture is responsible for collecting sensor data to gather information about the application environment. Perceiving each context in an FBCOP application often requires multiple sensors, so our architecture must be able to work with several sensors.

Following the ontology idea, each physical sensor will be modeled by a node of the ontology model which we will call the sensor abstraction. To be able to retrieve data from physical sensors, we have added a proxy to each of those sensor nodes. We preferred to add a proxy for each sensor to solve the sensor identification problem, as we wanted to integrate as much logic as possible into our sensor layer. These proxies will act as listeners which are responsible for collecting data from sensors. The proxies do not contain logic, their only goal is to collect data and verify if the data is exploitable by the sensor abstraction. These proxies solve the sensor identification problem as each sensor will only have to send its data to its corresponding sensor proxy.

5.1.2 Logical part

Now that we have modeled each sensor and established how their data will be collected, we need to add logic for the reasoning component of the *discovery layer*. According to the ontology definition, the relationships, and thus the logic, between the sensor abstractions and the defined contexts are represented using the edges. Each edge of our architecture represents a rule that takes sensor abstractions as input and can influence the application's contexts de/activation.

We want our sensor layer to leave it up to the developer to provide their logic for selecting contexts. To incorporate the logic provided by the developer, we have defined a basic reasoning that the developer's logic will complete. The rule's reasoning skeleton is depicted in Equation 5.1 where $\mathcal{C}_a$ and $\mathcal{C}_d$ are provided by the developer. $\mathcal{C}_a$ ($\mathcal{C}_d$ respectively) represents a set of contexts that the rule will activate (deactivate) when the condition is verified.

$$\textbf{if } \text{condition } \textbf{then} \quad \textbf{activate } \mathcal{C}_a \textbf{ and deactivate } \mathcal{C}_d \tag{5.1}$$

The *condition* takes the form of an inequation where *threshold* is a constant inherent to the rule. On the other hand, *value* represents the variability of the context depending on the data sensed by the sensors of the sensor layer. It can directly take the value of one of the sensors or represent the result of a calculation that takes more than one sensor as input.

$$\text{value} > \text{threshold} \tag{5.2}$$

The context activation and deactivation processes are part of the FBCOP's paradigm itself. This is where the sensor layer intersects with the FBCOP paradigm.

5.2 Implementation

Before delving into the explanation of the implementation of this sensor layer according to the proposed architecture in section 5.1, we would like to discuss the language choice we made for this implementation. We chose to use the Ruby language to maintain continuity with the RubyCOP framework.

Figure 5.2 illustrates the class diagram of the sensor layer components proposed in this master thesis. The interaction of the sensor layer and the RubyCOP framework is also represented thanks to the grayed components of the class diagram. In the rest of this section, we will detail the implementation of these different components but also their interaction.

5.2.1 Sensor data retrieval

Sensor

Listing 5.1 depicts our class proposal to implement the sensor abstraction in the sensor layer. Each sensor instance has several attributes.

Among them, there is an ID attribute that is unique for each sensor instance. This attribute plays an important role in the rest of the sensor layer and this is why we do not let the developer choose the sensor's ID himself. The sensor's ID value initialization, line 6 is done thanks to the *global_id* class variable. A class variable is a variable that is shared by all instances of this class. Therefore, each time the developer creates a new sensor instance, its ID will be equivalent to the previous instance's ID increased by one.

According to the architecture defined in section 5.1, each sensor has its own proxy which is why we have chosen to instantiate the proxy at its corresponding sensor creation. The sensor proxy class is depicted in section 5.2.1.

The developer can optionally assign a name to the sensor instance to facilitate the development process. The sensor object also instantiates, lines 9 and 10, other attributes that will be useful later. The *value* sensor attribute can be read outside the class. We have taken this decision to allow the developer to use this data in his features if he wants.

To implement the interaction between our sensor abstraction and our rules, we used the *Observer design pattern*. When the sensor receives and updates a value, the *update_value* method is called and will notify all the rules thanks to

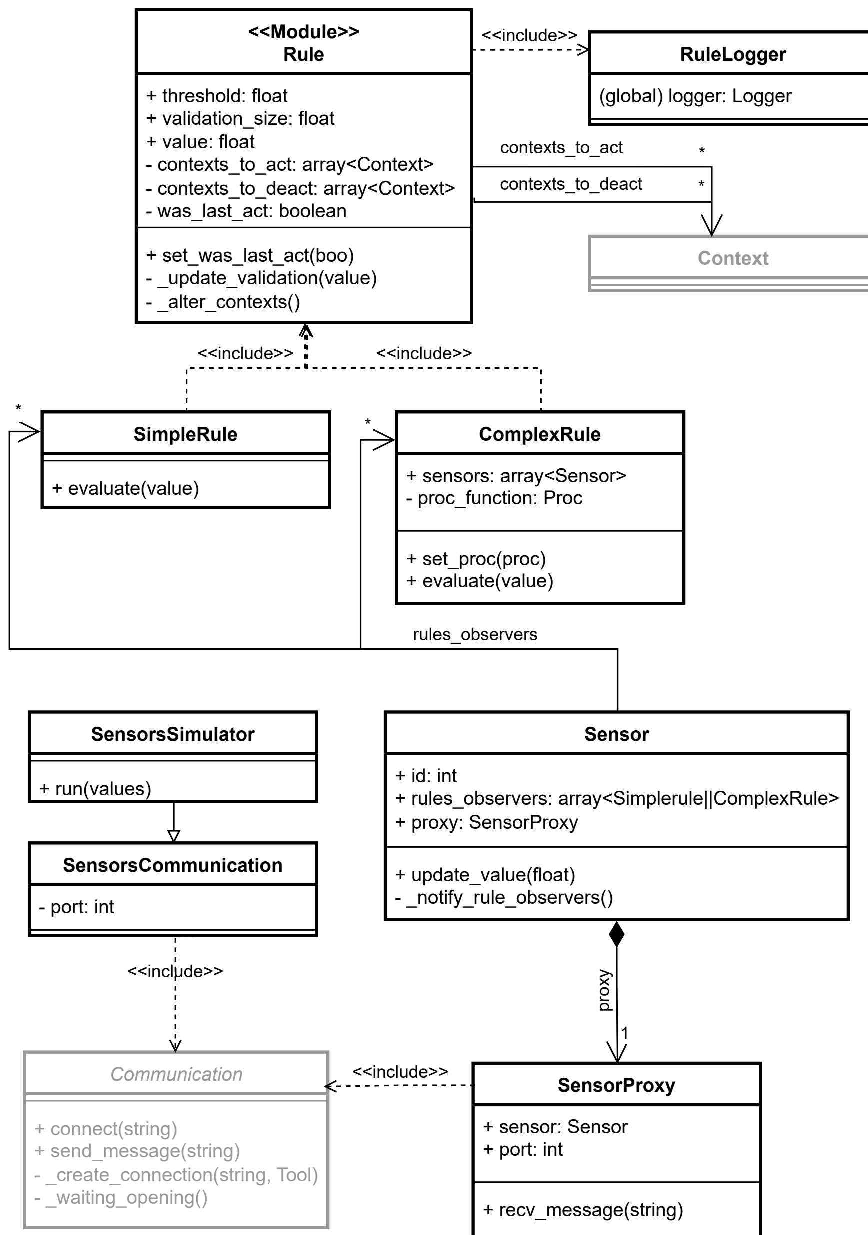

Figure 5.2: Class diagram of the sensor layer implementation of RubyCOP. Some information is not represented in the diagram for readability reasons. Rubycop's components have been grayed out to highlight those of the sensor layer.

the *_notify_rule_observers*. The rules can be added to the sensor using the *add_rule_observer* method.

```
class Sensor
    # attributes getters and setters
    @@global_id = 0
    def initialize(name = nil)
        @@global_id += 1
        @id = @@global_id
        @proxy = SensorProxy.new(self, "#{name}Proxy")
        @name = name
        @value = nil
        @rules_observers = []
    end
    def update_value(value)
        @value = value
        _notify_rule_observers()
    end
    def add_rule_observer(rule)
        @rules_observers << rule
    end
    # More code
    private
    def _notify_rule_observers()
        for rule in @rules_observers
            rule.evaluate(@value)
        end
    end
end
```

Listing 5.1: Sensor class in **sensor.rb**

Sensor proxy

The sensor proxy class is depicted in Listing 5.3. It includes the module *Communication*, see Listing 5.2, that is already provided by the RubyCOP framework. This module was initially developed for the communication between RubyCOP and the visualization tools. We have adjusted the *connect* method by adding the hostname argument. This preserves the use of localhost addresses while allowing the use of another communication address. This is important for our sensory layer since we want to be able to enable device-computer communication. The purpose of the module is to connect the instance to a pre-existing communication channel. The module has several methods already implemented but has only defined the

signature for the *recv_message* method which must therefore be completed.

```
module Communication
  def connect(hostname = 'localhost')
    if @ws.nil?()
      _create_connection(hostname)
      _waiting_opening()
    end
  end
  def send_message(data)
    # Send data if @ws is defined
  end
  def recv_message(msg)
  end
    # More code
  private
  def _create_connection(hostname, communication_tool=self)
      # creates a WebSocket connection
  end
  def _waiting_opening()
      # waits until the opening
  end
end
```

Listing 5.2: Communication module in **communication.rb** [Duh22]

The sensor proxy keeps the sensor responsible for its creation in its attribute to be able to communicate with it when the proxy receives new sensor data. Possible sensor port values start at 1111. The connection of the proxy to its corresponding channel is done in the instantiation process. Its corresponding channel is the computer's IP address since we want computer-device communication where the port is the value 1110 increased by the ID of the sensor that created the proxy. The proxy should not be the instance responsible for containing logic. Its sole purpose is to collect raw data from sensors and ensure that these are usable by the sensor instances. The *recv_message* method begins by casting the collected value as a float. It drops the data if the conversion is not possible after printing an error message. If the conversion of the value was a success, then the sensor proxy communicates the converted value to its sensor creator line 16.

```ruby
class SensorProxy
    include Communication
    # attributes getters and setters
    def initialize(sensor, name=nil)
        @sensor = sensor
        @port = 1110+sensor.id
        @name = name
        self.connect(Constants::IP_COMPUTER)
    end
    def recv_message(msg)
        begin
            value = Float(msg)
        rescue ArgumentError => e
            puts "Couldn't cast #{msg} into a float"
        else
            @sensor.update_value(value)
        end
    end
    # More code
end
```

Listing 5.3: SensorProxy class in **sensor_proxy.rb**

Channels instantiation

The last component we did not cover is the communication channel that the proxy uses to collect raw data from the sensors. RubyCOP framework already provides a communication channel class, see Listing 5.4. The class creates a channel of the *EventMachine* module provided by the Ruby language.

```ruby
class Channel
  attr_reader :channel, :tool, :port
  def initialize(tool, port)
    @channel = EM::Channel.new
    # arguments assignment
  end
end
```

Listing 5.4: Channel class in **channel.rb**

The initialization process of those channels is done in the *server.rb* of RubyCOP, see Listing 5.5. This file is already responsible for creating communication channels

for the visualization tools. We added lines 7 to 11 to create sensor channels based on the given arguments in the command line. The developer must specify the number of sensor channels required in the command line arguments to guarantee the correct operation of his application.

```
def create_channels()
    channels = [] # tools channels
    sensor_channels = []
    opts = Slop.parse() {
      |option|
        # other options here
        option.int '-s', '--nSensorChannels', 'Number of
           channels to create' do |value|
            for port in Array(1111...1111+value)
                sensor_channels << Channel.new("
                   SensorChannel-#{port}", port)
            end
        end
    }
    return channels, sensor_channels
  end
```

Listing 5.5: Sensor channels instantiation in **server.rb**

5.2.2 Logic part

The logical part is an important element of the sensor layer. It is responsible for reasoning about the data captured by the sensors and filtered by the upstream components of the sensor layer. Hereunder we describe the implementation part of the sensor layer responsible for the logic injection by the developer. This logic part is composed of a main block which is the Rule module (see Listing 5.6) which we will first describe. Then we will talk about how we implemented the two classes that the developer can use to add logic in this sensor layer. We will explain their differences but also how these choices were made so as not to limit the developer's logical possibilities.

Rules

The possible logic in a sensor layer is very varied and can take different forms. However, they all have common parts such as the initialization of some attributes or the de/activation of contexts. We chose to gather all the common parts of such logic into a module. We chose the module and not a parent class because this common part can not be used directly by the developer to inject logic into his

sensor layer. He needs one class that includes this module to be able to provide the logic he has defined. This implementation choice does not affect the extensibility of the logic part of the sensor layer. Indeed, even if there are currently only two classes using this module, it's perfectly feasible to expand the logic possibilities with more classes that would also use this module.

```
module Rule
    include RuleLogger
    def initialize(threshold, validation_size,
       contexts_to_act, contexts_to_deact)
        # parameters as attributes instantiation
        @was_last_act = true
        @validation_counter = 0
        @value = nil
    end
    # More code
    private
    def _update_validation(value)
        if @was_last_act # last action was an activation
            if value <= @threshold # search for a
               deactivation
                @validation_counter += 1
            else
                @validation_counter = 0
            end
        else # last action was a deactivation
            if value > @threshold # search for an activation
                @validation_counter += 1
            else
                @validation_counter = 0
            end
        end
    end
    def _alter_contexts()
        if @was_last_act
            contexts_changes = _format_act(
               @contexts_to_deact).concat(_format_deact(
               @contexts_to_act))
            @was_last_act = false
            @validation_counter = 0
            ContextActivation.instance.alter(*
               contexts_changes)
        else
            contexts_changes = _format_act(@contexts_to_act)
```

```
                .concat(_format_deact(@contexts_to_deact))
34             @was_last_act = true
35             @validation_counter = 0
36             ContextActivation.instance.alter(*
                contexts_changes)
37         end
38     end
39     # More code
40 end
```

Listing 5.6: Rule module in **rule.rb**

Now that we justified our choice of implementing a module, we will explain the implementation details described in Listing 5.6. The first thing we notice is the presence of the *RuleLogger* module. This module is responsible for creating an instance of the *Logger* class provided by Ruby such that we have one common log file for the whole sensor layer. This log file monitors the status of each rule when one of the rule's input sensors collects new valid data. It makes it easier and more precise to observe the conditions under which a rule's logic is executed. It also helps the developer to understand an unexpected reaction from the sensor layer.

Several attributes are defined in the initialization process (lines 3 to 8). The first one, the *threshold* attribute, is the reference value used in the inequation defined in Equation 5.2. The second attribute is the *validation_size* which is the reference value used in the validation process described below. The two last parameters define two sets of contexts, one to be activated, and one to be deactivated when the rule condition is verified. The attribute *value* has nil as default value. We made this choice to represent the fact that we do not have collected a value for this rule yet.

Once these parameters have been defined by the initialization process, the attribute *was_last_act* boolean is defined (line 5). This is an important attribute of the rule's behavior. It represents the current status of the rule by defining a theoretical last operation done by the rule. It was named after the following question: "Was the rule's last action an activation?". Thanks to it, the rule knows if it is searching for an activation or a deactivation of the rule. By rule activation, we mean that the rule is looking for a *value* that verifies the initial condition: value $>$ threshold. Conversely, by deactivating the rule, we mean that the rule is looking for a *value* that verifies the inverse condition: value $<=$ threshold. The default value for the *was_last_act* attribute is *true*. This signifies that after the initialization, the rule will be searching for a deactivation and then evaluating the

incoming data with the following condition: value <= threshold. The module has a setter for this attribute which means that the sensor layer can be adapted to the default contexts chosen by the developer when he defined the application context model.

The first important method of this module, the *_update_validation* private method, represents the validation process of the rule. It ensures that the current trend of the sensor data is reliable enough to perform a context switch. This process avoids asking the *ContextActivation* RubyCOP component to switch contexts unnecessarily. The *validation_size* parameter of the initialization process lets the developer define the *validation_size* constant of the rule. It describes the number of data, that verifies the inequality, that is required to execute the rule, and therefore update the application contexts. These data must be received one after the other, without a value that does not verify the inequality interrupting the sequence. In the event of a value interrupting the sequence, the counter will fall back to 0. The validation process is therefore strict and does not admit any unexpected values.

The responsibility for defining the value of the *validation_size* attribute is left to the developer, as the value depends largely on the rule itself. Indeed if we take our Smart Home application that has two contexts that define the temperature inside the house. The *Warm* context represents the context when the temperature inside the house is higher than 21°C in opposition to the *Cold* context. The rule responsible for defining these two contexts takes a temperature sensor as an input. The validation process ensures to not flood the *ContextActivation* component if the temperature sensor measures data that fluctuates between 21°C[1] and 22°C[2]. In the case of these two contexts, it is recommended to have a validation process by defining the attribute *validation_size* strictly higher than 0. On the contrary, if we take the Smart Home application as an example. The application has motion sensors that detect when someone enters the house or when someone exits the house. These data are used to define respectively the *SomeoneInside* and *NobodyInside* contexts. The reasoning needed from the provided raw data must not contain a validation process since the application must react at each motion.

The other important method defined in the *Rule* module is the *_alter_contexts* private method. It is responsible for asking the *ContextActivation* component to de/activate the contexts when the rule detects a change of application context.

First of all, the method starts by defining whether the rule was looking for de/activation. This defines which of the two inequalities was used to evaluate the data. If the rule *was_last_act* attribute is set to true, it means that the

[1]This data describe the *Cold* context.

[2]This data describe the *Warm* context.

last execution of the rule was an activation and that the rule is now looking for a deactivation. In this case, the inequation used to evaluate the data was value <= threshold. In the other case, as the rule was looking for an activation, the inequation used to evaluate the data was value > threshold.

Now that we have identified the desired execution of the rule, we have to prepare the parameters that we will give to the *ContextActivation* component to alter the desired contexts. This component takes a list of *ActionOnEntity* objects. In the case of a rule activation, lines 32 to 37, we want to activate the *contexts_to_act* and to deactivate the *contexts_to_deact*. In the other case, lines 27 to 31, we do the opposite since we want to undo the rule activation. The *contexts_changes* will then have objects that request the activation of the *contexts_to_deact* and the deactivation of the *contexts_to_act*. We also update the rule attributes *was_last_act* and *validation_counter* to reflect the rule execution and status change. Lastly, we only need to send our *ActionOnEntity* list to the *ContextActivation* component with its *alter* method that will execute the changes.

Simple Rule

The first class that includes the *Rule* module that we will present is called the simple rule due to its naïve approach. This *SimpleRule* class depicted in Listing 5.7 represents the simplest possible version of the logic part. It takes only one sensor as input which makes the rule condition computation trivial. Indeed, we can build a rule that takes the sensor data as it is to evaluate the inequation. This class instantiation does not require more parameters than the module initialization process.

```
1  class SimpleRule
2      include Rule
3      def initialize(threshold, validation_size = 0,
           contexts_to_act, contexts_to_deact)
4          super(threshold, validation_size, contexts_to_act,
               contexts_to_deact)
5      end
6      def evaluate(value)
7          @value = value
8          _update_validation(value)
9          if @validation_counter > @validation_size
10             _alter_contexts()
11         end
12         log_rule_status
13 # End of the class
```

Listing 5.7: SimpleRule class in **simple_rule.rb**

The *evaluate* method, lines 6 to 13, is the entry point of the rule. It takes as a parameter a value given by the sensor that triggered the method call. The first step of this method is to call the *_update_validation* method with the given value. This updates the *validation_counter* rule attribute according to the validation process defined by the *Rule* module. Then, we need to compare the updated *validation_counter* attribute to the *validation_size* rule constant, line 9. If the *validation_counter* variable is strictly higher than the *validation_size*, it means that the current sensor data trend is stable enough from the developer's point of view[3] to execute the rule. If the validation condition is verified, the rule can then update de application contexts using the *_alter_contexts* method from the *Rule* module, line 10.

The method ends by logging the status of the rule using the *RuleLogger* provided by the module. We decided to monitor the rule with the logger at each call of the method *evaluate* to be able to observe the evolution of the validation process.

Complex Rule

The second class that includes the *Rule* module proposed by this sensor layer implementation is called *ComplexRule* class depicted in Listing 5.8. This rule is more complex than the simple rule because instead of taking a single sensor as input, the complex rule can depend on several contexts. The other difference between these two rules is that the complex rule does not take the sensor value as it is. It allows the developer to introduce some logic in the form of a procedure to calculate a value from the sensors the rule has as input.

```
1  class ComplexRule
2      include Rule
3      attr_reader :value, :sensors
4      def initialize(threshold, validation_size, sensors,
          proc_function = nil, contexts_to_act,
          contexts_to_deact)
5          super(threshold, validation_size, contexts_to_act,
              contexts_to_deact)
6          # other parameters as attributes instantiation
7      end
8      def set_proc(proc)
9          @proc_function = proc
10     end
11     def evaluate(value)
12         # Ensures the procedure attribute is defined
```

[3] Since the *validation_size* attribute was defined by the developer himself.

```
13            values = @sensors.map{|sensor| sensor.value}
14            if values.all? {|value| !value.nil? }
15                @value = @proc_function.call(values)
16                _update_validation(@value)
17                if @validation_counter > @validation_size
18                    _alter_contexts()
19                end
20            else
21                log_rule_warn "At least one sensor has no value:
                      #{@sensors.map(&:name)}"
22            end
23            log_rule_status
24        end
25    end
```

Listing 5.8: ComplexRule class in **complex_rule.rb**

First, we describe the initialization process of a complex rule instance. It requires some additional arguments in comparison to the simple rule initialization process. The *sensors* parameter is a list containing the instances of the sensors that the complex rule takes as input. The *proc_function* parameter contains the procedure defined by the developer that computes a value based on the data coming from the sensors.

This attribute can be defined after the rule instantiation thanks to the setter proposed by the class in lines 8 to 10. We made this implementation choice to give the developer more possibilities to implement logic. By defining the procedure after the rule instantiation, the procedure can interact with the *value* complex rule attribute. If we take our Smart Home example to illustrate the interest in this implementation choice and define a context that describes the fact that there is at least one inhabitant inside the house, the rule must be able to track how many people went in and how many came out. To define such a rule, the developer needs a variable that keeps count of the number of people inside the house. This is where the *value* complex rule attribute comes in. This variable will not be reinitialized at each rule execution and can store information throughout the life of the complex rule. This is why we chose to let the *value* attribute be readable.

The *evaluate* method begins by ensuring that the *proc_function* is defined. Then, it gathers the values of each sensor instance defined in the *sensors* rule attribute, line 13. We need to ensure that each sensor has a measured value, line 14. We made this implementation choice because, since the rule depends on several sensors,

it can not effectively reason over data if some are missing as the *proc_function* requires each sensor data. This choice may be restrictive, but it ensures that the rule behaves with all the data it needs. This is where the *proc_function* comes in, line 15. We use the procedure provided by the developer to compute the *value* complex rule attribute that will be used for the condition evaluation. Once the *value* attribute is updated with the procedure computation result, the rest of the complex rule behavior is the same as the simple rule one, lines 16 to 20 and line 23. The final difference is line 21, where we use the *RuleLogger* to highlight the fact that the rule has not been evaluated because not all sensors have a measured value.

5.2.3 Simulator

The use of a sensor layer can be challenging due to the variability of sensor data and the range of their combinations. This is why we offer a sensor simulator in addition to the sensor layer. This *SensorsSimulator* depicted in Listing 5.9 aims to simulate a sensor by sending data as the sensor would have done with the difference that in this case, the data is theoretical. The developer can then send specific data to the sensor layer to test its behavior.

```
1  class SensorsSimulator < SensorsCommunication
2      def initialize(port)
3          super(port)
4      end
5      def run(values)
6          values.each do |value|
7              send_message(value)
8              sleep(1)
9          end
10     end
11 end
```

Listing 5.9: SensorsSimulator class in **sensors_simulator.rb**

This *SensorsSimulator* inherits from the class *SensorsCommunication* that we have defined by using the *Communication* class provided by RubyCOP. Its difference is that it connects itself to the computer IP address defined in *constants.rb* instead of the localhost address.

At its initialization, the developer needs to define the destination port of the simulator, lines 2 to 4. The port must match the port of the desired *SensorProxy*. Then the developer can send its theoretical data to the sensor layer by providing them as an array of values to the parameters of the *run* method. This method,

lines 5 to 10, will send each data individually to the defined port using the method *send_message* from the *Communication* module.

Chapter 6

Validation through the case study

This chapter presents the validation process of our sensory layer through our case study which is the Smart Home application introduced in chapter 1. As the aim of this case study is to validate our implementation of the sensor layer, during its development we focused on developing a *Context-Feature mapping* which was not a one-to-one mapping between contexts and features. A one-to-one mapping would have undermined the benefits of using the FBCOP paradigm, as contexts and features could have been merged, negating the need for distinct separation. This validation process aimed to highlight the strengths and weaknesses of our proposed sensor layer implementation.

The first step of our validation process, in section 6.1, is to design and develop our running example in the RubyCOP framework with its sensor layer. In this step, we first had to think about a case study that was representative of a realistic context-aware application. We also had to ensure that it was representative of the possibilities offered by the RubyCOP framework so that we could validate the functional aspects of our sensor layer.

With the implementation complete, we proceeded to test our Smart Home application sensor layer using physical sensors and the simulator tool (see section 6.2). In this phase, we simulate various context changes and observe the application's reactions to ensure that they align with our expectations defined in the previous step.

Then, in section 6.3, we analyze the sensor layer implementation from the point of view of code quality by defining its strengths and weaknesses. We also propose refactors to improve the implementation of this sensor layer.

Lastly, in section 6.4, we depict the threats to the validity of this process.

6.1 Smart Home design and implementation

This section presents the FBCOP design of this Smart Home application but also the design and the implementation of its sensor layer using the solution proposed in this master thesis. To design the FBCOP part of this application, depicted in subsection 6.1.1, we followed the methodology proposed by Duhoux [Duh22]. Then, we present the Smart Home sensor layer, in subsection 6.1.2 where we explain the Smart Home sensor layer design, our choices of sensors and also the sensor layer implementation details.

6.1.1 FBCOP design

We begin by detailing the first phase of the FBCOP application methodology which is the lexicon. It compiles descriptions of all contexts and features present in the application. Following this, we present the FBCOP diagram of the Smart Home application which introduces the *Context model*, the *Feature model* and the *Context-Feature mapping*. Together, these components form the comprehensive design of our running example as a FBCOP application.

Lexicon

The lexicon is a useful way for defining more rigorously what we intend to achieve with our application. It delineates the scope of each context and the expected behavior of each concrete feature. In our case, each context represents a partial aspect of the environment and their combinations form the comprehensive global context of the Smart Home environment. We start by describing the contexts and then the features.

Contexts

- **Occupation**: Describes if there is someone or not in the house. We are only interested in the authorized people. By authorized, we mean that the person has not broken into the home and entered through the main door. We have set this limitation to our Smart Home application to only monitor the main door entrance comings and goings.
  - **SomeoneInside**: At least one authorized person is inside the house and has entered through the main door.

  - **NobodyInside**: No authorized persons are present inside the house, people have left through the main door.

- **ExternalLuminosity**: Describes the external luminosity around the house.
  - **Sunny**: The luminosity is above a predefined threshold.
  - **Cloudy**: The luminosity is below a predefined threshold. This context does not represent the night context since it will be used to monitor the production of the house solar panels.

- **DifferentialTemperature**: Describes the difference between the temperature inside and outside of the house.
  - **WarmerInside**: The outside temperature is strictly lower than the temperature inside the house.
  - **ColderInside**: The outside temperature is higher than the temperature inside the house.

- **InsideTemperature**: Describes the temperature inside the house.
  - **Warm**: The temperature inside the house is above 21°C which is the inhabitant's ideal temperature.
  - **Cold**: The temperature inside the house is below or equal to 21°C.

Features

- **LightsController**: Open the lighting circuit breaker to let the current flow into the circuit when the feature is activated, and close it when the feature is deactivated. While it is activated, it listens to the requests of the owner to adapt the light intensity.

- **WifiController**: Request the Wi-Fi router to turn on the Wi-Fi when it is activated, and request to turn it off when it is deactivated.

- **Security**: This feature aims to secure the house. When activated, it ensures that the ground-floor windows set are closed and that they stay closed while this current feature is activated. This helps to limit potential break-in attempts. When this feature is deactivated, it ensures that the ground-floor and the first-floor windows sets have a coherent status depending on the corresponding deployed feature.

- **HeavyDeviceController**: At its activation, it creates one object for each large appliance managed by this feature.[1] These summarize the programs that are available for the appliance and other useful information such as the maximum waiting time before executing a request. During its activation and when it receives a request intended for the large appliances, it parses the request with the additional information needed.

  - **HeavyDevicesOn**: At the feature activation, it first initiates standby requests for each device. While it is activated and a request is received, it executes it directly if the corresponding device is available.
  - **HeavyDevicesOff**: When a request is received, it adds it to the waiting queue of the corresponding device if there is still room, otherwise it ignores the request. For this case study, we have chosen a queue size of 1 since for the selected appliances (dishwasher and washing machine), human intervention is required before starting a new program. Once in the queue, it starts a timer corresponding to the maximum waiting time before executing the request. When the time is up, the request is executed.

- **WindowsController**: At its activation, it first creates two objects representing the ground-floor windows and the first-floor windows respectively. It also declares the processes of sending commands to the home automation and more precisely, the open and close methods. They first look at the status of the windows sets to avoid sending unnecessary commands to the home automation.

  - **WindowsOpener**: At the feature activation, it first gathers the windows set it can act on. By default, it can act on the two sets defined by the feature *WindowsController* which are the ground-floor and the first-floor windows sets. In the case where the feature *Security* is deployed, the constraint on the windows affects the domain of this current feature. It can only act on the first-floor windows in such a configuration. Once the targets are defined, it sends a command to the ones that it wants to open by using the method declared by the *Windows Controller* feature.
  - **WindowsCloser**: At the feature activation, it sends a command to ask the home automation to close the windows of both ground-floor and first-floor.

[1] Two in our case, one for the dishwasher and one for the washing machine.

- **RadiatorsController**: Request the radiator's home automation to turn on the radiators when this feature is activated. At its deactivation, it requests to turn the radiators off.

Smart Home FBCOP diagram

The complete FBCOP application design is represented in Figure 6.1. The *Context model* is represented on the left and the *Feature model* on the right. They are both represented in their tree-like structure with the relationships between their nodes. The *Context-Feature mapping* is represented by the table between the two models. We have added arrows between the different parts of this diagram to highlight the links between these three concepts. In the context model, default contexts are identified by an asterisk.

Let's dive into the context-feature mapping we have defined.

- *SomeoneInside* → *LightsController*, *WiFiController*
  When the context *SomeoneInside* is activated, the feature selection process will select and activate the features *LightsController* and *WiFiController*. This mapping enhances the security of the Smart Home by deactivating the Wi-Fi when the context *SomeoneInside* is deactivated thanks to the *WiFiController* feature. It limits the possibility of cyber-attacks, particularly on the entire home automation system.

- *NobodyInside* → *Security*
  When the context *NobodyInside* is activated, the feature selection process will select and activate the feature *Security*. This feature ensures that ground-floor windows remain closed, enhancing security by reducing the risk of intrusion.

- *Sunny* → *HeavyDevicesOn*
  When sunlight is detected the context *Sunny* is activated. Thanks to this context, we can deduce that the house's solar panels generate electricity. The Smart Home can now take advantage of this renewable energy source by activating the feature *HeavyDevicesOn*, initiating program requests for large appliances and those previously on hold.

- *Cloudy* → *HeavyDevicesOff*
  When the context *Cloudy* is detected, the feature *HeavyDevicesOff* is activated because we prefer to delay requests and see if we can benefit from this renewable energy source in a reasonable time.

- *Warm*, *WarmerInside* → *WindowsOpener*
  If the contexts *Warm* and *WarmerInside* are perceived, it indicates that the

indoor temperature is above the ideal level of 21°C and that it is warmer inside the house than outside. The proposed solution to reach the ideal temperature, as mapped in the system, is to utilize the cooler outside temperature by opening the windows through the *WindowsOpener* feature.

- *Warm*, *ColderInside* → *WindowsCloser*
  If the contexts *Warm* and *ColderInside* are perceived, we still want to reduce the temperature inside the house, but the temperature outside the house is higher than inside. Opening the windows would lead to an increase in the indoor temperature. Therefore, the *WindowsCloser* feature is triggered to maintain the indoor temperature as low as possible.

- *Cold* → *WindowsCloser*
  If the context *Cold* is perceived, the mapping will trigger the *WindowsCloser* feature. Since we want to increase the temperature inside the house within a reasonable time, we aim to conserve the existing heat inside the house.

- *Cold*, *SomeoneInside* → *RadiatorsController*
  When the context *Cold* is activated, indicating that indoor temperature is below the ideal level of 21°C, and the context *SomeoneInside* is perceived, the Smart Home application triggers the *RadiatorsController* feature. This feature instructs home automation tools to turn on the radiators, thereby increasing the indoor temperature.

Smart Home design evolution

The current design is not exhaustive and could be extended with additional contexts and features to make it more adaptive and precise to its environment. For instance, we could include a context describing the battery level of the house, which stores excess electricity generated by the solar panels. This addition would improve the use of green electricity within the house. Another enhancement could involve the *Security* feature by integrating security camera controls. We could also enhance the concrete contexts children of the abstract *Occupation* context by perceiving if someone unauthorized is present in the house and alerting the police in this situation.

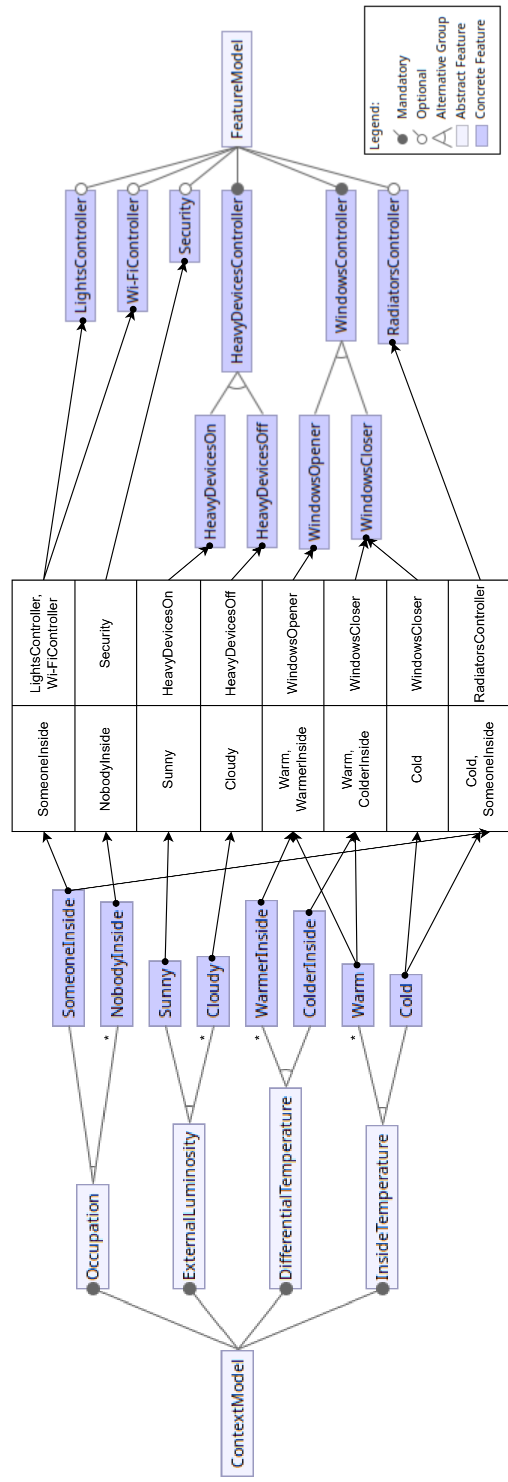

Figure 6.1: FBCOP diagram for the SmartHome application

6.1.2 Sensor layer

The sensor layer aims to perceive contexts from raw data received from physical sensors. In the case of our application, we need to perceive two sets of partial contexts: some inside the house and others outside the house as defined in subsection 6.1.1. This sensor layer can be defined in two parts where the first one is about defining the sensors themselves and the other is about their interaction with the contexts of our FBCOP application.

Before delving into the implementation of the Smart Home application sensor layer, we begin its design by describing our selection of sensors for this case study. Each of these choices is justified with the application *Context model*. Then explain how we connected these sensors from a hardware point of view. Lastly, we define the implementation of the Smart Home sensor layer and we explain how we defined the rules.

Selection of sensors

Here we describe which physical sensors we have used and why we chose them from a technical point of view. One important criterion in the sensor selection process was ease of use. We preferred "plug and play" sensors over those requiring additional hardware work. This allowed us to avoid using extra resistors and ensured a straightforward and efficient setup. We will also define them by the mode they use as defined in section 4.1. They will be organized according to the contexts they aim to perceive.

***SomeoneInside* and *NobodyInside* contexts** To accurately perceive these two contexts, we monitor the activity around the house's main entrance door. This assumption prevents the application from detecting a break-in as a normal entry into the house. To detect these comings and goings, we used two passive infrared (PIR) sensors as motion detectors. These sensors operate in *By Event* mode, which means they only react when an event occurs and, in this case, when motion is detected. They are strategically positioned on either side of the main door, with one sensor located outside and the other inside. This setup ensures that when someone enters the house, the external PIR sensor is triggered first, followed by the internal sensor. Conversely, when someone leaves the house, the internal PIR sensor is activated first, followed by the external one. This combination not only detects motion but also identifies its direction.

***Sunny* and *Cloudy* contexts** To accurately perceive these two contexts, we use a light-dependant resistor (LDR) sensor. This type of sensor provides a binary

output, indicating either light (value = 0) or dark (value = 1). It operates in the *Uninterrupted* mode, allowing continuous data retrieval to monitor data changes. We opted for a binary sensor over a continuous one because our Smart Home application does not require continuous data about external luminosity levels. The primary purpose of these contexts is to adjust the application to optimize the use of green electricity produced by the house's solar panels. Thus, we only need to know when the solar panels are actively generating electricity. This can be achieved by configuring the sensor's threshold to match the threshold for electricity production by the solar panels. To maximize its efficiency, this sensor is positioned outside the house, specifically near the solar panels.

***Warm*, *Cold* contexts and *WarmerInside*, *ColderInside* contexts** To accurately perceive these four contexts, we used two DS18B20 sensors: one positioned inside the house and the other outside. These sensors measure temperature and provide continuous data. They operate in the *Uninterrupted* mode which means that we can retrieve data whenever we like. For the *Warm* and *Cold* contexts, only the outside sensor is necessary, as we are solely interested in knowing the outdoor temperature. However, for the *WarmerInside* and the *ColderInside* contexts, both sensors are required. These contexts depend on the temperature difference between the inside and the outside of the house, making it essential to monitor both temperatures to determine the correct context accurately.

Arrangement of physical sensors

Now that we have selected the physical sensors we want for perceiving the contexts of our Smart Home application, we need to address the hardware aspect. To closely simulate the application's realistic use, we decided to use two Raspberry Pi 4B[2]. We chose this type of device because it is user-friendly, thanks to its GPIO pin system which simplifies connecting and managing sensors. We decided to use two devices to be as close as possible to a real use of our Smart Home application. The code responsible for collecting and sending data from sensors is written in Python. One Raspberry Pi will gather data from the sensors monitoring the inside of the house, while the other will handle the sensors monitoring the outside.

The setup details for each Raspberry Pi are illustrated in Figures 6.2 and 6.3. These figures depict the GPIO pins of the Raspberry Pi and the connected sensors. The red and orange lines represent VCC[3] connections (respectively 5V and 3.3V). The black lines indicate the ground connections and the blue lines show the data

[2] `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`
[3] VCC stands for Voltage Common Collector

connections. Those connections were made using jumper wires. We can retrieve data from a specific sensor by reading the information collected on the pin where the corresponding sensor is connected.

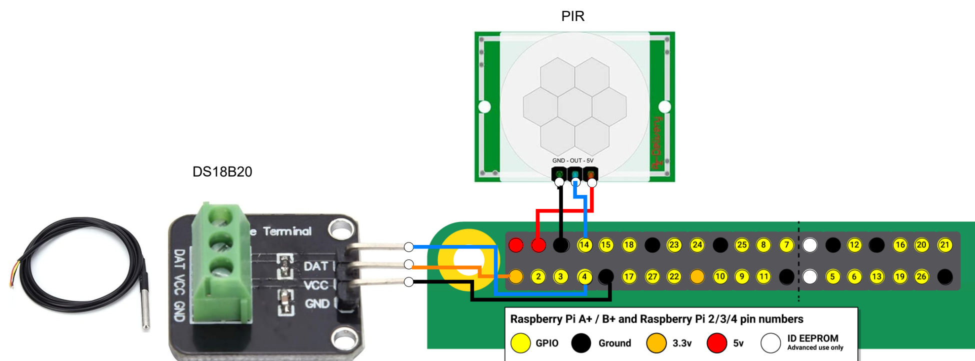

Figure 6.2: Hardware connections for the inside Raspberry Pi

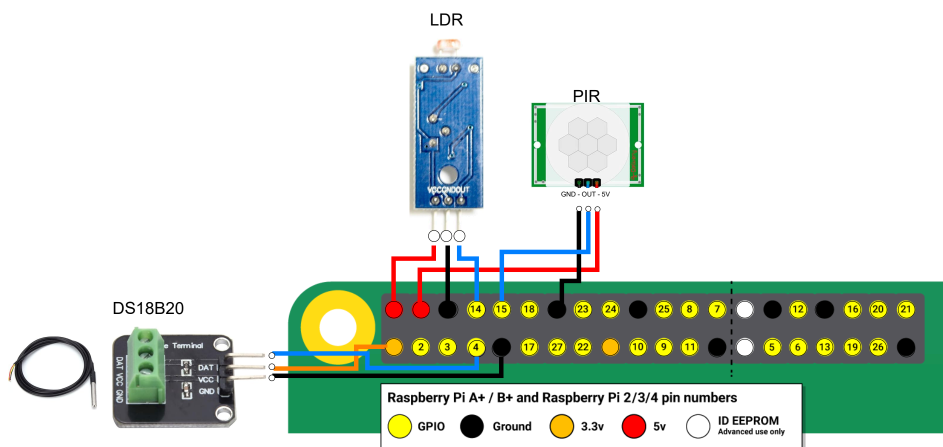

Figure 6.3: Hardware connections for the outside Raspberry Pi

Listing 6.1 represents the part of the code we used to measure the data of the LDR sensor and to send it to its destination port. We first need to specify how we want to refer to the Raspberry Pi pins, line 2. Then we define the pin where

we connected our sensor, line 3, and the port where we want to send the data we collect from the sensor, line 4. The last thing we need to do before measuring and sending the data is specify that the sensor we have connected to this specific pin is an input sensor. It means that the pin is used to read data. Then we only need to measure the data, line 7, and send it to the port we have defined, line 8.

```
# More code
GPIO.setmode(GPIO.BCM)
LIGHT_PIN = 14
port = "1111"
GPIO.setup(LIGHT_PIN, GPIO.IN)
while True:
  sensor_value = GPIO.input(LIGHT_PIN)
  client.send_msg(str(sensor_value), port)
  time.sleep(0.2)
```

Listing 6.1: LDR sensor python code

The other codes we used for other sensors to measure their data and to send it are available on `https://github.com/bealambert/memoire-rpi`. They follow the same idea as Listing 6.1.

Rules

To finalize the definition of our Smart Home sensor layer, we still need to define the *rules* that bind the sensors and the FBCOP contexts. We assume that we have declared one instance of the class *Sensor* for each physical sensor we have previously defined, as shown in Listing 6.2.

```
@ldr = Sensor.new("Luminosity")
@thermometerIn = Sensor.new("Thermometer_inside")
@thermometerOut = Sensor.new("Thermometer_outside")
@PIRIn = Sensor.new("Movement_detector_inside")
@PIROut = Sensor.new("Movement_detector_outside")
```

Listing 6.2: Sensors declaration in the *AppOntologyModelDeclaration* class

In the rules definition process, we first need to define the rule itself and its parameters, such as the validation size, the threshold, or the procedure depending on the rule's complexity and the contexts to be de/activated by it. We then add the rule to the sensor observers that the rule takes as input. This step ensures the rule will be evaluated each time one of the sensors receives new data.

***Sunny* and *Cloudy* contexts** Listing 6.3 depicts de rule responsible for perceiving these two contexts. We want to activate the context *Cloudy* if the LDR sensor detects no light, i.e., if the value measured is equal to 1. For this rule, we set the threshold to 0 which is the first parameter of the initialization. Moreover, to ensure stability, we also want a validation period of four continuous data.

This is why we set the second parameter of the rule initialization to 3. Lastly, we add the rule to the LDR's observers (line 2) because the rule takes the LDR's value as input.

```
luminosityRule = SimpleRule.new(0, 3, [@contexts.cloudy()],
    [@contexts.sunny()])
@ldr.add_rule_observer(luminosityRule)
```

Listing 6.3: External luminosity rule in the AppOntologyModelDeclaration class

***Warm* and *Cold* contexts** Listing 6.4 depicts the definition of the rule that perceives these two contexts. It is similar to the one we have just defined. The threshold is 21 since we want to activate the context *Warm* if the inside temperature sensor's read value is strictly above 21°C. We have set the same validation period as the previous rule.

```
insideTemperatureRule = SimpleRule.new(21, 3, [@contexts.
    warm], [@contexts.cold])
@thermometerIn.add_rule_observer(insideTemperatureRule)
```

Listing 6.4: Inside Temperature rule in the AppOntologyModelDeclaration class

***WarmerInside* and *ColderInside* contexts** Listing 6.5 represents the definition of the rule responsible for triggering these two contexts. It requires a *ComplexRule* instance. Indeed the logic of the *SimpleRule* class is not sufficient for the logic we want to implement since to be able to identify the temperature difference, we need to take the two temperature sensors into account. We want to determine if the temperature inside the house is strictly higher than the temperature outside the house. From a mathematical point of view, this gives:

$$T_{in} > T_{out} \tag{6.1}$$

$$\Leftrightarrow T_{in} - T_{out} > 0 \tag{6.2}$$

Where T_{in} and T_{out} are the values measured by the indoor (respectively outdoor) temperature sensor. We can implement a rule that has this logic by using a procedure defined in Equation 6.1. This procedure takes into argument an array and will reproduce the logic of Equation 6.2. We implemented the equation

Equation 6.2 over the Equation 6.1 because the procedure is used to compute a value that will then be compared to a threshold by the *ComplexRule*. In this case, the threshold is 0 and the procedure computes the difference between the two sensors. The input array of the procedure is defined in the rule initialization (third argument) on the second line. Now that we have implemented the logic, we still have to add the rules to each sensor's observers (lines 3 and 4) to be sure that the rule will be notified as soon as one of them receives a new measure.

```
thermometer_proc = Proc.new{|arr| arr.reduce(:-)}
thermometersRule = ComplexRule.new(0, 0,[@thermometerIn,
    @thermometerOut], thermometer_proc, [@contexts.
    warmerInside()], [@contexts.colderInside()])
@thermometerOut.add_rule_observer(thermometersRule)
@thermometerIn.add_rule_observer(thermometersRule)
```

Listing 6.5: Differential Temperature rule in the AppOntologyModelDeclaration class

SomeoneInside **and** ***NobodyInside*** **contexts** These two contexts are perceived by our two PIR sensors positioned on either side of the main door. PIR_{in} designates the indoor PIR sensor while PIR_{out} deisgnates the outdoor PIR sensor. Each sensor sends a timestamp when motion is detected. Listing 6.6 depicts the rule responsible for perceiving those contexts based on the two PIR sensors.

The logic we want to implement in this rule is the following: if someone enters, we increment the rule's value, if someone exits, we decrement the rule's value. The rule's value represents the number of people inside the house. To implement this logic, we begin by initiating our rule (line 1) with its threshold set to 0 since if its value is strictly greater than 0 it means that there is at least one person inside the house. We also set the validation period to 0 because we are interested in the events and there is no need for validation in this case. The *ComplexRule*'s procedure argument is set to nil for now. We first need to initiate the rule because we want to have access to the rule's value in our procedure. This is why we chose to first declare de rule and then add it to the procedure later. We set the rule attribute *was_last_act* to false (line 2) since we want our sensor layer to adapt to the application *Context model*. The default context for these two contexts is *NobodyInside*. When the application is launched, this rule must seek to activate *SomeoneInside*.

Before we can implement this logic, we first need to mathematically define entering or leaving the house with PIR data as input. If someone enters the house,

the PIR_{out} will be the first to detect motion and as a consequence send the measure timestamp. On the contrary, if someone leaves the house, the PIR_{in} will be the first to detect motion and as a consequence send the measure timestamp. This logic is represented in the procedure in line 4. One strategic step of this procedure to ensure that the rule behaves as we expected is to consume the data as soon as we use them (lines 5 to 8). Otherwise, the rule might trigger context changes in unexpected situations. Now we only have to update the rule's value and let the *ComplexRule* do the rest of the logic for us. Lastly, we need to add the procedure to the rule and add the rule to the interested sensors's observer as done for the previous rules.

```
movementDetectorsRule = ComplexRule.new(0, 0,[@PIRIn,
   @PIROut], nil, [@contexts.someoneInside], [@contexts.
   nobodyInside])
movementDetectorsRule.set_was_last_act(false)
movement_detector_proc = Proc.new do |arr|
    sensor_value = arr.reduce(:-) # PIRin-PIRout
    movementDetectorsRule.sensors.each do |sensor|
        sensor.value = nil
    end
    value = movementDetectorsRule.value || 0
    if sensor_value > 0 # Someone enters
        value += 1
    elsif sensor_value < 0 # Someone exits
        value -= 1
    end
    value
end
movementDetectorsRule.set_proc(movement_detector_proc)
@PIRIn.add_rule_observer(movementDetectorsRule)
@PIROut.add_rule_observer(movementDetectorsRule)
```

Listing 6.6: Occupation rule in the AppOntologyModelDeclaration class

6.2 Testing the sensor layer

Now that our Smart Home application is fully implemented, from its sensors to its features. We are ready to test our sensor layer behavior through the Smart Home application.

In this section, we describe the two approaches we used to test the adaptative behavior of our application thanks to the sensor layer. The first approach we

describe is the simulator approach, see subsection 6.2.1. It allows the user to send predefined theoretical data to the sensor layer. The second one, see subsection 6.2.2, is the use of real physical sensors and their configuration as defined in section 6.1.2.

In addition to these two approaches, we decided to use the visualization tool: "Context and Feature Model Visualiser" proposed by Duhoux et al. [DDLM19] to observe real-time context changes more easily. We chose to complement our testing process with this visualization tool because it allows us to automatically and instantaneously observe changes in context and potentially feature changes. It also allows us to observe changes within the application *Context model* and *Feature model* step by step. This tool is useful to test our implementation since those behavior adaptations are not necessarily instantly visible, as some features are only revealed through user interventions. For instance, the *HeavyDevicesOn* and *HeavyDevicesOff* features have no visible behavior until the user sends requests to the Smart Home application. In contrast, the *LightsController* feature has a visible behavior both upon activation and just before its deactivation. For such a feature, we can monitor the application's behavior adaptation only by direct observation.

Finally, we set up unit tests to precisely validate the sensor layer corner cases, in subsection 6.2.3.

6.2.1 Simulator

The first approach we chose to use to perceive the surrounding contexts is the use of the simulator tool. This tool enables generating artificial data and sends it to the sensor layer as the physical sensors would have done. The sensor layer is not aware that it is a simulator and not a physical sensor that is sending the data. This approach is useful as a first test phase. We can partially test our implementation by activating one rule at a time and use the visualization tool to verify whether everything worked as expected. Additionally, it allows for testing with corner cases since we can precisely define the data we want to send. Moreover, having complete control over the sensor data makes it easier to focus on specific expected behavior changes during testing.

For instance, the Listing 6.7 depicts the code responsible for perceiving the *Warm* context. To do so, we must begin by creating an instance of the *Simulator* class, with the port parameter set to the target port of the data we want to send, lines 1 and 2. Then, we can declare our theoretical data and send them. In our case, we aim to perceive *Warm* context while we currently are in the *Cold* context. We must therefore send numerical values that are strictly greater than 21. We also must pay

attention to the validation process of the rule. We set the *validation_size* attribute to 3 which means that the rule needs to evaluate four values received one after the other that verify the condition to change the context. Concretely, we need to send four values that are strictly higher than 21 to change the context to *Warm*.

```
port = 1112
simulator = SensorsSimulator.new(port)
simulator.run(Array.new(4){22})
```

Listing 6.7: Simulator code to change Warm context to Cold context

6.2.2 Physical sensors

The other approach we can use to test the dynamic adaptability of our application thanks to the sensor layer is the use of real sensors and in our case, the hardware setup presented in section 6.1.2. This approach requires more work than the simulator approach before being able to adapt the application's behavior. First of all, we need to connect the sensors and ensure that we can not only read their data but also send it to the port of a specific address, using a script as Listing 6.1 for example.

This approach may be more complex, but it allows us to test our application under more realistic conditions. By concurrently perceiving all the partial contexts defined in the application, we can form the overall environmental context. Using this approach, we also want to be able to observe changes in context, and consequently changes in behavior, within a reasonable timeframe. Consequently, we need to adapt to the environment so that the sensor is in the situation we want to reproduce. If we take up the change of example contexts from subsection 6.2.1, *Warm* context to *Cold* context, we only need to make sure that the temperature sensor will measure a temperature below or equal to 21. To do so we can use a bowl of cold water since the sensor is waterproof or any other mechanism that would ensure a temperature below 21°C.

6.2.3 Code quality

To validate our sensor layer, we made unit tests using *rspec*. To enable us to carry out these tests, we used mock-ups to focus solely on the sensor layer and not on the entire framework.

These tests validate the corner cases of the *evaluate* method of the *SimpleRule* by looking at the *was_last_act* rule attribute value. We have also tested the validation

process by providing data that fluctuates around the rule threshold. In this case, the *was_last_act* rule attribute remains constant and the *validation_counter* is reset to zero.

We also tested the data flow from the sensor to the rule. We had to define a sensor and add the rule we wanted to test to it. Then we give the value to the sensor using the *update_value* sensor method that will notify the rule itself.

6.3 Results

In this section, we present our thoughts on how our sensor layer works, following the validation process of section 6.2. We talk about its strengths, but also its weaknesses and the limitations we identified.

6.3.1 Strengths

First of all, the main challenge was to enable automatic context detection and activation depending on the application's surrounding environment but also to solve realistic problems. Our case study and the validation process helped us to validate these objectives. The case study was designed as a prototype for a potential realistic application. Moreover, the sensor layer behaves in a reasonable timeframe. Indeed, the only visible delay is in launching the application. Table 6.1 presents the time required to start the application without the sensor layer and with the sensor layer. To obtain these values, we used a script to obtain 100 measurements, which we averaged for each situation. The delay of around 5 seconds is inherent in the addition of the sensor layer. This is due to the time required for each sensor proxy to connect itself to its communication channel created by the server. Once this start-up phase is complete, the application behavior shows no visible delay between receiving sensor data and activating the corresponding contexts.

Application range	Time (s)
FBCOP	0.006
FBCOP with the sensor layer	5.016

Table 6.1: Table comparing the time needed to start the Smart home application with solely FBCOP vs. with the sensor layer in addition to FBCOP.

Then, we can confirm that the sensor layer is compatible with *continuous*, *discrete* and *event-driven* physical sensor data as long as their data are convertible in the

float type. Indeed, the case study we used in the validation process used a DS18B20 sensor which provides *continuous* data. This data represents the temperature. The Smart Home application also used an LDR sensor that provides *discrete* data representing the light or the absence of light. We were also able to test the behavior of our sensor layer with the *event-driven* physical sensors thanks to the PIR sensor. This type of sensor validated the benefits of our *ComplexRule* design and more precisely its procedure mechanism. Indeed, during our test phases, we realized that the data values of the *event-driven* physical sensors were as valuable as the element itself. Therefore, adding a procedure to the *ComplexRule* design allows us to consume the data in the same way as we did for the *movementDetectorsRule* in Listing 6.6 to focus on the event itself.

6.3.2 Weaknesses

We validated the behavior of our sensor layer using raw data that could be converted into floating values but for other data types, the developer needs to do a mapping before sending the data to the sensor layer. The ideal sensor layer would be compatible with all types of sensor data and would convert data as needed. This would reduce the developer's workload and bring all the logic together in the sensor layer.

Another weakness we have identified in the validation process is the binarity of the rules. They only have two states and thus two possible actions: [activate($\mathcal{C}_a$), deactivate($\mathcal{C}_d$)] or [activate($\mathcal{C}_d$), deactivate($\mathcal{C}_a$)]. Take the Figure 6.4 as an application design example to illustrate this weakness of our sensor layer. It is another possibility of the FBCOP diagram of our case study for the temperature contexts. According to this *Context model*, the context changes are more complex than in the design proposed in subsection 6.1.1 and they are not binary. If we want to activate the context *WarmerInside* while the context *ColderInside* is active, we need to perform this request: **[activate(*WarmerInside*), deactivate(*ColderInside*)]** to ensure the validity of the *Context model*. Now, if we want to activate the context *Cold* while the context *WarmerInside* is active, we need to perform this request: **[activate(*Cold*), deactivate(*WarmerInside*)]**. Another possible context change is when we want to activate the context *ColderInside* while the context *Cold* is active, we need to perform this request: **[activate(*ColderInside*), deactivate(*Cold*)]**. There is one remaining context change possible but we can also highlight the limitation of the sensor layer with these three context changes and their requests. Indeed, when we look at the possible requests, we see that one binary rule of our sensor layer can not monitor these three contexts since we only define two sets of contexts at the rule initialization.

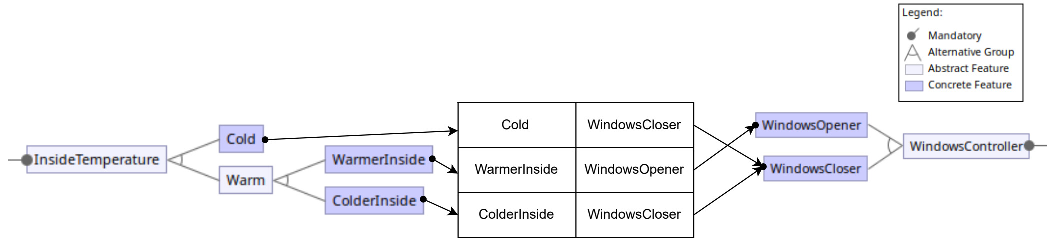

Figure 6.4: Context Feature model of an application that represents a weakness of the sensor layer

In subsection 7.2.2, we discuss about how we can solve this weakness to improve the sensor layer by increasing its capabilities.

6.4 Threats to validity

This sensor layer validation presents certain vulnerabilities due to several factors.

One primary threat to validity is the fact that we have tested our sensory layer implementation on only one application. Although we aimed to make this application as representative as possible of the possibilities offered by FBCOP, we may have not covered all the capabilities of this framework.

Another threat involves our sensor selection. We chose physical sensors to ensure a diversified range of data types and sensor modes (e.g., *On request*, *By event* modes). However, we have not validated the use of the sensor layer for sensing the inner components of a device. While we can theoretically assert that it works as long as the desired parameter is accessible, practical validation is missing. Once we have access to the parameter value, we need a script to retrieve this value and send it to the appropriate port to adapt the application's behavior based on that context. It would be interesting to validate this sensor layer by implementing another application with different sensor requirements such as the *Smart City Guide* presented by Cardozo and Mens [CM22]. This application requires user inputs to select certain contexts such as users' age and language. It also involves different aspects of the environment, such as the battery level or even the connectivity of the device (Wi-Fi or 4G).

Chapter 7

Conclusion

This chapter presents the conclusion of this master thesis, in section 7.1 where we review all the work accomplished. The section 7.2 depicts the future work we have identified for the sensor layer.

7.1 Conclusion

Building context-aware applications might reveal itself complicated due to the wide range of application contexts and their possible combinations. The RubyCOP framework provides support to the developer to implement FBCOP applications. Currently, it does not provide support for the context discovery process. It is up to the developer to work with the *Context Activation* component to de/activate contexts representative of the application environment.

This master thesis aims to enhance the RubyCOP framework by proposing a sensor layer implementation responsible for perceiving and de/activating the contexts of the application. This sensor layer was designed to work with physical sensors and with a validation process to avoid flooding the rest of the framework with context change requests. It has been designed to allow the developer to define the logic part of the sensor layer so that it can be adapted as much as possible to his application. We chose to use ontology modeling to implement the sensor layer in order to represent the sensors as concepts and the logic as relationships between these concepts and the application contexts defined by the developer.

The proposed sensor layer implementation was tested through a RubyCOP application we implemented: Smart Home application. We wanted a case study as representative as possible of the applications in which our sensor layer would prove useful. Thanks to this application and our sensor selection, we were able to

observe the behavior of the sensor layer in real-life situations. The conclusions we were able to draw from the validation process are promising since our sensor layer is not only compatible with naive logic. It can therefore be used to develop a wide range of context-aware applications. However, there are still improvements to be made in order to continue to extend the possibilities offered by this sensor layer. We present some of the in section 7.2, along with our proposals on how to tackle these challenges.

7.2 Future work

7.2.1 Extend data type used

One limitation we identified in subsection 6.3.2, is the compatible sensor data types. The future work we propose is to extend the sensor layer to support sensor data types that can not be naturally converted into floats.

This limitation could be partially solved by allowing the developer to communicate a mapping that would convert the values into floats. The developer could define this mapping when he defines the interested sensor. At the sensor creation, it would give this mapping to its sensor proxy which is responsible for converting the raw data into float. Even if this solution is not perfect, it would limit the developer's workload when working with sensors that supply, for example, string values with a limited (discrete) domain as the LDR sensor.

7.2.2 Extend the possible actions of the rules

Currently, the rules proposed in our sensor layer have binary actions since they have only two possible actions. This is a limitation we have identified in subsection 6.3.2. A possible future work would be to improve the rules or add a new one to allow the sensor layer to change contexts in a more complicated way than the binary actions already present.

This can be done by providing a *RuleBase* class as support to the developer. This class would be responsible for the rule initialization and would also provide methods to help the developer. The developer would have the responsibility to implement the rule *evaluate* method. With this proposed solution, the developer would have total freedom over the logic part of the sensor layer.

7.2.3 Enable sensor data validation

Currently, the only condition to use sensor data is that it can be cast into a float. We could enhance this part by letting the developer define a domain for each sensor. For example, the developer could define the expected sensor domain by setting a minimal and a maximal value.

This can be done through the sensor creation in the ontology model declaration. The sensor will define the domain into its sensor proxy that currently filters data that could not be cast into float.

Bibliography

[AP19] Asma Alotaibi and Charith Perera. Smart home human activity simulation tool for openhab-based research. Technical report, Technical Report, 2019.

[CC12] Yeong-Sheng Chen and Yu-Ren Chen. Context-oriented data acquisition and integration platform for internet of things. In *Conference on Technologies and Applications of Artificial Intelligence, TAAI 2012, Tainan, Taiwan, November 16-18, 2012*, pages 103–108. IEEE Computer Society, 2012.

[CM22] Nicolás Cardozo and Kim Mens. Programming language implementations for context-oriented self-adaptive systems. *Inf. Softw. Technol.*, 143:106789, 2022.

[DDLM19] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. Dynamic visualisation of features and contexts for context-oriented programmers. In José Ignacio Panach, Jean Vanderdonckt, and Oscar Pastor, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2019, Valencia, Spain, June 18-21, 2019*, pages 10:1–10:6. ACM, 2019.

[DMD19] Benoît Duhoux, Kim Mens, and Bruno Dumas. Implementation of a feature-based context-oriented programming language. In *Proceedings of the Workshop on Context-oriented Programming, COP@ECOOP 2019, July 15-19, 2019, London, UK*, pages 9–16. ACM, 2019.

[DMD22] Benoît Duhoux, Kim Mens, and Bruno Dumas. Rubycop: A feature-based context-oriented programming framework. In *BENEVOL 2022*, September 2022.

[Duh22] Benoît Duhoux. *Feature-based context-oriented software development.* PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2022.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[HT08] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proceedings of the 12th International Software Product Line Conference (SPLC)*, pages 12–21. IEEE Computer Society, 2008.

[KM03] Panu Korpipää and Jani Mäntyjärvi. An ontology for mobile device sensor-based context awareness. In Patrick Blackburn, Chiara Ghidini, Roy M. Turner, and Fausto Giunchiglia, editors, *Modeling and Using Context, 4th International and Interdisciplinary Conference, CONTEXT 2003, Stanford, CA, USA, June 23-25, 2003, Proceedings*, volume 2680 of *Lecture Notes in Computer Science*, pages 451–458. Springer, 2003.

[LLH11] Wei Liu, Xue Li, and Daoli Huang. A survey on context awareness. *2011 International Conference on Computer Science and Service System (CSSS)*, pages 144–147, 2011.

[LY02] Kalle Lyytinen and Youngjin Yoo. Issues and challenges in ubiquitous computing. *Communications of the ACM*, 45(12):62–65, December 2002.

[MCD16] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming, COP@ECOOP 2016, Rome, Italy, July 19, 2016*, pages 7–12. ACM, 2016.

[RMM+] Vincent Ricquebourg, David Menga, Bruno Marhic, Laurent Delahoche, David Durand, and Christophe Loge. Service oriented architecture for context perception based on heterogeneous sensors network. In *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 4557–4562. ISSN: 1553-572X.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**