Informatics

# An Extendable Multi-Agent System for Thermal Comfort Control Leveraging KNX-IoT Semantics

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Master program Computer Engineering**

by

**Thomas Rapberger, B.S.c.**
Registration Number 51867996

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Ass. DDipl.-Ing. Dr.techn. Gernot Steindl, B.S.c.
Assistance: Christoph Gehbauer, M.S.c.

Vienna, March 31, 2025

Thomas Rapberger  Gernot Steindl

# Erklärung zur Verfassung der Arbeit

Thomas Rapberger, B.S.c.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 31. März 2025

Thomas Rapberger

# Acknowledgements

I want to express my sincere gratitude to Dr. Gernot Steindl. His supervision during the preparation of my master's thesis was invaluable. Allowing me to work on such an interesting topic was a fascinating time, and I am very grateful for this opportunity. Throughout the many challenges I encountered, his constructive feedback and continuous support enabled me to further enhance this work's quality and scientific relevance.

I would also like to thank Mr. Christoph Gehbauer, whose expertise contributed to this thesis, especially during the literature research on topics covering advanced control strategies in building automation.

Furthermore, I am deeply grateful to my family and friends, who supported me throughout my studies and always helped me during challenging times.

Lastly, I acknowledge all the resources used in this thesis, as they provided valuable insights and allowed me to expand my knowledge in various areas.

# Kurzfassung

Während des Bestrebens, die globalen Emissionen auf ein Minimum zu reduzieren, ist es unerlässlich, den Energieverbrauch von Gebäuden zu berücksichtigen. Zur Senkung des Energieverbrauchs in diesem Sektor ist die Implementierung fortschrittlicherer Regelungsstrategien zur Steuerung der Innentemperatur und anderer Verbraucher unabdingbar. Diese Strategien müssen eine Vielzahl von Variablen berücksichtigen, wie beispielsweise Wettervorhersagen, Belegung und die thermischen Eigenschaften des Gebäudes. Die Steuerung großer Gebäude mit vielen Zonen kann komplex sein, da viele Variablen in einem Regler berücksichtigt werden müssen. Eine weitere Schwierigkeit besteht in der Entwicklung eines Systems, das sich leicht erweitern und skalieren lässt. In dieser Arbeit wird daher ein Multi-Agenten-System vorgestellt, das mit der KNX 3$^{rd}$ Party API arbeitet, um eine bestehende Technologie mit unmissverständlicher Kommunikation zu unterstützen. Die KNX 3$^{rd}$ Party API dient hierbei als Schnittstelle zwischen den steuernden Agenten und den Gebäude-Aktoren und -Sensoren. Da KNX als standardisiertes Bus-Kommunikationsprotokoll entwickelt wurde, unterstützt die für die Gestaltung des Systemverhaltens verwendete Software, die sogenannte ETS, diverse Modellierungsinformationen eines Gebäudes. Daher ist es von entscheidender Bedeutung, zu analysieren, welche Informationen innerhalb der ETS modelliert werden müssen, um den Agenten genügend Informationen zur Verfügung zu stellen, und welche Informationen nicht modelliert werden können. Darüber hinaus wird eine Ontologie erstellt, die es dem System ermöglicht, Flexibilität zu gewährleisten, indem verschiedene Geräte, Zonen oder Agenten zum System hinzugefügt oder entfernt werden können. Diese Ontologie wird verwendet, um die Fähigkeiten der Agenten zu beschreiben. Da manche Agenten zusätzliche Fähigkeiten anderer Agenten benötigen, veröffentlicht jeder Agent die Beschreibung seiner Fähigkeiten in einem zentralen Datenspeicher, sodass andere Agenten diese finden und mit ihnen kommunizieren können. Die Implementierung jedes Agenten erfolgt als Mitglied des Web of Things (WoT). Die Fähigkeiten eines jeden Agenten werden innerhalb der sogenannten Thing-Description beschrieben. Dies ermöglicht es anderen zu erkennen, wie eine Interaktion mit dem Agenten durchgeführt werden soll. Das System wird in einem simulierten Gebäude getestet, um die Implementierung zu evaluieren. Im Anschluss werden die Ergebnisse diskutiert und die Effektivität der vorgeschlagenen Lösung analysiert. Hierzu wird der Controller mit einer Basisimplementierung verglichen, sowie die Erweiterbarkeit und Skalierbarkeit des Systems untersucht.

# Abstract

In an effort to reduce global emissions to a minimum, the energy consumption of buildings must also be considered. To reduce power consumption in this sector, more advanced control strategies for controlling indoor temperature and other consumers have to be implemented, including a variety of variables, e.g., weather predictions, occupancy, thermal characteristics of the building, and many others. Controlling large-scale buildings with many zones can become quite complex as many variables are considered at one controller. Furthermore, developing a system that allows it to be easily extended and scaled poses another difficulty. Therefore, this thesis proposes a Multi-Agent System (MAS) that operates on the KNX 3$^{\text{rd}}$ Party interface to facilitate existing technology with unambiguous communication. This interface will act as a bridge between the controlling agents and the building actuators and sensors. As KNX was developed as a standardized bus communication protocol, the software called Engineering Tool Software (ETS), which is used to design the system's behavior, does not support a great variety of modeling information for a building. Therefore, it is essential to analyze which information has to be modeled inside of the ETS to provide enough information for the agents and which information cannot be modeled. Furthermore, an ontology is created that allows the system to become flexible in the sense of adding and removing different devices, zones, or agents to the system. This ontology is used to describe the skills of each agent. Moreover, each agent also requires additional skills from other agents. Therefore, each agent publishes a description of its skills to a central data storage to allow other agents to discover these skills and communicate with the corresponding agent. Each agent is implemented as a member of the Web of Things (WoT). The skills of each agent are described inside of the Thing Description (TD), as this allows others to identify how an interaction with the agent is carried out. The system is tested on a simulated building in order to evaluate the implementation. It is shown that the system achieves better performance than a benchmark controller and that it can be extended easily.

# Contents

CHAPTER 1

# Introduction

Buildings are one of the primary energy consumers, as they account for about one-third of the energy produced worldwide [GTPLC+22]. They also contribute in large numbers to the CO2 emission. Based on the IEA, residential buildings directly produced about 2 Gt of CO2 [IEA23]. In recent years, there has been a growing trend towards sustainable energy resources and the careful use of these resources to drastically reduce greenhouse gas emissions. Single-family homes, as well as office buildings and industrial plants, have the means to convert their energy in order to reduce emissions and lower the operating costs of buildings. Photovoltaic systems, heat pumps, and batteries provide consumers with different quantities of energy at different times. Also, net providers use Renewable Energy Sources (RES) and sell them at varying prices as their generation depends on the environment. Furthermore, buildings are equipped with increasingly more sensors, allowing them to track user presence or measure the outdoor temperature making the controllers more intelligent, flexible and efficient in achieving the users requirements. Nevertheless, reducing greenhouse gases further requires more complex control strategies as more variables are considered. This requires accurate modeling of the optimization functions. In the following, a flexible and decentralized system is proposed that is able to include different controllable assets within a building.

## 1.1 Problem description

KNX[1] is a prominent standard for controlling buildings. It enables the integration of many different components to create basic control systems. However, there are many other technologies due to the emergence of the Internet of Things (IoT) that are installed and interacted within the building. This led to a proliferation of different protocols, which lead to interoperability issues between devices is no longer guaranteed. To overcome this issue,

[1]https://www.knx.org/knx-en/for-professionals/index.php

the Web of Things (WoT) [Con] was established that allows one to semantically describe what information a device can offer. It also regulates how devices can communicate with each other to create an open standard that every vendor can participate in. This allows WoT devices to exchange information without knowledge of a specific communication protocol. This is particularly important in the building domain, as devices from many manufacturers need to work together to achieve a common goal.

However, this brings with it another problem: the large number of variables in the system leads to a very complex system with many volatile variables and uncontrollable constraints. These include, for example, the weather, which greatly influences the indoor temperature of buildings. Energy companies offer their services at varying prices and at different times. Controllers are therefore required to also adapt to these variations. Moreover, the energy demand in the building is not constant, as occupancy and outdoor conditions influence the indoor temperature. In order to regulate the thermal comfort of a room, it is necessary to consider several additional factors, including occupancy, shading, ventilation, CO2 levels, and others.

A common approach for controlling these variables is to use a Building Management Systems (BMS). This can be done at one central point, i.e., the data for each room is gathered at one unit that solves the optimization problem. By using an accurate model of the building, very precise results can be achieved. Creating such a model is very complex and requires identifying unknown parameters of the building. These models become convoluted, and finding an optimal solution will be even more complex [FLZ+22].

One solution to this problem is to simply address the optimization problem of each zone individually. This approach reduces the complexity of the problem, as it no longer has to consider all variables simultaneously. Also, other advantages are given with this approach, as the system can be scaled more efficiently as different zones can be created and removed as required. These zones will optimize a smaller subset of the global problem, e.g., the thermal comfort in a zone. For each zone, a problem-solving instance is created that communicates with other instances to achieve its goals. One example of such a communication is that the instance optimizing the thermal comfort in a zone acquires information of an instance having knowledge about weather predictions. This distributed method is often referred to as a Multi-Agent System (MAS).

When a system consists of multiple entities, communication between the entities is an important part. In addition to the usual requirements for a communication medium, such as transmission security, throughput, etc., there is another important property that such a system should have: the ability to share knowledge consistently. Existing systems are often designed in such a way that only those who understand what data is being exchanged and its meaning can participate in the network. The use of ontologies creates a knowledge base that eliminates ambiguity. Especially in times when existing infrastructure is being retrofitted with new components or control strategies, uniform communication is vital. Ontologies are a suitable solution to that ensure that future components can be integrated into the system. As already described, added devices should also know how to interact with the system, and the existing agents should also be able to interact with these devices. Unified interaction can be achieved using standards like WoT. Therefore, this research aims to increase the efficiency of building controllers by utilizing a MAS, which uses semantic messages to exchange information.

## 1.2 Research Questions

As mentioned, KNX is a well-established standard of building automation and represents a possible candidate for overcoming the problem of ambiguity in BMS since they are currently developing a new standard called the KNX-IoT 3rd Party Application Programming Interfaces (API) This API is part of the KNX-IoT standard and should enable any devices to extract semantic information from the system as well as control individual components in the building. Currently, not all relevant data for a MAS can be modeled with the Engineering Tool Software (ETS) software tool provided by KNX, which is used to develop a BMS. Therefore, the question of what information has to be modeled within the KNX system so that different optimization strategies like rule-based or Model Predictive Control (MPC) can retrieve knowledge and optimize a building. Some of this data will never be provided by KNX as its purpose is to create a standardized communication for actuators and sensors in a building. Therefore, it necessitates the research of what information is not provided by a KNX model and the KNX 3rd Party API to determine which additional information must be added when developing a KNX-IoT system to ensure that all essential data is available for a controlling system.

> **Research question 1**:
> *What information has to be additionally provided along the KNX-IoT information model in order to be capable of controlling and optimizing the thermal comfort of a building with a multi-agent system?*

The next step is to use the information provided by a uniform building interface, like KNX, to develop a framework that optimizes the use of a building. Any controller will be able to access information via the KNX-IoT interface and can also directly control actuators. However, it must be considered that components can become defective and have to be replaced. Also, new components and zones can be added during the lifecycle of a building. To ensure the system stays operational, existing controllers must be informed about these changes. In the best scenario, the system takes responsibility for changes and recent integrations of devices and zones without requiring further input from the user. However, at least within the context of the KNX-IoT system, these components and zones must undergo another integration process since devices and usages of rooms are configured when implementing the KNX project.

Since devices can be added to the building at any time, it is plausible that new controllers will be added as well. Therefore, the Multi-agent System should allow the integration of new agents at runtime with as little user interaction as possible. This raises another problem: how do agents know what functionality other agents provide? An example that illustrates this use case is the installation of a heat pump in a building. The agent responsible for energy-efficient heating should now consider the newly integrated device with its controlling agent, as it can generate hot water at a lower cost. The system needs to recognize the new device and determine what it can do. To solve this problem, the capabilities of all agents are described by an ontology to later facilitate matching between an agent that provides a capability and an agent that is looking for a specific capability.

**Research question 2**:
*What is an appropriate architectural design within a multi-agent building energy management system to facilitate a semi-automatic integration by using existing information from the KNX IoT interface to provide a scalable and extendable energy optimization solution?*

## 1.3 Methodology

To answer the first research question, a literature review was carried out to identify the most important parameters. Special interest was given to the heating of buildings as this is the pronounced energy consumer. Therefore, primarily the IEEE Xplore, MDPI, and ScienceDirect were queried for the following to identify relevant literature: *building control*, *thermal modeling*, *optimization techniques*, and *control strategies*. By systematically analyzing references from initial sources, further studies on *heat transfer*, *system identification*, and *predictive control* were identified. These studies contribute to the understanding of *HVAC performance* and *building energy management*. As the prevalent heating system in Europe is radiator heating, it was necessary to be determined which inputs and outputs are needed in this scenario. This work contributes to identifying the most significant parameters. A parameter is considered in this thesis if it is relevant for a controller that increases the comfort of the inhabitants, e.g., controlling the indoor temperature, lighting, or ventilation. Furthermore, a parameter must be essential for this controller or at least contribute to the overall goal of improving the user experience. A parameter that does not meet these criteria was excluded from further consideration. With this information, it is possible to discover if the modeling possibilities in the KNX ETS software allow modeling those that are relevant for thermal control. The discovered parameters were evaluated during multiple discussions with an expert in this field, who provided extensive feedback and approved the final selection.

In order to find a remedy for the problem formulated in the second research question, a MAS, as a proof of concept, was implemented. This system is the first iteration of the artifact in the Design Science Research framework to show the feasibility of the developed approach. To design the agents that are present in such a system, the methodology Multiagent Systems Engineering (MaSE)[DWS01] was used. As mentioned briefly, special attention has been given to how to make the system scalable, s.t. the installation of new physical components with their corresponding agents will not result in ample integration effort. The proof of concept was implemented to operate on a simulated building to provide fast and comparable results. To evaluate if the MAS facilitates effortless integration of new components during simulation, different changes to the system were made, and it was analyzed how much interaction with the system is necessary to include the new agent. Furthermore, the effectiveness of the controller was analyzed by computing the total discomfort and the total energy consumption. Thermal discomfort and energy demand are the considered performance indicators for quantitatively evaluating the controller's performance. Furthermore, typical scenarios that occur over the lifetime of a building were considered to provide a qualitative evaluation of the proposed system. These three scenarios include integrating new equipment, scaling the number of zones, including new agents, and replacing existing ones.

CHAPTER 2

# State of the Art

This chapter briefly describes the most relevant topics and the concepts used for the implementation are elaborated. To understand how the proposed system operates, the basic concepts of an ontology, a MAS, the WoT, and the parameters that can be used for controlling thermal comfort inside of a building are briefly explained. Ontologies help to create an unambiguous understanding of a system, s.t. all agents of a MAS, which might be implemented at different lifetimes of the building, have the same understanding of exchanged information as well as their capabilities. For describing these capabilities, the WoT can provide the means of a unified interface. The parameters used to model the behavior of the building are explained to understand which information is relevant when controlling different aspects of a building.

## 2.1 Basic Concepts of Ontologies

Ontologies help to represent knowledge and can be used to model systems. A detailed description of ontologies and their creation can be found in [AH11]. However, the most important contents are summarized here. Special attention must be given to unambiguity when modeling systems that are extensible. Suppose a *temperature value* is transmitted from a sensor to a controller in a BMS. For a newly added component to an already existing system, it might not be clear what this value means because it could be the current measurement value of a zone, a setpoint, or a control value for a heater. All of these values are temperatures, but they have different meanings. Similar to Object-Oriented Programming (OOP), they all derive from the class *temperature value*. Ontologies have some similarities to object-oriented programming in that they permit the definition of classes and the assignment of subclasses to them. In the previous example, one would define the class *TemperatureValue*, which has the subclasses *MeasuredValue*, *TargetValue* and *HeaterSetpoint*.

Furthermore, the classes must be related to each other. The simplest relation is the *hasSubclass*-relation. A relation is called an object property, and it is also defined in ontology. For instance, a class that represents a room in a building could have an object property *hasSensor* that allows it to be associated with different sensors, such as temperature or presence sensors. The *TemperatureValue*, *hasSubclass*, and *TargetValue* form a triple of the type subject, predicate, and object. These triples are created with the Resource Description Framework (RDF)[CWL14], which is a core utility of the Semantic Web, which is a promotion of the web to a semantic level.

In contrast to typically used web pages, the Semantic Web also links data presented on pages with a reference called Unique Resource Identification (URI), which is additionally machine-readable. By uniquely identifying the data displayed on websites, the proliferation of static references shall be avoided. Updates are, therefore, distributed to all pages that reference such a triple. Also, search engines support reading these RDF-triples to allow more accurate results. In the Semantic Web, ambiguity can be resolved by referencing an already-defined triple. As mentioned, ontologies have some similarities to OOP, but they also are different in many cases. OOP is only defined for a software system, while ontologies are publicly accessible, s.t., they are reused and extended. Ontologies rely on the fact that they are reused by others, as creating individual ontologies for every system would defeat the purpose of unambiguity. Furthermore, ontologies use an extension of RDF called Web Ontology Language (OWL). In [vHM04], additional features are presented, which include cardinalities, equalities and other properties.

As already mentioned, it is important for ontologies to be reused by others. Suppose each system was developed with a new ontology. Then, this would not fulfill the purpose of an ontology because different devices from different manufacturers would not be able to exchange data with each other. Ontologies must, therefore, be accessible to others s.t. they can be used consistently. This allows developers to include those that are appropriate for their domain and to reuse concepts that are used there. Known ontologies in the building domain are briefly described below:

- Brick [BBF$^+$16]: This ontology describes metadata in buildings. This includes, among other things, the equipment, the data points, and the structure of the building.
- QUDT [FAI15]: This ontology models a large set of quantities, units, dimensions, and types. It can be used, for example, for units of temperature values.
- SAREF [GCLPVD23]: This ontology can be used for modeling smart applications. SAREF offers several parts for different domains, such as Energy, Buildings, Smart Cities, and the SAREF Core. The latter contains concepts for the general modeling of smart applications.
- KNX Information Model [KNX23]: KNX is developing an ontology that should be automatically generated when implementing a KNX installation. This exported

ontology can be used to generate an API that enables the user to interact with the building by reading sensors, setting control variables, or extracting basic information about the building structure.

- Real Estate Core (REC) [HWKH19]: With this ontology, a building with its assets can be semantically described as classes about many building elements and types of equipment are provided. The Brick ontology also heavily relies on this ontology, by reusing these types instead of creating new ones.

A system can be modeled by combining different ontologies, usually a so-called domain-specific and standardized ontology or sometimes referred to as an *upper ontology*. The domain-specific ontology is mainly relevant to the system itself since it is highly specific for this system and an upper ontology that enables a common vocabulary [INC08]. Different ontologies are also used in the system presented in this thesis, but in this case, there are more than two. Still, they can also be divided into a generic ontology and a domain-specific ontology. The generic ontology is intended to ensure that the data exchanged is interpreted consistently. The ontologies mentioned above, as well as others, are used for this purpose. Another important ontology is the one that classifies the skills of the agents. This is explained in more detail in the next section.

### 2.1.1 Skills, Resources and Products

In order to build a modular agent-based system, the agents must be able to communicate with each other, even if they are not from the same manufacturer. To provide the means of unambiguous communication, an ontology can be used that describes a common dictionary s.t. agents can exchange knowledge about their capabilities. Therefore, creating this ontology is an important step that needs to be done carefully. In the system, agents are looking for other agents that can help to better solve their own goals, e.g., an agent controlling indoor temperature can look for knowledge about weather forecasts. Therefore, it is essential to describe the capabilities of the agents. These capabilities are made public by the agents so that other agents can use their knowledge. The ontology aims to fully describe the capabilities of the agents so that they can describe their required capabilities as well as their own skills. This description of requirements can also be found in the skill-based engineering of industrial automation systems [Dor20]. The aim here is to produce products faster and more cost-effectively by reusing machines. This method should make it possible to describe production steps as skills, s.t., various products that can make demands on the production facility when manufacturing them. These requirements should now be matched with the skills in order to provide suitable resources to carry out the production steps. These concepts are defined in more detail in [MBW$^+$18].

In skill-based engineering, a basic distinction is made between the product, which is manufactured through a sequence of production steps, a resource such as a robot, and a capability. The product goes through several production steps, which are described by various requirements. Resources represent machines that are significantly involved in production. The capabilities of the resources are also described semantically. To define the capabilities of the resources, domain experts create descriptions that computers can then interpret. Flowcharts are created to determine the requirements during production, and the required skills are defined for each manufacturing step. The challenge here is that the manufacturers of machines, and therefore also describers of the given skills and producers of products, could use different vocabularies. Ontologies offer a remedy here as they can be reused and expanded as required.

Another type of skill description can be found in [RWG+24], where the differences between agents and digital twins are explained. In order to classify the differences and similarities, the purposes, properties, and capabilities of each system are determined. The purpose describes the future state to be achieved and is divided into different categories: planning, scheduling, control, diagnosis, user assistance, monitoring, virtual commissioning, and process optimization. The properties are used to characterize the system after it has achieved its goals and are grouped into the following categories:

- Fidelity: Fidelity indicates how accurately the real system is captured by the constructed abstraction and how precisely it can be interacted with.
- Intelligence: An intelligent system can react appropriately to changes in the environment. Entities in the system must be able to decide how to respond to changes as they sense their environment.
- Autonomy: Entities with this property operate continuously without user intervention.
- Sociability: This property is assigned to entities that communicate with others. A key characteristic is that the participants exchange information.

Finally, a total of 24 different capabilities have been defined. These can be used to identify how the system will reach its goals. Some of them are presented below:

- Adaptability: The system is able to adapt to changes in the system itself.
- Condition Monitoring: To achieve the goal, the system can capture its environment.
- Mobility: Nodes are able to move within the network without losing their program state.
- Pro-activity: The system achieves its goals by facilitating a strategy like rule-based controllers.
- Prioritization: The system can prioritize different tasks and receive messages to solve its given problem.

This classification into purpose, properties, and capabilities allows a much more flexible design since no explicit processes need to be known, only the intention and capabilities of an agent. Therefore, this classification could also be used to define the capabilities of agents within a MAS. However, this flexibility comes with a certain degree of imprecision. For example, using this procedure to define the production steps would result in a less precise classification compared to the findings when using the skill-based engineering methodology mentioned before. Overall, a trade-off will be necessary to unambiguously define the capability of an agent's skills without reducing the flexibility of the system. This way, agents can be added or exchanged during the system's operation.

Besides using concepts like these for skill-based engineering, they can also be adopted for the purpose of creating an interoperable system. Instead of defining the skills required for the fabrication of a product, the capabilities of the individual agents in a MAS have to be specified. In the same fashion, agents will then be able to discover each other based on which information they require. For example, a temperature-controlling agent may also facilitate the information of a weather prediction agent by finding the corresponding skill provided by an agent.

## 2.2 Building Control with Multi-Agent Systems

MASs have found great support in building control as they can work on complex tasks by sharing their knowledge with other agents. One advantage of agents is that they are capable of learning about their environment [YSX+21]. This allows them to adapt to changes and also work under unexpected situations. In [DB00], a system is proposed that allows runtime reconfiguration. This property is fundamental as buildings are currently adapted to operate on RES, which is why equipment might be replaced at any time. As presented in [MMK24], control algorithms are becoming increasingly sophisticated. Model-based algorithms like MPC help to provide more accurate results by leveraging detailed mathematical equations that represent the building's thermal behavior. Model-free approaches are also used in MAS, as they do not require the tedious step of creating such a model. Instead, an agent perceives the environment and learns about how the actions it applies influence it. Nevertheless, approaches that do not require a model are based on a learning algorithm that also requires a thorough understanding of its underlying mechanisms. Both approaches are currently used in MAS, as they provide the means for learning, perceiving, and interacting with their environment as well as exchanging information.

A crucial asset of agents is their information exchange, which requires an unambiguous vocabulary. This is a challenge in many MAS, especially when a system is designed to be extensible. The Semantic Web offers a remedy to this problem as it provides the means for standardized communication [MKA+23, RCR17]. By exchanging information, it is possible to apply divide-and-conquer mechanisms that solve complex problems by delegating many small sub-problems to other agents. This is particularly important for a building energy management system, as the number of zones to control varies for each building. Therefore, a flexible and scalable solution is desirable. When implementing a MAS, these properties can be achieved by designing a structured communication [SMK24]. Nevertheless, many of the reviewed papers face the problem of not having a standardized way of interacting with the building. Therefore, this thesis also explores how a MAS can be deployed on existing infrastructure, such as KNX.

## 2.3 Multi-Agent System Development Methodologies

As building automation systems become more complex and have to take into account a greater number of variables, an approach is required that does not consider the complete problem at one point but distributes a subset of it to multiple units. One way to partition the problem is using a MAS that acts as a building energy management system. Splitting the problem allows the agents to solve each sub-problem themselves, and by communicating with other agents, the overall goal is solved based on the solutions of the individual agents. This approach, called divide-and-conquer, is a well-known approach in computer science and is often used for complex problems to achieve a solution faster through parallelism. Another benefit such an approach brings with it is the separation of different controllers. A centralized approach is less flexible as components cannot be removed or added as easily, which also reduces the ability to be contentiously operational even during the failure of one component.

A critical problem of the agents, which is not easy to solve, is their communication. This aspect is often not described in detail in classical system modeling methods and, therefore, requires a different methodology designed for MASs. This section explains the basics of MASs, and then the different methodologies for creating such are presented. Finally, one methodology has been chosen to be used for developing this system.

### 2.3.1 Background and Key Concepts

Agents are used in a wide variety of areas, which is why there are different definitions for them. In [DKJ18], various MASs were examined in order to establish a standardized definition:

> *An entity that is placed in an environment and perceives various parameters that are used to make a decision based on the entity's goal. Based on this decision, the entity performs the necessary action in the environment.*

In this definition, the entity is an agent that resides in an environment, e.g., a controller for optimal thermal comfort inside an office building. This environment can vary in complexity, depending on the parameters that the agent can read. An agent always tries to achieve a certain goal. This does not necessarily have to align perfectly with the overall system goal but contributes to the overall target by fulfilling its objective.

The agent can influence the system through its actions. For example, to control the valve position of a radiator, one essential parameter would be the zone identifier of the zone with the radiator to control. With this information, the agent gathers information on the preferred setpoint and how the temperature must be adjusted. However, the systems are not always as simple as here, as there are often influencing variables that cannot be predicted, and the agents must, therefore, continuously react to their environment. For example, when the sun shines through the window and increases the temperature inside.

Agents are often used in combination with artificial intelligence. Since agents can perceive their environment, they are also able to learn something about it. The agents can exchange the gained knowledge with other agents in order to solve the complex problem collectively.

In [PEFST13], agents are assigned the following properties: *Autonomy*, *reactivity* and *pro-activity*.

- *Autonomy*: This means that the agent can work independently, i.e., it pursues its own goals and is not dependent on assistance from components outside of the system to achieve its desired result.
- *Reactivity*: As already mentioned, an agent constantly reacts to changes in its environment in order to reach its goal and stay there.
- *Pro-activity*: Furthermore, an agent must achieve its goal without being told to do so, i.e., it is always searching for a better solution.

This also aligns if the first definition of an agent that states similar properties, apart from the autonomy. Nevertheless, this is an important skill of an agent.

Most of the time, a fourth characteristic is assigned to agents: *Sociability*. This means that agents can communicate with each other and exchange knowledge about their respective areas of responsibility [PEFST13].

Agents can be composed into multiple modules that interact with each other. An agent's structure in an energy management system has been defined in [WXHG07]. The agents can perceive the environment with sensors and monitor the received data to detect changes. In this case, the agent will decide how to react to this event and apply corresponding actions to the environment. Furthermore, an agent can exchange messages with other agents via a communication module. The structure is also depicted in Figure 2.1. This typical agent structure is later also used when implement the MAS.

MASs are capable of solving complex problems. The divide-and-conquer strategy divides the overall problem into smaller sub-problems. Each agent considers only one part of the problem, so fewer variables need to be considered for that part. A key feature for success is that agents are able to communicate with each other. In the review [MSC$^+$19], a detailed introduction to MASs in the energy sector is given, as well as the Agent Communication Language (ACL) is analyzed. Agents can use two different types of languages: a standardized ACL like the one defined by the Foundation for the

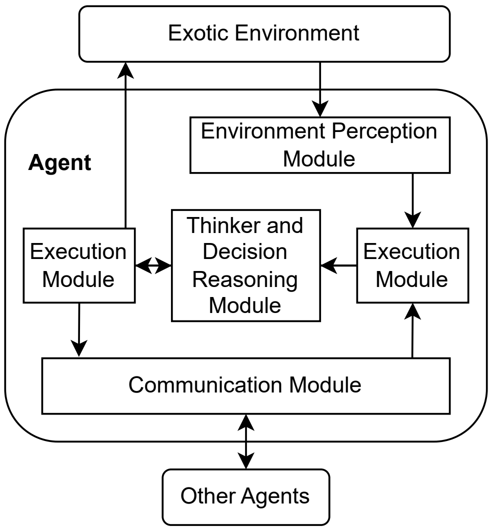


Figure 2.1: The structure of an agent in energy management systems [WXHG07].

Intelligent Physical Agent[1], which defines a syntax [FIP] that can be used to exchange messages, or relying on a principle that facilitates the use of existing or specifically created ontologies[Frü21]. The latter would make it possible to create a system that can be easily extended without knowledge of the standardized ACL, while the first solution has to be created during the design process with designated methodologies.

Furthermore, MASs can be easily extended. As all agents can communicate with each other, the system can also scaled without great effort. As described in[DKJ18], a MAS still differs from expert systems and object-oriented programs. Both are not as flexible as MASs because their communication partners are predefined, and object-oriented programs are often not designed to exchange semantic information.

These differences are also noticeable in the methodologies for developing complex systems. Since communication is essential in MASs, special attention must be given to this when designing such a system. For this reason, scientists have considered methodologies for developing MASs.

As briefly mentioned above, there are similarities between MAS and object-oriented programming. This is not the only area that has similarities to MASs, as a Microservice System (MSS) also has some similarities. It is, therefore, understandable that the question arises as to where the differences between the two systems lie and whether a MSS would not be more suitable. This question was also addressed in [WCOLO19], and

[1]http://www.fipa.org/

the differences were clarified. Both concepts divide the tasks into several sub-areas. In addition, separate resources are used to solve each sub-area, s.t. several computers or at least separate processes are required. Another key similarity is that both systems are scalable. This means that nodes can be added and removed as required. Another important feature of both systems that is essential for their function is the sociability of the participants. Agents exchange semantically enriched messages, while RESTful APIs are often used in a MSS. The autonomy of both systems is also similar, as both agents and microservices operate independently without human intervention.

However, as stated in [WCOLO19], MSS and MAS are already somewhat different when the reactivity of the two systems is compared. A microservice has been developed to answer Hypertext Transfer Protocol (HTTP) requests. An agent, on the other hand, perceives its environment and reacts to changes. Pro-activity is also a major difference between the two systems, as an agent acts continuously and does not wait to be notified. This is different from MSS because a microservice only starts to work when it receives a request to do so. Both concepts can also be combined, which is also done for the proof of concept.

### 2.3.2 Multi-Agent System Design Methodologies

As there are many different methodologies for designing a MAS, it is important to find the right one for the corresponding use case [Frü21]. The following section briefly describes three of them in order to explain why a particular methodology has been selected. Some of the most used methods are Gaia, MaSE, and PASSI.

#### Gaia

The Gaia methodology [WJK00] requires an agent or the system to fulfill several requirements before the system can be modeled:

1. Agents have enough capacity to solve the assigned problem. So they need to run on a computer or some other device.
2. Agents do not work against each other but follow a common goal.
3. The implementation is independent of the programming language or the system on which the agent acts.
4. The MAS is static, which means that agents cannot be added to the system during operation.
5. The functions of the agents do not change while the system is active.
6. Gaia allows you to design a system with a maximum of 100 different agents.

The basic process of Gaia can be seen in the figure 2.2. There are two phases in this method: the analysis phase and the design phase. The analysis phase aims to retrieve a description of the system based on the requirements. Then, *roles* can be defined along with *interactions* that there will be. Roles are comparable to activities that have to be carried out in the system. An agent can be responsible for one or more roles and take on the associated *responsibilities*, *authorizations*, *activities*, and *protocols*. The responsibilities define what should be achieved in this role, but also what should be prevented. Activities are used to specify the individual tasks that are necessary to fulfill the responsibilities. Finally, the protocols define how they communicate with other roles, as help from others might be necessary. Gaia will not be used for developing the MAS, as it requires agents to be static.

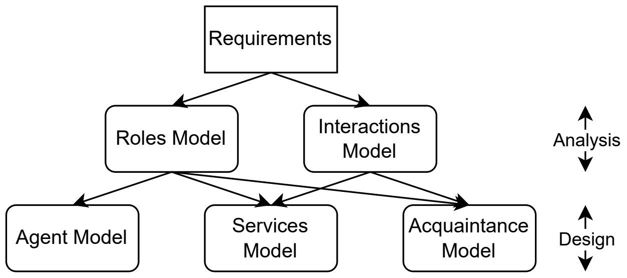


Figure 2.2: The fundamental process of Gaia[WJK00].

### MaSE

Another commonly used methodology for creating MASs is one called MaSE. As with Gaia, this method involves going through an analysis and design phase. In each phase, different steps are performed, most of which involve the creation of diagrams. The steps are usually performed in sequence. However, certain processes may be performed several times to consider the findings of earlier iterations. This might be necessary as some specialties are only noticed later. The phases of MaSE are described in detail in [DWS01] but are also briefly discussed below.

As can be seen in Figure 2.3, this methodology involves several steps. First, the goals of the system are defined and organized hierarchically. To give an illustration, the goal *optimal thermal comfort* could have the sub-goals *optimal temperature*, *optimal humidity*, and *optimal CO2 level*. The advantage of defining system goals is that they rarely change over time compared to other general descriptions of systems. The functional requirements serve as the first reference point from which the goals can be extracted.

The second step is to create use cases for the system. These are not based only on the previously defined goals, but they help if they are already defined, as the goals can be easily covered when defining the use cases. It is important to pay special attention to the possible communications in the system, as the interaction with each other is of particular

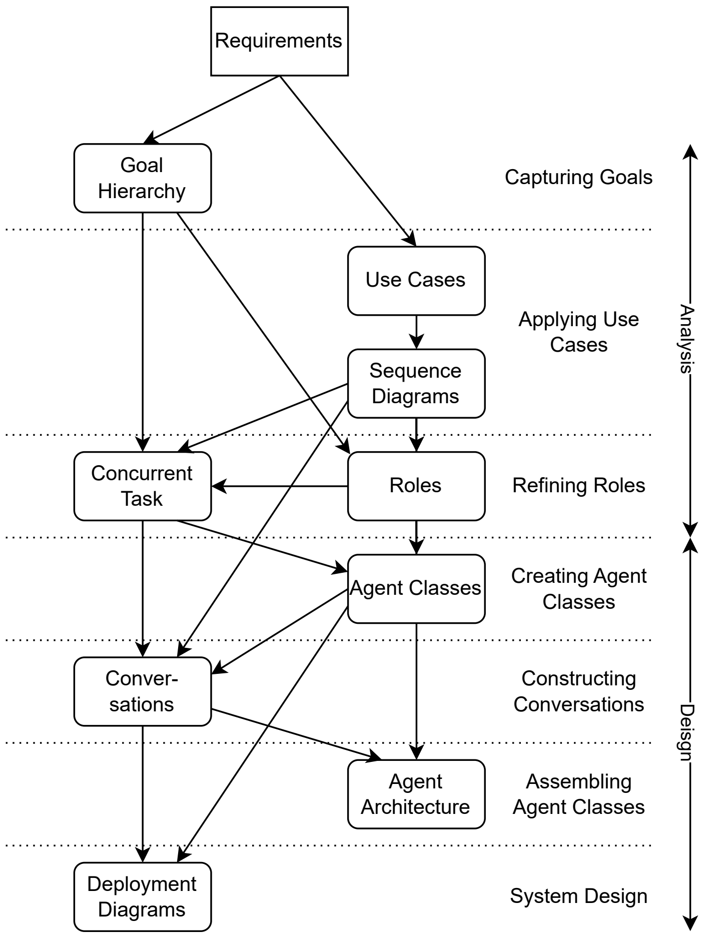


Figure 2.3: The phases of MaSE [DWS01].

interest. Sequential diagrams are created based on the use cases. Tentative roles are assigned to participants in the defined use case. For example, an interaction between *optimal temperature* and *weather forecasts* may be necessary.

The methodology's third step requires defining roles in the system. Similar to Gaia, roles can be thought of as assigned areas of responsibility. The tentative roles of the sequence diagrams help to create the actual roles. Roles are often directly assigned to a goal or at least extracted from one. This is also a requirement for completing this step, as more roles have to be assigned as long as some goals are not fulfilled by a role. Roles always consist of at least one task, which will also be specified in a task diagram. It is important to distinguish between persistent and transient tasks. In [DWS01], a transient task differs from a persistent task as the transient has to wait for a specific event. For example, an agent that controls room temperature continuously regulates the temperature, while an agent that controls outdoor lighting waits for an event, the detection of a person.

Once the analysis phase has been completed, there should now exist a MaSE Role Model diagram showing the roles, their assigned tasks, and the interaction between the roles.

The following steps are part of the design phase, which aims to create a complete system design. The first step involves creating the agent classes. In this phase, roles are combined into classes. The interactions between roles are maintained, but here, the relations represent a semantic message exchange between agents. External resources such as databases are also added to the diagram to obtain a complete view of the whole system. The semantic communications are represented in more detail in additional diagrams. There are always two diagrams required for each communication: one for the initiator of the message and one for the receiver of the message. This step also includes how an agent class reacts to erroneous messages. The last two steps describe the internal structure of the agents and finally model the system. The necessary agents are instantiated during these steps.

### PASSI

The Process for Agent Societies Specification and Implementation (PASSI) methodology is described in [CS14], but the essentials are summarized here. The phases are similar to those of MaSE, but each consists of multiple sub-tasks, which are also described in much more detail. This method is primarily used in large projects with many agents, as at the end of each phase, documents are created that summarize the results. These are then used in the subsequent steps. Moreover, graphs are created during this process, similar to MaSE, but here they are Unified Modeling Language (UML)-based graphs. The phases performed in PASSI are briefly described below:

1. System requirements: Initially, the use cases are identified and recorded in detail. The second task is to define the agents and assign them to specific functionalities within the system. The use cases serve as a basis for this. Roles and their tasks are then defined. It also holds for this methodology: An agent plays at least one role, which consists of at least one task.

2. Agent society: In this phase, the dependencies between agents are modeled. An ontology is created, which is used to exchange messages between agents. Information can be exchanged via existing protocols, or a new one is created and clearly described in this phase. Finally, the roles of the agents are recorded in a document. As agents can play different roles, it is defined here when the agent plays which role.

3. Agent implementation: As the name suggests, this phase documents how the agents are structured and how they behave. A distinction is made between the whole MAS and the individual agents.

4. Code: In this step, the agents are created based on the implementation. PASSI tries to keep the effort low by using reusable programs.

5. Deployment: Finally, the agents are instantiated in the system.

In the PASSI methodology, the individual tasks described above are always assigned to different people. For instance, a domain expert is required to specify the use cases once they have been identified. Therefore, modeling with this methodology is very detailed and thorough. It requires serval documents which is only essential for large projects. This is why MaSE has been chosen to model the MAS.

## 2.4 Web of Things

The WoT is a specification that is intended to help avoid the large fragmentation of IoT protocols. The standard has been developed to enable the vendors of IoT devices to offer a uniform interface to create simple interoperability. Devices are called *Things* and require a so-called Thing Description (TD). This description is a requirement as it defines how devices can interact with each other, as well as what capabilities a device has. Each interaction point can be referenced with a Uniform Resource Locator (URL). By uploading the TD to a central server, called the Thing Description Directory (TDD), other devices in the network can retrieve the TD of other Things in the network and gain knowledge on how to interact with others.

The following explanations are based on the detailed descriptions in [KKM23] but have been summarized here. The TD is created in a special JavaScript Object Notation (JSON) format called JSON for Linked Data (JSON-LD). Therefore, the TD always consists of key-value pairs, but with JSON-LD, it is possible to connect JSON and RDF. Consequently, linking to external resources inside of the TD is possible. This allows using a standardized vocabulary when defining the Thing's properties. Some mandatory attributes are required to create a valid description. One of these is *title*, to give an example. The specified value is only used by the end user to easily recognize the device. Another mandatory attribute is `@context`, which can specify a list of keys used in the TD. This list always refers to an ontology describing the attributes used in the file. A unique identification number is specified with the optional `id` field. The specified value corresponds to a URI.

The functionality of a Thing is defined using `properties`, `actions`, and `events`, which are also known as interaction affordances. Properties can be used to query an internal status, such as a measured temperature value. Actions and events can be used to perform an action or subscribe to a specific event. A useful value for the interaction is the `@type`, which semantically describes the returned value. The value specified here is usually defined within an ontology, which must be included in the context beforehand. Each interaction affordance requires an endpoint through which the corresponding service can be called. The `forms` attribute is therefore required for this. Not only a link but also further information about how the interaction can be defined here.

An example of a TD can be found in Listing 1. The context contains an additional attribute, namely `brick`, which is included in line 3. The value refers to the Brick ontology. The Thing has only one affordance to interact with: `getSetpoint` (line 11). As can be seen from the description of this affordance, it requires the specification of an input. The type of this input is of a type defined in the brick ontology, namely a zone identification number. The device responds with an ordinary number of type `brick_Air_Temperature_Setpoint`. The interaction must be started with the specified link and query the data with a `POST` method. The content of the message contains the zone ID encoded in JSON-LD format.

```
{
  "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    {"brick": "https://brickschema.org/schema/Brick#" }
  ],
  "id": "urn:uuid:12345-temp-cntrl-6789",
  "title": "Temperature Controler",
  "securityDefinitions": { "nosec_sc": { "scheme": "nosec" }},
  "security": "nosec_sc",
  "actions": {
    "getSetpoint": {
      "title": "Get setpoint for zone",
      "description": "Returns the setpoint for a given zone.",
      "input": {
        "zoneId": {
          "type": "string",
          "@type": "brick:Zone",
          "description": "The ID of the zone."
        }
      },
      "output": {
        "@type": "brick:Air_Temperature_Setpoint",
        "type": "number",
      },
      "forms": [
        {
          "href": "http://tempcontroller/actions/getSetpoint",
          "contentType": "application/json",
          "method": "POST"
        }
      ]
    }
  }
}
```

Listing 1: Example of a Thing Description.

A Thing can, therefore, be a server that can be interacted with using various HTTP methods. There are currently various frameworks that can be used to automatically create a server from a TD. One of these is node-wot[2], which is based on JavaScript. The advantage of this framework is that it allows the use of complex TD containing different contexts, thus incorporating the essential functionality of JSON-LD. As mentioned above, a TDD is used to publish the description of the device. WoT-Discovery is a key feature

[2]https://github.com/eclipse-thingweb/node-wot

that allows Things to find each other [MTCT23]. The simplest way to discover other Things is just to provide a link to the TD. Another option is to use a directory, the TDD. Here, all descriptions are simply added. Such TDD should also support updating a TD and avoid creating duplicates of the same instance.

Additionally,SPARQL Protocol and RDF Query Language (SPARQL) [SH13] helps to create complex queries on RDF to fetch only those TD of relevance. A server that implements all the required functionality is wot-hive[3]. Various HTTP methods can be used to add, remove, or modify TDs.

The combination of MAS and the WoT have already been researched in [RCR17]. In this paper, the behavior of animals is analyzed with multiple agents. One essential benefit of the WoT is that communication is standardized, allowing humans and computers to query information from the system quickly. As the interface is machine-readable and accessible through the Web, it is possible that a common goal can be divided and assigned to multiple agents and that the agents exchange results.

## 2.5 Relevant Parameters for Thermal Control Strategies

The parameters relevant to implementing different control strategies are presented in this section. It is important to understand which factors are relevant for most controllers, as this information has to be queried in the MAS. This data ranges from the characteristics of the building envelope to the type of heating system and occupancy predictions.

If MPC[DAC+20] is used to implement a control strategy, many parameters are required. These controllers facilitate a model to predict how inputs to the system would change the measured value. For example, when controlling the indoor temperature, a thermal model of the room is used to predict how inputs and external factors influence the temperature. Then, the inputs for the next $k$ time steps are calculated to achieve the desired temperature. After the first step, the calculations are repeated to account for errors.

When using MPC for controlling the temperature inside of a building, a thermal model of the whole building is used to predict how disturbances and heat inputs will change the indoor temperature. The most common approach when creating such models is an Resistor–Capacitor (RC) model, where thermal resistances and capacitances are used to create an analogous electrical circuit, allowing the use of electrical engineering methods when solving the problem. The concepts of RC modeling are now further elaborated.

A wall is constructed of multiple layers that act as an insulator and have the ability to store heat. These resistances and heat storage elements can also be interpreted as electrical resistances and capacitances. This allows the thermodynamic processes to be represented in an electronic system, where each construction layer represents an RC circuit. A series of RC components can represent an entire wall. The analogue of a

[3]https://github.com/oeg-upm/wot-hive

heat source is a voltage source that can also be modeled in the electronic circuit. The appropriate differential equations for the RC circuits must be identified and translated into a state space representation to determine the indoor temperature for a given outdoor temperature and heat load. The procedure is also explained in more detail in [BULA+19].

Usually, these sophisticated models can be avoided for simpler methods like rule-based controllers[BB14]. Nevertheless, it is essential to know the type of heating system present in the building since the rules for controlling a radiator differ from those for controlling an electric air heater. Therefore, the following section briefly summarizes the most relevant parameters. Many considerations are made for MPC because this control strategy is very effective and widely used in the scientific community and because simpler controllers rely on a subset of these parameters.

### 2.5.1 Parameters of Building Envelope Models

The thermal model of the envelope is a very important asset for thermal control. Therefore, developing such a model is an important step that can be carried out in different ways. A *white-box* model can be created if all physical parameters are known. Such models have high complexity, which makes it hard to use optimization algorithms. According to [PDKH17], such models are commonly only used in building energy simulation software as extracting these parameters is quite complex. Therefore, data-driven models called *gray-box* and *black-box* models are used instead. The black-box model automatically determines and continuously adjusts the parameters based on algorithms, which are also used to estimate these. The problem with black-box models is that they lack reliability because they are trained to work only for the inputs they were taught. If the real building receives abnormal inputs that are outside this range, the model may produce incorrect outputs. Gray box models are a combination of the above. They depend on a subset of the envelope parameters or linearize the state space to reduce complexity and also use learning algorithms to determine more relationships between parameters or unknown coefficients [AAC+15, DAC+20]. Hence, gray box models are commonly used for thermal models of buildings.

Creating a gray box model requires information about the building envelope, such as the dimensions, approximate resistances, and capacitances, which allows the learning algorithm to find the optimal solution without getting stuck at a local optimum. In [CFM+19], an RC model was created for a two-story building, which can be applied to any other building. Equation 2.1 presents a simplified typical heat balance equation used in a controller to calculate the air temperature within a zone.

$$\begin{aligned} C_z \frac{dT_z}{dt} &= \frac{T_{wall} - T_z}{R_w/2} + \frac{T_{amb} - T_z}{R_{win}} + \frac{T_{attic} - T_z}{R_{attic}} \\ &\quad + w_1 \cdot Q_{loss} + w_2 \cdot Q_{hvac} + w_3 \cdot Q_{sol} \end{aligned} \tag{2.1}$$

The air capacity of the zone $C_z$, the temperature inside the zone $T_z$, the temperature of the wall $T_{wall}$, the temperature of the attic $T_{attic}$, and the ambient temperature $T_{amb}$ are considered. Additionally, the thermal resistance of the wall $R_w$, the window $R_{win}$, and the attic $R_{attic}$, as well as the building heat loss $Q_{loss}$, the heating and ventilation gains $Q_{hvac}$, the solar gains $Q_{sol}$, and a weighting factor $w_i$ are included in the analysis.

Indoor and outdoor temperatures can be measured easily, while determining the wall temperature is not trivial. Some approaches try to estimate this state, while others rely on a measurement or prediction of it. As mentioned, the RC parameters can be provided by a white box model or a learning algorithm, but they still have to be determined at some point for the controller.

### 2.5.2 Gains Through Solar Radiation

An important factor when considering the indoor temperature of a building is the gains through solar radiation. These can vary heavily during different times of the day and the year. A very detailed analysis has been carried out in [EM15]. The most important findings are summarized next.

There exist three primary ways in which heat can be transferred from solar radiation to the internal mass inside of a building. This mass is represented by the walls and interior and the internal air, which heats up differently than the heavy mass. The ways how the heat is transferred are the following:

1. Direct transmission through the glazing.
2. Radiation emitted by the glazing due to thermal absorption of the window.
3. Convective heat is also released by the window as it absorbs radiation from the sun.

All of these methods heat heavy internal masses. Therefore, the surfaces themselves transfer heat to the indoor air temperature, which affects thermal comfort. The resulting equation, 2.2, describes the Solar Response Factor (SRF), which defines the total convective heat flow from the indoor air temperature.

$$SRF = SRF_t \cdot \tau_g + SRF_r \cdot r_{g,r} + r_{g,c} \tag{2.2}$$

The formula consists of the SRF by direct transmission and the radiation of the window, represented by $SRF_t$ and $SRF_r$. $\tau_g$, $r_{g,r}$ and $r_{g,c}$, which are the solar transmittance of the glazing, the solar energy absorbed by the glazing and then released as radiation and convection, respectively. $r_{g,r}$ and $r_{g,c}$ require information about the shading of the window, the area of the window, the solar radiation, and the G-factor of the window, which represents the solar transmittance through the glazing. $SRF_t$ depends on the absorption coefficient of the wall, the area of the wall, the thermal resistance of the material, the reflectivity of the material, and the surface factor, which is a complex number that describes the periodic and steady heat flow across the surface of the wall. $SRF_r$ also depends on the dimensions of the room, the area of the window, the radiant heat transfer coefficient of the glazing, and the complex surface factor of the window, in addition to the thermal resistance of the window. For the final estimation of the solar heat gain, a Fourier analysis must be performed. The equation 2.3 gives the Fourier series with period $P = 24$ hours and order $N$.

$$Q_{sol}(t) = A_g \cdot \left[ \overline{SRF} \cdot \overline{I}_g + \sum_{n=1}^{N} |SRF_n| \cdot |I_{g,n}| \cdot cos \left( \frac{2\pi n}{P} \cdot t + \varphi_{SRF_n} \right) \right] \tag{2.3}$$

$A_g$, $\overline{SRF}$, $\overline{I}_g$, $I_g$ are the area of a given window, the mean solar irradiance, and the measured solar irradiance.

Even more factors have to be considered, as the external walls and the roof are also exposed to direct solar radiation. These masses also gain, store, and transmit heat. How this can be modeled has already been explained in the previous section. Nevertheless, the solar effects have to be explained more thoroughly. In [CFM+19], the equations to include the solar gains are presented. These are also briefly explained here:

$$T_{sol,w} = \frac{\alpha_w}{h} \cdot F_w \cdot I_g + T_{amb} \tag{2.4}$$
$$T_{sol,r} = \frac{\alpha_r}{h} \cdot F_r \cdot I_g + T_{amb} \tag{2.5}$$

$\alpha$ and $F$ are the absorption coefficient and the view factor, respectively. The convective heat transfer $h$ can be estimated with the following formula [JWC96]:

$$h = 5.7 + 3.8 \cdot S_w \tag{2.6}$$

where $S_w$ is the current wind speed.

With this information, the RC model can be refined. The variables $T_{wall}$ and $T_{attic}$ given in equation 2.1 can be determined with the following equations:

$$C_w \frac{dT_{wall}}{dt} = \frac{T_{sol,w} - T_{wall}}{R_w/2} - \frac{T_{wall} - T_z}{R_w/2} \tag{2.7}$$
$$C_r \frac{dT_{attic}}{dt} = \frac{T_{sol,r} - T_{attic}}{R_{roof}} - \frac{T_{attic} - T_z}{R_attic} \tag{2.8}$$

### 2.5.3 Parameters of HVAC systems

Many controllers like the MPC can calculate the necessary heat input $Q$ for a zone. However, this value is not feasible to be given as input for, e.g., the KNX 3$^{rd}$ Party API as this requires a concrete control value, like a partial load or a valve position. Therefore, a controller placed in an actual building must calculate inputs that need to be applied. In the case of radiator heating, a valve position may be relevant, while in the case of Air Conditioning (AC) cooling, a percentage value representing the partial load may be utilized. The actual values depend on the Heating, Ventilation and Air Conditioning (HVAC) system, as well as its interface. Therefore, the heating system itself must be specified. For the purpose of this research, only radiator heating will be assumed, as it is one of the most prominent heating systems in Europe.

For this purpose, a heat curve has been derived in [LBW+22, PDKH17], which can be used to determine the supply water temperature for the heating emission system. The resulting equation is also given in 2.9, and the equation for determining the relative heat load is given in 2.10.

$$T_{sup} = T_{set} + \left(\frac{T_{sup,n} + T_{ret,n}}{2} - T_z\right) \cdot Q_{rel}^{1/m} + \frac{T_{sup,n} - T_{ret,n}}{2} \cdot Q_{rel} \tag{2.9}$$

$$Q_{rel} = \frac{Q_{cur}}{Q_{nom}} = \frac{T_{lim} - T_{amb}}{T_{lim} - T_{air,n}} \tag{2.10}$$

The supply water temperature $T_{sup}$, the zone setpoint $T_{set}$, the nominal supply water temperature $T_{sup,n}$, the nominal return water temperature $T_{ret,n}$, the current heat input to the zone $Q_{cur}$, the nominal heat load $Q_{nom}$, the heating limit temperature where heating is still required $T_{lim}$, and the nominal outdoor air temperature $T_{air,n}$ are considered in this equations.

The heating limit temperature is the minimal outdoor temperature at which the heating system can provide sufficient heat and typically varies with different climate zones.

Finally, a function to calculate the valve position has been presented in [TBB+21]. The equation to determine the control value is also given in the equation 2.11.

$$u_{valve} = \frac{1}{1 + e^{-\alpha_v(T_{set}+T_o-T_z)}} \tag{2.11}$$

$\alpha_v$ and $T_o$ are constants to modify the slope of the sigmoid function and the offset temperature that is used to account for temperature differences between the temperature sensor and the temperature at the valve. This function represents a sigmoid function. This comes with a problem as it never reaches 0 or 1. Nevertheless, this problem can easily be prohibited by rounding to a certain number of decimal points.

Additionally, ventilation and air infiltration, i.e., unwanted air exchange due to leakage, also play an important role, as presented in [CFM+19]. The presented formulas allow for this estimation of complex effects. These equations are also given below.

$$Q_{vent} = C_p \cdot V \cdot \rho \cdot (T_{amb} - T_z) \tag{2.12}$$
$$Q_{infli} = C_p \cdot C_w \cdot S_w \cdot (T_{amb} - T_z) \tag{2.13}$$

$C_p$, $V$, $\rho$, and $C_w$ are the specific heat capacity of air, the mass flow rate, the air density, and a coefficient to scale the wind speed, respectively.

### 2.5.4 List of all Parameters

Table 2.1 presents a list of all parameters. Some additional variables are listed, such as outputs from the controller to adjust actuators in the building. Also, the information about the user's preferred setpoint and energy costs are essential, as MPC implements a cost function that needs to be minimized. This function usually consists of the deviation from the setpoint and the total energy costs. Finally, it is shown whether parameters must be instantiated (I.) or whether they can be learned (L.). However, a starting condition is usually given to learn a parameter to guide the learning algorithm in the right direction. If parameters are known in advance, they obviously do not need to be learned and can be instantiated as well. The classification into these categories is not predefined and may differ from implementation to implementation. Some of the presented parameters are also identified in [GLW23], which includes a table with some relevant parameters.

| **Variable** | **Description** | **Unit** | **Source** | **I./L.** |
|---|---|---|---|---|
| $DNI$ | Direct normal irradiance | W/m$^2$ | Forecast | - |
| $DHI$ | Diffuse horizontal irradiance | W/m$^2$ | Forecast | - |
| $T_{out}$ | Outdoor dry-bulb temperature | °C | Forecast | - |
| $S_w$ | Wind speed | m/s | Forecast | - |
| $P_{equip}$ | Equipment power use | W | Prediction | L. |
| $Q_{occ}$ | Sensible occupant load | W | Prediction | L. |
| $O_z$ | Occupancy prediction for in zone $z$ | 1 | Prediction | L. |
| $R_{energy}$ | Rate for energy | €/kWh | Forecast | - |
| $R_{demand}$ | Rate for power demand | €/kW | Forecast | - |
| $\mu$ | HVAC heating efficiency | 1 | Setting | I. |
| $cop$ | Cooling coefficient of performance | 1 | Setting | I. |
| $Q_{HVAC\_lim}$ | HVAC power limit | W | Setting | I. |
| $T_{set}$ | Thermostat setpoint | °C | Output | - |
| $U_{boi}$ | Boiler load | 1 | Output | - |
| $T_{sup}$ | Supply water temperature | °C | Output | - |
| $u_{valve}$ | Control value for a valve | 1 | Output | - |
| $V$ | Mass flow rate of ventilation | kg/s | Prediction | L. |
| $R$ | Thermal resistances of building materials | W/K | Setting | L. |
| $C$ | Thermal capacitance of building materials | J/K | Setting | L. |
| $C_{air}$ | Thermal capacitance of indoor air | J/K | Setting | L. |
| $C_{im}$ | Thermal capacitance of internal mass | J/K | Setting | L. |
| $Z_{dim}$ | Dimension of a zone (length, width, height) | m | Setting | I. |
| $A_g$ | Area of the glazing | m$^2$ | Setting | I. |
| $T_{sup,n}$ | Nominal supply water temperature | °C | Setting | I. |
| $T_{ret,n}$ | Nominal return water temperature | °C | Setting | I. |
| $Q_{nom}$ | Nominal heat load | W | Setting | I. |

*Continued on the next page*

| **Variable** | **Description** | **Unit** | **Source** | **I./L.** |
|---|---|---|---|---|
| $T_o$ | Temperature offset between sensor and valve | °C | Setting | I. |
| $F$ | View factor for exterior walls, the roof, and all windows | 1 | Setting | I. |
| $G$ | G factor of glazing | 1 | Setting | |
| $\alpha$ | Absorption coefficients of materials | 1 | Setting | L. |
| $\rho$ | Reflectance of the internal and external walls | 1 | Setting | L. |
| $S$ | Surface factor of internal walls | 1 | Setting | L. |
| $w_i$ | Weighting coefficients | - | Setting | L. |
| $C_w$ | Wind speed scaling coefficient | - | Setting | L. |

Table 2.1: List of all relevant parameters.

These parameters allow the implementation of many different controllers. If the KNX 3rd Party API interface provides these, sophisticated controllers can be implemented without additional information.

## 2.6 BOPTEST

A helpful tool for evaluating the performance of a controller is BOPTEST [DJS+21], which is why it is also used for evaluating the proof of concept. It comes with several building models that can be instantiated with different scenarios[4]. These scenarios allow the user to simulate the thermal behavior of the building for a desired period, as well as different energy price modes. The system also allows the collection of data in the building that would usually be measured by sensors. In addition, input signals such as setpoints can be passed to the simulation via a RESTful interface[5]. The simulation is controlled with an endpoint that advances the simulation time for a specified number of seconds. Another feature of BOPTEST is that it calculates a set of key performance indicators ranging from thermal comfort dissatisfaction to total energy consumption, which can be used to evaluate the controller. Since many control strategies such as MPC also require predictions of the weather and occupancy, there is also an endpoint to query forecast data.

Another advantage of BOPTEST is that it allows users to work with a simulated building without developing a model, which is a highly complicated task and requires knowledge of the tools used for this purpose. As mentioned above, a RESTful API allows accessing the runtime environment. The emulator containing the simulated object can be instantiated with different test cases. For each test case, there is information about some building

[4]https://ibpsa.github.io/project1-boptest/
[5]https://ibpsa.github.io/project1-boptest/docs-design/index.html

parameters that can be used for advanced control strategies. After specifying which test case shall be simulated, a date is specified and a specific price scenario, that can either be constant, dynamic, or highly dynamic.

After the initialization of the scenario, the simulation can be advanced using the `/advance` endpoint by a desired number of seconds, which is specified using the `/step` endpoint. In order to apply control values to the simulated building, the corresponding values are passed when calling the *advance* endpoint. Since BOPTEST comes with a benchmark controller to compare the results of newly implemented controllers, the corresponding control signal must be overwritten if a value other than the one provided by the benchmark controller is to be applied. For example, if the radiator valve is supposed to be fully opened, the input `conHeaLiv_oveActHea_u` is set to 1 (as the allowed values range from 0 to 1), but also the overwrite-bit `conHeaLiv_oveTSetHea_activate` has to be set to 1. Note that the name of the input also indicates that the valve position for the radiator in the living room will be changed.

The key performance indicators can be queried at any time to determine the effectiveness of the controller. This can be done with the `/kpi` endpoint. Data relevant to controlling the indoor temperature, like the zone temperature, can not be queried directly. All current states are returned after advancing the simulation. Nevertheless, this interface allows thermal controllers to be tested very efficiently. Also, it allows for the comparison of different controls as their settings can be repeated as desired.

## 2.7 KNX

KNX is a standard used in building automation. It defines a protocol used to exchange data between components in buildings. Mainly equipment like heaters, air conditioning, blinds, and lights can be controlled with KNX. For communication, the devices use a bus that follows the KNX specification. Other devices, like agents in a MAS, can, therefore, not participate in the same network without the use of a gateway that translates incoming requests from the agents to KNX messages and vice versa for responses. The functionality of such a device is currently specified in the KNX-IoT 3rd Party API[6]. It allows devices outside of the KNX network to communicate with the KNX devices by sending HTTP messages to the KNX-IoT 3rd Party API Server, which is responsible for translating the incoming requests.

To be able to provide functions for each device inside of the building, the KNX designer models the system with a software provided by KNX called the ETS. Each physical device inside the building capable of KNX must also added to the software model. Furthermore, the functionalities are implemented, and communication between devices is specified. If, for example, a light switch should control a specific light actuator, then a *communication object* is created and assigned to both of the devices. This object is then used when an action at the light switch is detected. Sophisticated models can be implemented to reduce

[6]https://support.knx.org/hc/en-us/sections/4404385397010-KNX-IoT-3rd-Party-API

energy consumption and increase the automation of the building to achieve greater user comfort and a better user experience. Once the model is complete, it can be exported from the ETS and imported into the KNX-IoT 3rd Party API to allow other devices to access these communication objects.

The exported file semantically describes the modeled system. KNX created an ontology, called the KNX Information Model, that is used to describe systems that can be modeled with the ETS [KNX23]. This ontology has four primary parts:

- Core model: With this model, an asset, with its devices and the corresponding software, is defined. The software consists of an application program, which is hosted on a device and implements a functionality. Multiple points can be assigned to a functionality. A point is used to interact with a device by reading or sending data.
- Location model: This can be used to describe the model of the building, i.e., the floors, rooms, spaces, etc.
- Tag model: Tags are used to provide any additional information. For example, a device can have the tag `tag:EquipmentType` to provide information about which equipment it actually is. This can be used to differentiate between a switch, an actuator, and a vent.
- KNX model: This model is used to describe the concepts used inside the KNX system. It specifies how the devices are implemented inside of the ETS.

Therefore, the functionality of the KNX gateway is solely based on the actual implementation inside the ETS. When developing the KNX functionality in the ETS software, the designer can also provide additional data apart from the usual communication between the devices. The software allows the creation of a rough building model by creating spaces, floors, and rooms. These can also be assigned a function, which could be *kitchen* or *bathroom*, among others. Devices can then be assigned to specific rooms, and the communication objects that are used in the system can be linked to these arbitrary functions.

These additional modeling options do not affect the system's behavior, but within the KNX-IoT 3rd Party API, they will be used for the server that allows access from other devices. For example, when creating a model of the building, it is possible to query which devices are present inside a specific zone. Therefore, internal resources can be located based on their assigned room and functionality. External devices can then identify which functions are available based on the assignments. For some control strategies, this information might be necessary. A control system implemented as a MAS requires to know which rooms are controlled by an agent.

The current modeling options are quite limited. Only very trivial design methods are possible. For example, modeling the structure of the building is quite restricted. as it is not even possible to define a floor level or specify which rooms are adjacent. Furthermore, assigning devices to rooms is also not unambiguous, as it does not state if these devices are located in this room or control it. For instance, a lighting actuator might be located in a distribution box that is not inside the room it controls. As the development of the API is currently not finished, this problem might be solved in future versions. Nevertheless, to this date, this is an unsolved problem.

As KNX's main usage is creating building automation, and it is not certain that they will ever include typical information given in a Building Information Model (BIM), there is a gap between the generated ontology and the necessary data provided in section 2.5. Simple controllers, like rule-based systems, might not require additional information, as they can operate with very few details. However, more advanced strategies might require an additional source of information to be fully operational.

Nevertheless, every controller requires some data. In Table 2.2 is a list of all data that must be queried from KNX. In a later chapter, it is analyzed whether this data can be provided by KNX or if additional information needs to be inferred.

| **Name** | **Description** |
|---|---|
| Temperature-controlled rooms | An essential information is which rooms are actually temperature controlled. A unique zone ID can be used to query other databases containing additional information later. |
| Heating system | The heating system employed in the building (e.g., radiators and floor heating). |
| Cooling system | Determine which components are used to cool the building. |
| Sensor data | This comprises all observable data from the building. These include metrics such as occupancy load, current zone temperature, inputs from users like preferred setpoints, current weather measurements, consumed energy, and numerous others. This data must be queried from the ontology and serve as inputs to the controllers. |
| Actuator data | Outputs in Table 2.1 must therefore, also be applied via the KNX 3rd Party API, which requires them to also be given in the ontology. |

Table 2.2: List of parameters that should be provided by the KNX Ontology.

CHAPTER 3

# Proposed System Design

In this chapter, the implemented system is presented. The first section explains the modeling of the MAS. The MaSE methodology was used to discover a set of possible agents, as this is essential for describing which skills have to be modeled in an ontology. The development of the skills is explained in Section 3.3. A test case was chosen and implemented with the ETS, in Section 3.4. Furthermore, the semantic export is analyzed to determine what information can be described within the ETS software. Before the agents are finally implemented, the skills are embedded into the TD of each agent in section 3.5. The final section explains the implementation of the MAS. It presents how each agent is created and how they compute control values for the simulated building.

## 3.1 Use Case Description

The test building simulated by BOPTEST is depicted in Figure 3.1. It consists of eight zones, while only the living room *Liv*, the bathroom *Bth* and the three bedrooms *Ro1*, *Ro2* and *Ro3* are equipped with controllable radiators. The attic, which is not given in the picture, and the garage are unconditioned. At the same time, the hallway *Hal* does have a radiator, which cannot be controlled to make sure that a water flow is always possible through the emission system. The rooms that are equipped with a radiator also have air conditioning installed to cool the building during summer, as the weather is emulated from climate data given in Bordeaux. A gas-powered boiler is used to heat water inside the emission system. Additionally, a three-way mixing valve and an emission pump can be controlled.

The system design is depicted in Figure 3.2. It consists of the MAS, the ontologies, the KNX 3rd Party API, the KNX system, and the building. Furthermore, there is a component labeled BOPTEST, which is used instead of an actual building in the proof of concept. The MAS consists of multiple agents, each with a corresponding TD. The TDD is used to discover other agents, as each agent will publish its TD there. In the

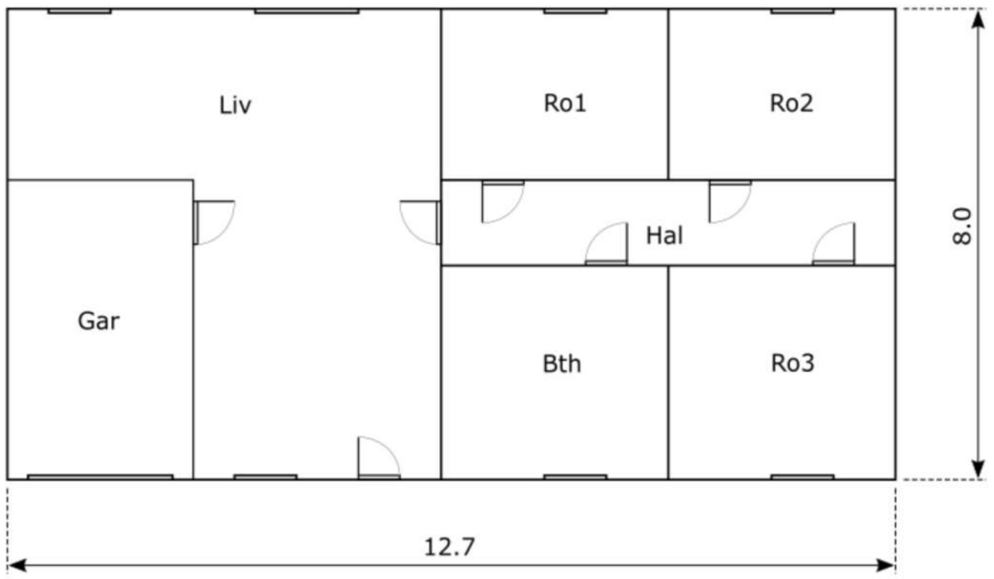


Figure 3.1: The multizone residential hydronic building from BOPTEST[1].

TD, an agent also describe its skills with the vocabulary defined in the *Skills Ontology*. To interact with the building, a KNX model is created, which consists of all actuators, sensors, zones, and communication objects used by the KNX components. This model can then be exported as an ontology, depicted as *KNX Ontology*. This ontology can be imported into the KNX 3rd Party Server, denoted with *KNX IoT API*. Currently, the KNX IoT Server only supports the provided example ontology. Therefore, the agents do not directly interact with the building, or in this case, the simulation; instead, they are are querying BOPTEST directly. In Figure 3.2, the building and the KNX 3rd Party Server are shown grayed as these components are not used in the implemented proof of concept. Nevertheless, the system is highly flexible, allowing the replacement of only the agent responsible for interacting with the building if the KNX IoT Server supports other exports. The component BOPTEST is also marked differently, namely with dashed lines, as it is used for testing purposes. A real-world implementation would not have this component.

## 3.2 Multi-Agent System Design

The first step involves the selection of a suitable MAS development methodology. In order to justify the choice of a particular methodology, it is necessary to have a basic understanding of the requirements of the desired MAS. It should include different agents with different tasks. The control of the internal zone temperature is of key interest. The occupancy of the room and the outside temperature should be taken into account, but only if data is available. This allows the agent system to be modular and different functions to be added or removed at runtime. This is a challenge as dynamic MAS

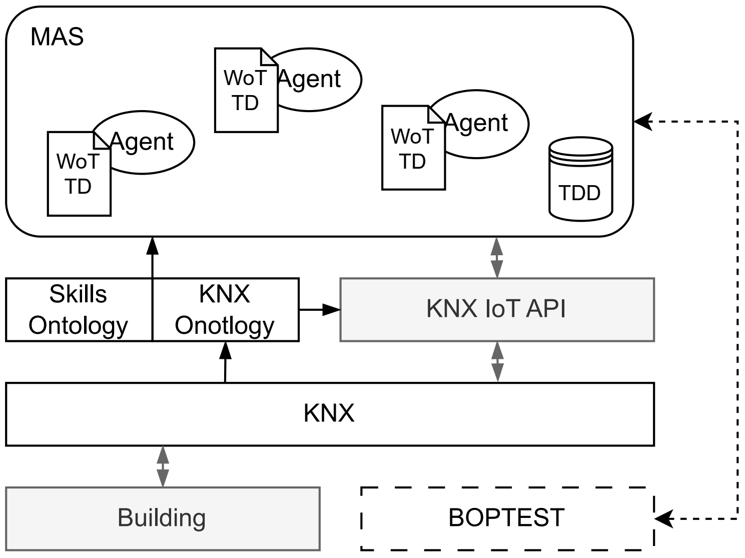


Figure 3.2: The proposed system design.

are not described in every methodology. As required by the Gaia methodology, such systems are unsuitable for modeling. The Gaia methodology also requires agents that do not change their abilities. Nevertheless, suppose one considers an agent capable of making predictions about the weather only if it has access to the necessary information. In that case, this agent is also a dynamic structure and, therefore, unsuitable for the aforementioned methodology.

With the PASSI methodology, on the other hand, the modeling process is very detailed and thorough. It also involves the creation of several documents along the process, which is essential for large-scale projects. For the required system, this imposes an overhead that is not as beneficial as it would be for extensive systems. Therefore, the MaSE methodology has been applied to construct a typical system that could be situated in a building automation system. The achieved system is then used to determine which agents are present. This step is crucial for developing the MAS and determining which skills have to be described in an ontology. With these descriptions of the skills, the agents inside the system can publish their abilities and search for other agents that might help them achieve their goals.

The methodology for discovering the relevant agents has been applied to a universal system consisting of many use cases. The agents that will later be implemented are only a subset of those to show how the developed ontology can be used.

### 3.2.1 Methodology execution

As mentioned in the chapter explaining the MaSE methodology, the first step is to discover the goals of the system based on the requirements. Multiple goals were considered to describe a wide range of skills later. One goal is that the system controls the user's comfort, e.g., illuminance, temperature, and CO2 levels. While doing so, it should also minimize energy consumption by preferably utilizing RES, but also utilize to other energy sources if necessary. This goal has the following subgoals:

- The system reaches maximal comfort for all occupants present, considering their preferred setpoints of all types.
  - A controller will determine the suitable control values based on a reconciled setpoint.
  - Disturbances and other influencing factors are predicted and considered during the calculation of the control values.
- The system optimizes energy efficiency.
  - To optimally utilize RES, a unit will have to predict the energy demands as well as the costs for it.
  - A controller will distribute the available energy inside the building to different consumers, i.e., batteries, electric vehicles, HVAC systems, and others.

These are just some goals of the system. Also, there are many additional subgoals to the already provided ones. The main functionality can be summarized by the following:

The system is able to control the zone air temperature, lighting levels, and ventilation. It determines the setpoints the users prefer and finds a consensus setpoint to achieve maximal comfort averaged over all occupants. Weather and occupancy forecasts are also taken into account. To reduce energy costs, the net price has to be determined. Furthermore, the power consumption of electric vehicles, HVAC, domestic water, and other equipment must be estimated to optimize the available resources across the building. To achieve this goal, the energy generation has to be predicted as well. An energy mode shall also be considered s.t. homeowners can save energy during more extended periods of absence. The complete goal hierarchy can be seen in Figure 3.3.

Although the goals stating optimal occupancy comfort and energy efficiency are very vague, they cover all of the essentials for implementing a BMS. One additional goal is very important to the success of this system: *The system has to be flexible enough s.t. it can adapt to changes in the building.* A possible use case is that a building is retrofitted with photovoltaics, and the generated energy needs to be allocated throughout the building as well as considered during the calculation of the energy costs. The main functionality that helps to achieve this goal is a TDD. In this case, agents can publish their TD, which contains information about their skills to allow other agents to discover them.

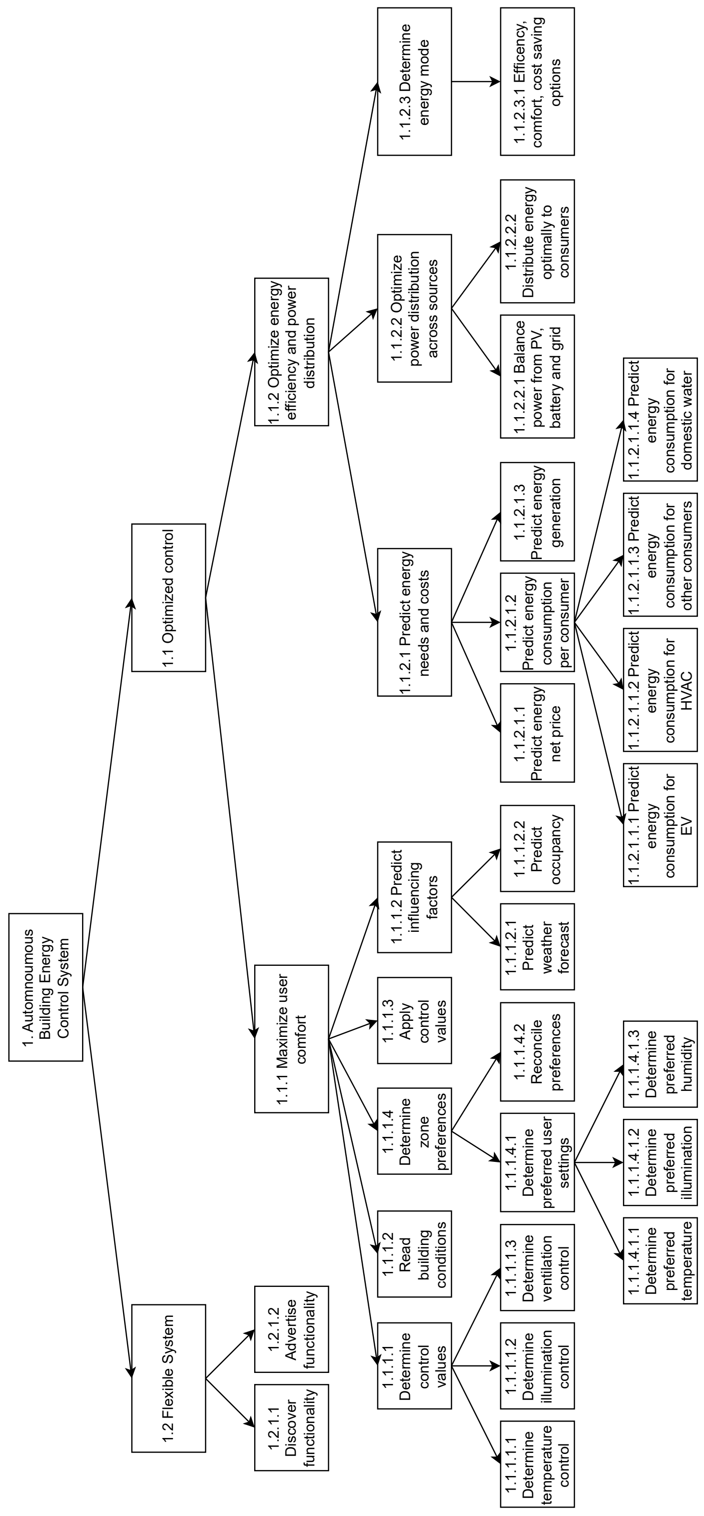

1. Autonmoumous Building Energy Control System
1.1 Optimized control
1.2 Flexible System
1.2.1.1 Discover functionality
1.2.1.2 Advertise functionality
1.1.1 Maximize user comfort
1.1.2 Optimize energy efficiency and power distribution
1.1.1.1 Determine control values
1.1.1.2 Read building conditions
1.1.1.4 Determine zone preferences
1.1.1.3 Apply control values
1.1.1.2 Predict influencing factors
1.1.2.1 Predict energy needs and costs
1.1.2.2 Optimize power distribution across sources
1.1.2.3 Determine energy mode
1.1.1.1.1 Determine temperature control
1.1.1.1.2 Determine illumination control
1.1.1.1.3 Determine ventilation control
1.1.1.4.1 Determine preferred user settings
1.1.1.4.2 Reconcile preferences
1.1.1.2.1 Predict weather forecast
1.1.1.2.2 Predict occupancy
1.1.2.1.1 Predict energy net price
1.1.2.1.2 Predict energy consumption per consumer
1.1.2.1.3 Predict energy generation
1.1.2.2.1 Balance power from PV, battery and grid
1.1.2.2.2 Distribute energy optimally to consumers
1.1.2.3.1 Efficency, comfort, cost saving options
1.1.1.4.1.1 Determine preferred temperature
1.1.1.4.1.2 Determine preferred illumination
1.1.1.4.1.3 Determine preferred humidity
1.1.2.1.1.1 Predict energy consumption for EV
1.1.2.1.1.2 Predict energy consumption for HVAC
1.1.2.1.1.3 Predict energy consumption for other consumers
1.1.2.1.1.4 Predict energy consumption for domestic water

Figure 3.3: The goal hierarchy of the proposed MAS.

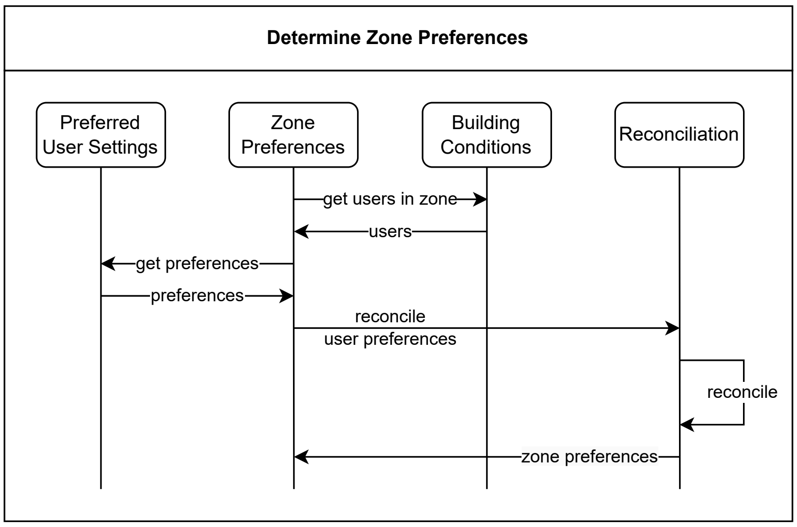


Figure 3.4: The sequence diagram depicts how the zone preferences are determined.

The next step in the MaSE methodology is to apply use cases, which are then depicted in sequence diagrams. Here, tentative roles will be defined to indicate interaction between roles. Figure 3.4 shows such a sequence diagram. It depicts the steps carried out to determine the zone preferences. The role responsible for the zone preferences must know which users are in the zone. Therefore, the first step is to get this information from the building conditions role. This role is the interface to the building and has knowledge of all the current sensor and actuator states. Here, the role is not named *Building Interface*, even though this would seem more intuitive, as in this step of the methodology, only roles are considered. The actual software components are extracted in subsequent steps. The zone preferences role queries the building conditions role to determine the preferred setpoint of each user present in the zone. The zone preferences role now interacts with another role called *Reconciliation*, which can reconcile multiple preferred setpoints to determine a harmonized setpoint for a given zone. After this step, any other role can query these calculated zone preferences and use these for their computations.

Another important sequence of interaction is that used by the role that computes the control values, which is depicted in Figure 3.5. It fulfills the goal of maximizing the comfort and minimizing energy costs. To accomplish this, it has to acquire information from multiple other roles. The first step is to determine the current discomfort, which is done by querying the building conditions role to retrieve the current settings, i.e., indoor temperature, humidity, and illuminance levels. Also, current and future predictions about energy prices are acquired. After receiving each user's preferences, the role can finally compute the current discomfort. Forecasts about the weather are also fetched before computing the subsequent control values. Finally, the occupancy and the energy mode are considered, and the next inputs are determined.

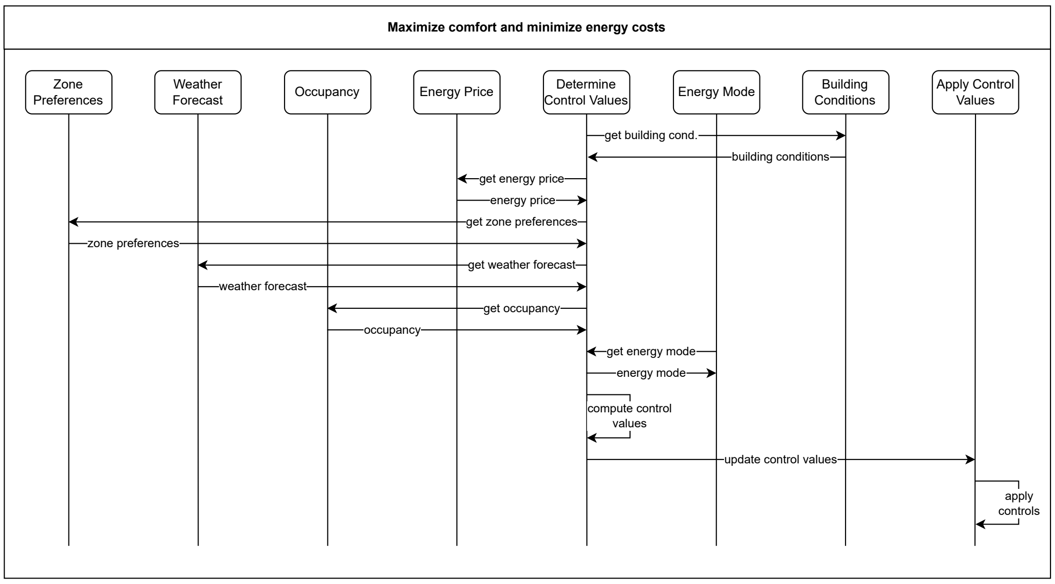


Figure 3.5: The sequence through which the role that determines the control values has to go through.

Note that most of the roles are optional, so if, for example, no weather predictions are available, determining the control values will not fail. Instead, the agent responsible for this role will continue to use default values or work around the missing data completely. Moreover, the actual data that is exchanged may also vary. For instance, the occupancy agent could only provide information about the current occupants or additionally provide predictions, which again could be based on previous observations or fixed schedules set by the users. Nevertheless, once the control values are computed, the role sends its updates to the role responsible for applying them to the actuators.

Every goal given in the goal hierarchy depicted in Figure 3.3 has to be covered by at least one sequence diagram. Nevertheless, not every goal must be covered by an explicit sequence diagram, as some goals are combined into one sequence diagram. For example, the sequences are identical for determining the preferred temperature, illuminance level, and humidity.

In the third phase of MaSE, the roles are defined. As roles have already been used in the previous step, the task here is to verify that the tentative roles actually cover all the goals. Furthermore, roles can be combined or split into multiple ones if necessary. As this step is closely related to the previous step, new interactions between roles might be found, which entails that the sequence diagrams have to be revised. The resulting role model, given in Figure 3.6, depicts the roles that have been discovered. Arrows between roles indicate an interaction between them, e.g., the *determine control values* role interacts with roles named *energy needs and costs*, *occupancy*, *energy mode*, *zone preferences*, *predict weather* and *building conditions*, which has already been presented in the previous step when explaining the sequence to determine the control values. Each role also specifies which goal it covers. For example, the role that acts as the building interface, namely *building conditions*, covers the goals *read building conditions (1.1.1.2)* and *apply control values (1.1.1.3)*.

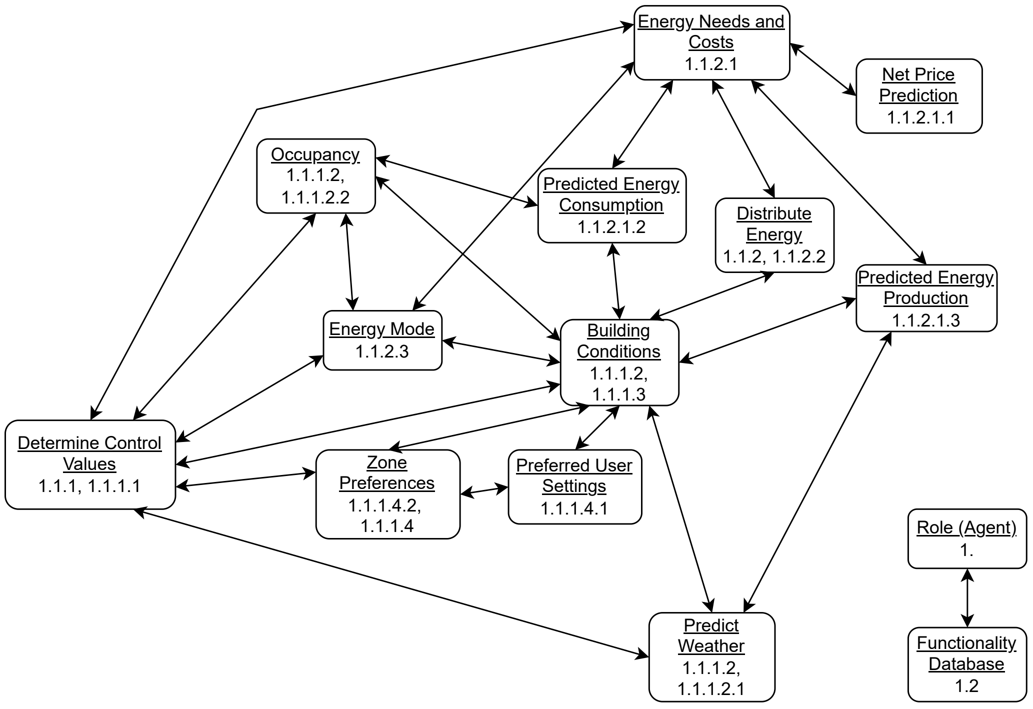

Energy Needs and Costs 1.1.2.1
Net Price Prediction 1.1.2.1.1
Occupancy 1.1.1.2, 1.1.1.2.2
Predicted Energy Consumption 1.1.2.1.2
Distribute Energy 1.1.2, 1.1.2.2
Predicted Energy Production 1.1.2.1.3
Energy Mode 1.1.2.3
Building Conditions 1.1.1.2, 1.1.1.3
Determine Control Values 1.1.1, 1.1.1.1
Zone Preferences 1.1.1.4.2, 1.1.1.4
Preferred User Settings 1.1.1.4.1
Role (Agent) 1.
Predict Weather 1.1.1.2, 1.1.1.2.1
Functionality Database 1.2

Figure 3.6: The role model of the MAS.

Another important interaction is displayed in the bottom right corner of Figure 3.6. As each agent in the MAS has to publish the skills it is capable of, a role called *Functionality Database* is introduced. Every other role interacts with this role by publishing its skills and discovering other skills it might need. For clarity, arrows connecting every other role to these two are neglected in the figure.

The next step during the MaSE methodology is to define the tasks each role has to perform. Each role has to have at least one task assigned to it. Figure 3.7 gives an example of such a task. This task is executed by a role named *Zone Preferences*. The syntax for a transition is as follows: *trigger [guard] ^ transmission*. The first transition is executed once the trigger `receive(getZonePreferences(zone), initiator)` is received. The role essentially waits until another role, called initiator, triggers the *getZonePreferences* function for a given zone. Then, the agent executing this task will send a transmission to the agent responsible for the role called *BuildingConditions* to receive information about the users in the given zone. If no response is received, the agent will try to execute this request again. After the users are finally known, the preferred setpoint is queried from the *UserManagment*-role. Once all preferred setpoints are known, the agent reconciles all setpoints to provide a collective setpoint that minimizes the discomfort of all users later. It could also consider different user priorities if necessary. After the reconciliation process, it sends an answer to the initiator containing the determined setpoint and waits until a new request is received. Similar tasks are created for humidity and illuminance levels.

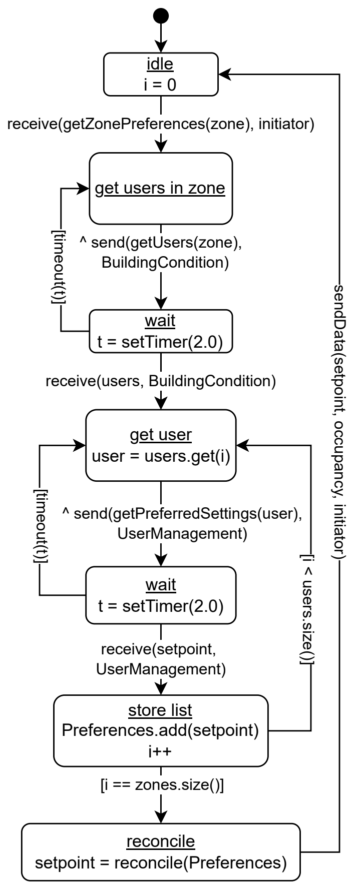
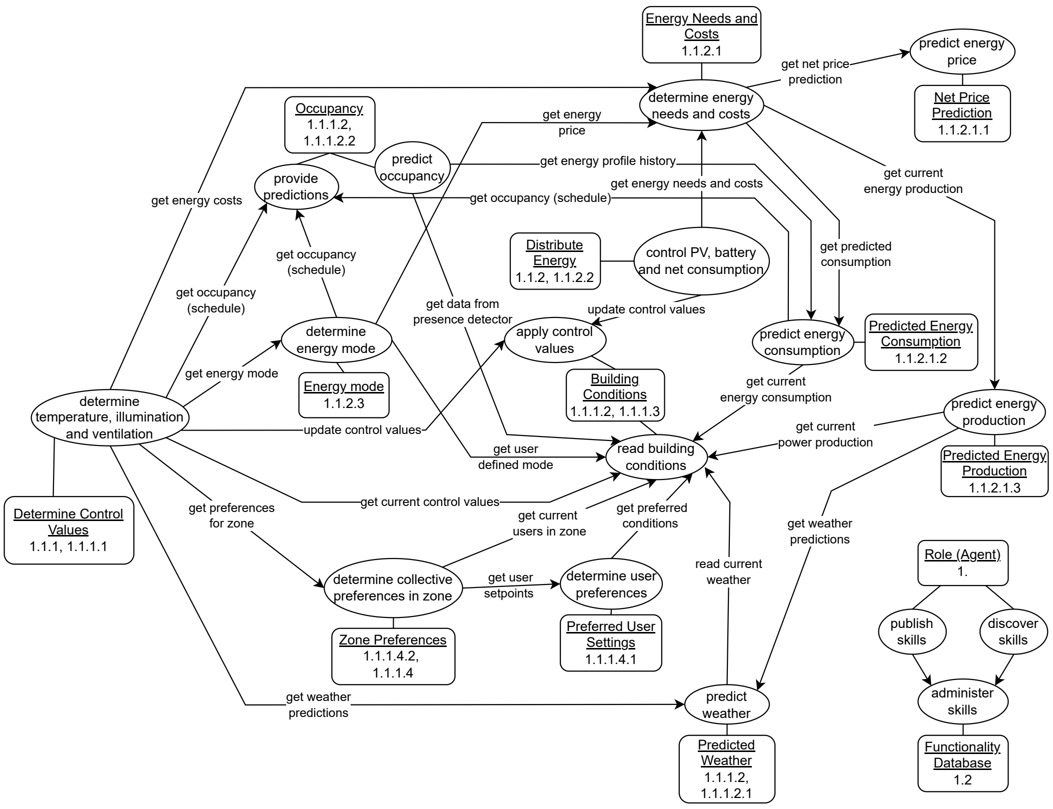

idle
i = 0
receive(getZonePreferences(zone), initiator)
get users in zone
^ send(getUsers(zone), BuildingCondition)
[timeout(t)]
wait
t = setTimer(2.0)
receive(users, BuildingCondition)
get user
user = users.get(i)
^ send(getPreferredSettings(user), UserManagement)
[timeout(t)]
wait
t = setTimer(2.0)
receive(setpoint, UserManagement)
store list
Preferences.add(setpoint)
i++
[i < users.size()]
[i == zones.size()]
reconcile
setpoint = reconcile(Preferences)
sendData(setpoint, occupancy, initiator)

Figure 3.7: The task executed by the zone preferences role.

Note that the agent who executes this task will not proactively calculate new setpoints. Therefore, this is an indication that the agent performing this task is a microservice rather than an agent. As noted in section 2.3.1, the main difference between an agent and a microservice is that a microservice waits for incoming HTTP requests before it starts operating in contrast to an agent.

To complete the analysis phase of MaSE, the MaSE role model is created. It provides an overview of all roles and their corresponding tasks. The result is depicted in Figure 3.8. In this picture, arrows indicate that there is communication with another task during the execution of a task. The task from which an arrow comes is the initiator of a conversation, and the task to which it points is the responder.



Energy Needs and Costs 1.1.2.1
determine energy needs and costs
get net price prediction
predict energy price
Net Price Prediction 1.1.2.1.1
get current energy production
Occupancy 1.1.1.2, 1.1.1.2.2
predict occupancy
provide predictions
get energy price
get energy profile history
get energy needs and costs
get occupancy (schedule)
get energy costs
get occupancy (schedule)
get occupancy (schedule)
Distribute Energy 1.1.2, 1.1.2.2
control PV, battery and net consumption
get predicted consumption
get data from presence detector
update control values
determine energy mode
apply control values
predict energy consumption
Predicted Energy Consumption 1.1.2.1.2
get energy mode
Energy mode 1.1.2.3
Building Conditions 1.1.1.2, 1.1.1.3
get current energy consumption
determine temperature, illumination and ventilation
update control values
get user defined mode
read building conditions
get current power production
predict energy production
Predicted Energy Production 1.1.2.1.3
Determine Control Values 1.1.1, 1.1.1.1
get preferences for zone
get current control values
get current users in zone
get preferred conditions
get weather predictions
read current weather
determine collective preferences in zone
get user setpoints
determine user preferences
Role (Agent) 1.
publish skills
discover skills
Zone Preferences 1.1.1.4.2, 1.1.1.4
Preferred User Settings 1.1.1.4.1
administer skills
get weather predictions
predict weather
Predicted Weather 1.1.1.2, 1.1.1.2.1
Functionality Database 1.2

Figure 3.8: The MaSE role model.

For this system, assigning agents to roles is a one-to-one mapping. Nevertheless, the subsequent steps of the MaSE methodology are skipped as the implemented system shall only be a subset of the described role model to prove the concept and discover skills agents might have. Furthermore, the agents will be instantiated as WoT devices. Hence, the communication between agents does not need to be designed explicitly as the WoT specification takes care of how this is done. The agents that are implemented are the following:

- Building interface agent: An agent responsible for reading all sensor data and applying the control values. As the implementation will work on BOPTEST, this agent also advances the simulation, as explained in the later sections.

- Weather prediction agent: As the name indicates, this agent predicts the weather forecast.
- Occupancy agent: This agent's task is to predict the occupancy behavior and provide the data to the agent, computing the control values.
- Zone preferences agent: The agent will provide a setpoint that shall be achieved in a given zone.
- Determine control values agent: Such an agent is created for each zone. It uses predictions from the occupancy and weather agents if available. The agent then computes the next control values that shall be applied based on the desired zone preferences.

## 3.3 Skills Ontology Modeling

This section examines how the skills of an agent can be properly described. The potentially present agents in an energy-optimizing system have been discovered in the last section. As mentioned in Section 2.1.1, there must be some tradeoff between flexibility and unambiguity when describing the skills. When developing the ontology used to describe the capabilities of the agents, a combination of the two described approaches was used to create a flexible agent description that still provided the required unambiguity. Furthermore, a helpful resource when developing the ontology was the sequence and the task diagrams determined in the previous section, as they highlight the interaction between agents.

By facilitating the skills ontology, the desired outcome is that agents can be added, removed, or replaced at system runtime. Therefore, it must also be possible to semantically describe the qualifications of agents that are added at a later point in time. Moreover, it should be possible for new agents to be discovered and interacted with by existing agents. For this reason, the description with capabilities, as in [RWG+24], is very suitable, as this system also consists of agents with different functionalities. For example, an agent might be capable of *predicting* the weather or *controlling* the indoor temperature.

For successful operation, this information must be offered by any agent and used by the temperature controller. Therefore, it has been modeled that an agent offers different capabilities but can also search for them at other agents. The matching of required and available skills is similar to the concept presented in [MBW+18].

Capabilities alone might not offer complete unambiguity, and consequently, another piece of information is modeled: A context allows the support of different types of agents. A tuple is created, which consists of the context and the capability. An agent could have the capability *predict*, but with the context *energy price* or *PV production*. Therefore, An agent publishes one (or more) context-capability pairs and searches for them to increase its performance.

Note that designing the ontology and creating the MaSE models were an interlinked process, i.e., discovered capabilities or contexts were added to the MaSE model, and roles elaborated during a MaSE phase, influenced the development of the ontology. For example, there are agents that can predict the weather or the net price. Therefore, the context *Weather* and *EnergyPrice* will be modeled in the ontology. When it comes to price predictions, one can distinguish between the different energy types. Consequently, a class *EnergyType* is created that has the following sub-classes: *Gas*, *Electrical*, *Net*, *Solar*, and *Hydrogen*. Moreover, the building shall also be capable of determining the temperature, illumination, and ventilation, as given by goal *Determine control values (1.1.1.1)*. The following considerations have been made to describe these contexts:

- Space heating and air conditioning can be used to bring the indoor temperature in a desired range. Therefore, corresponding contexts (*Space Heating* and *Air Conditioning*) are created. These are later combined with the capability *External-SystemControl* to indicate that this agent can control, e.g., the air conditioning.
- To control the illumination, the lights inside a zone and the shutters can be used. Therefore, a context called *Lighting* is created that has two sub-classes: *Shutters* and *Lights*.
- A class called *Ventilation*, that is simply used for agents that are related to this context.

The defined classes are also depicted in Figure 3.10. In this figure, all contexts and capabilities of the ontology are displayed. An agent can then use any combination of these to describe its skills.

As the presented MAS does not consider all possible automation areas in a building, other typical sub-areas of a Building Automation Management System have been analyzed. The usual areas are shown in Figure 3.9. These include the control of lighting and HVAC systems, but also other services such as alarm systems and access control. For example, light control classes have been previously determined, but others like *FireSafety* or *Window* have been added to the ontology. Also, searching through existing ontologies like Brick[BBF+16] helped to discover further contexts, as these contain classes for the typical equipment inside of buildings.

The discovered capabilities are mainly derived from [RWG+24]. However, not every single one of them is described there. Others are also significant when describing typical agents. Moreover, not every capability given in [RWG+24] is also modeled in the proposed ontology. For example, in this paper, *context awareness* has been defined, which would not be a meaningful property of any agent in this case, as it is very ambiguous. As can be seen in Figure 3.10, the capabilities that are going to be used for describing skills are

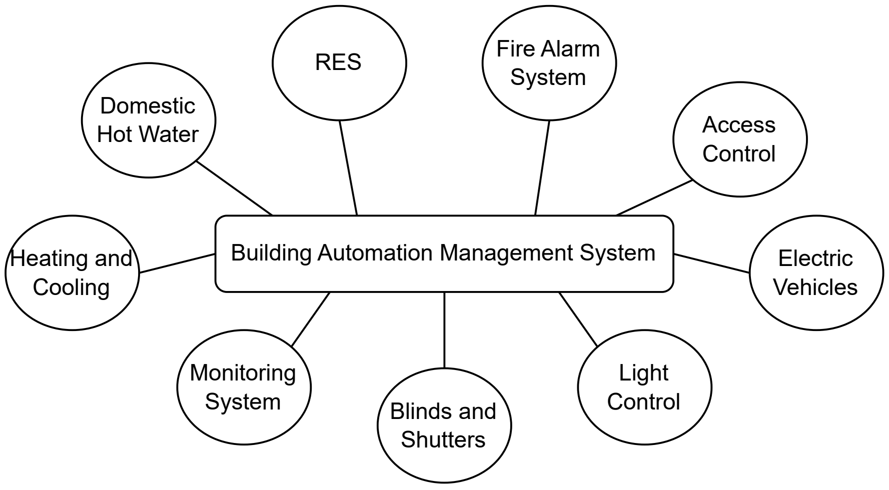


Figure 3.9: Components of a Building Automation Management System.

the following: *computation*, *condition monitoring*, *data storage*, *external system control*, *learning*, *negotiation*, *predict*, *prioritization*, *proactivity*, and *trading*.

To show how an agent is related to the defined classes, an agent class and the one representing a context-capability pair are also shown in Figure 3.10. The object properties, i.e., the relationships between the agents, are shown with arrows. The *has subclass* property is shown in blue. In addition, an orange and gray object property between the agent and the context-capability-pair represents *discovers* and *advertise*, respectively. Any number of pairs can be assigned to an agent with these relationships.

Moreover, object properties exist between the context-capability pair and the two classes named Context and Capability. This is used to assign any available context and capability to every pair. For example, an agent could discover a particular pair, which has assigned the context *EnergyPrice* and *Electrical*, as well as the capability *Predict*.

Using sequence diagrams like the one depicted in Figure 3.5, the derived contexts and capabilities can be applied to check whether the ontology allows modeling the skills of the discovered agents. The agent in the role of determining the control values has to discover agents with the following context-capability pairs:

- $\langle \text{BuildingControl}, \text{ConditionMonitoring} \rangle$: To gain access to current conditions inside of a zone, an agent has to be discovered that advertises this pair.
- $\langle \text{EnergyPrice}, \text{Predict} \rangle$: This pair is given by an agent that can predict energy prices. This skill can also be further refined to distinguish between gas or electrical prices by adding another context to this pair.
- $\langle \text{Setpoint}, \text{Negotiation} \rangle$: To discover an agent that provides a negotiated setpoint among all users in the zone.

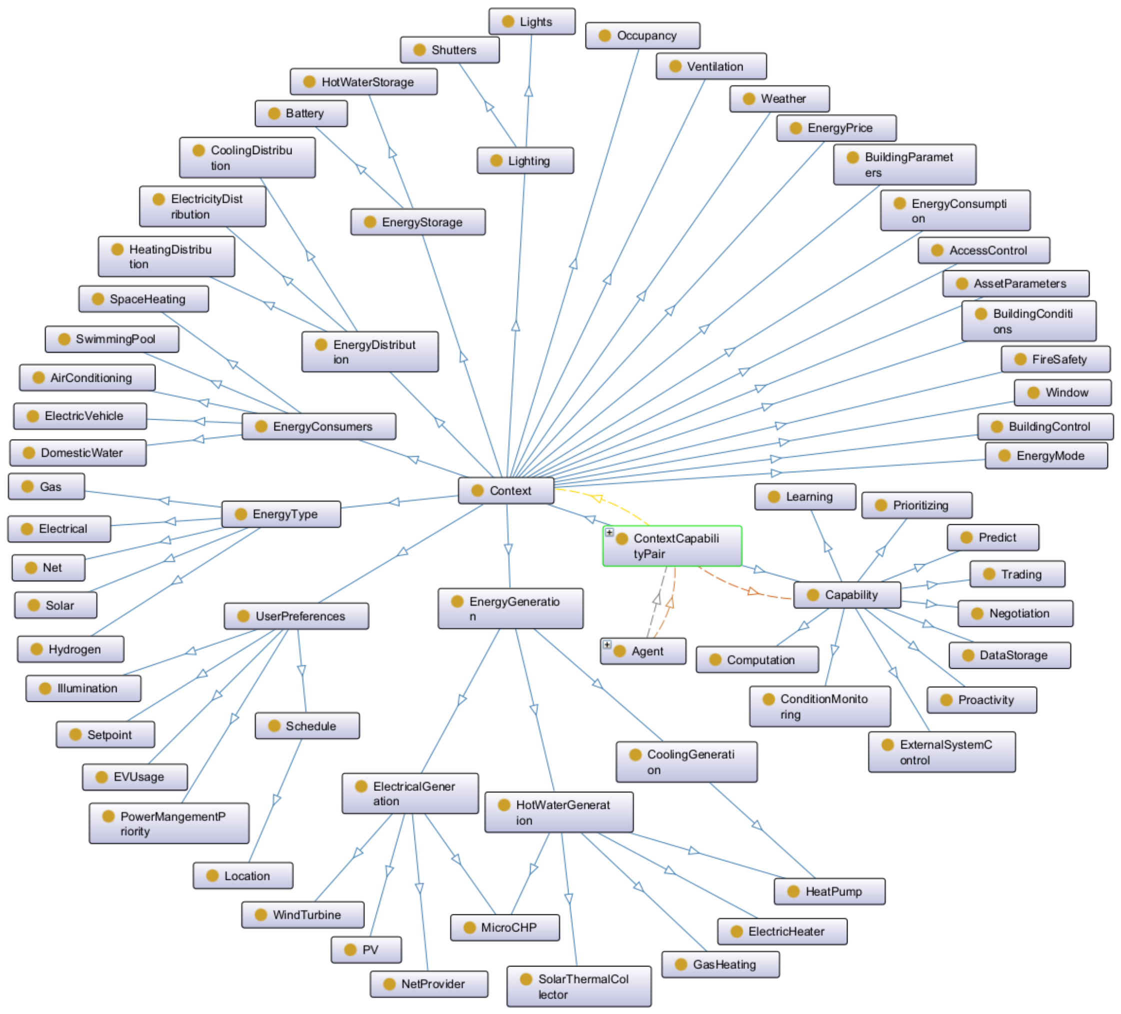


Figure 3.10: The ontology used to describe an agent’s skills in the MAS.

- ⟨Weather, Predict⟩: As the agent should include weather forecasts, this pair is discovered. An analogous pair can be created to represent the skill that enables the prediction of occupancy.

- ⟨EnergyMode, Computation⟩: The agent that computes the energy mode advertises this pair.

- ⟨HeatingDistribution, ExternalSystemControl⟩: This pair is given by the building interface as it allows the control of the actuators responsible for heating distribution.

## 3.4 KNX Semantic Modeling

As mentioned in section 2.7, the ETS software does not allow a multiplicity of modeling options for designing the model of the building. Therefore, the discovered parameters presented in Table 2.1 cannot be queried from the KNX-IoT 3$^{rd}$ party server. This server will later also be referred to as the gateway between the MAS and the KNX network. Nevertheless, it is analyzed if enough data is given to instantiate a simple rule-based or Proportional-Integral-Derivative (PID) control. For this purpose the BOPTEST test case *Multizone Residential Hydronic*[1] was chosen, as it provides multiple zones and radiator heating.

### 3.4.1 Modeling in the ETS

First, the structure of the building presented in Section 3.1 was modeled using the ETS software. Three spaces were created: the attic, the ground floor, and a garden. On the ground floor, each room is added with its intended purpose. KNX allows one to choose between a set of given purposes for each room. Some of the options are *garden*, *bathroom*, *bedroom*, *garage*, *hallway*, *living room* and others. It seems natural that these purposes are later used when creating the semantic export, i.e., the ontology used by the gateway. Finally, KNX can also model a *distribution box*, and as most actuators are installed in one, such an entity has also been created.

After roughly modeling the structure of the building, controllable radiator valves were added to each room, as this information is mandatory for the agents. Otherwise, they cannot determine whether a room can be heated. All valves are controlled by a central unit that is located inside the distribution box. Next, communication objects for each valve were created to specify a setpoint. Furthermore, communication objects used to exchange the current indoor temperature and an object for determining if someone is present inside of a room were created and assigned to corresponding devices. Devices for temperature and presence measurements were added and linked to rooms equipped with heating or cooling systems. Also, a sensor and a communication object were created and placed in the garden to measure the outdoor temperature.

The last available modeling option within the ETS software is to define arbitrary functions and assign them to the previously created rooms. Currently, there are five options when creating such a function: *switchable light*, *dimmable light*, *sun protection heating (switching variable)*, *heating (continuous variable)*, and *custom*. Each of those options will result in similar results, as the designer can freely choose the name, and apart from the icon, there are no differences visible in the ETS. After creating a function, communication objects can be assigned to them. One example of why these functions might be necessary is that they could help determine which rooms can be heated. In the next section, it will be analyzed if this is the case.

[1] https://ibpsa.github.io/project1-boptest/docs-testcases/multizone_residential_hydronic/index.html

### 3.4.2 KNX's Semantic Export

The rooms created beforehand in the ETS can also be found in the exported ontology. The Listing 2 shows an excerpt of the exported RDF code in turtle syntax. Here `prj:P-0C99-0_BP-4` is the subject of the RDF-triple, which is the living room in the presented case. In line one of the code, the defined name is given. Line two shows the implementation state, which can be set inside the ETS. It indicates if something is still under development, being tested, or finished. As it has not been used during modeling, it is set to *Undefined*. Followed by the state, the equipment located inside of this room is listed: The temperature and presence sensor. The different functions that were assigned to the room are also linked, as given in lines five and six. The presented values are the IDs of the heating and presence functions created earlier. As mentioned above, choosing between different purposes when creating a room is possible. This feature is also reflected here, as the *location usage* is set to be a *living room*. Finally, it is defined that the subject is of type `loc:Room`, with the *a*-keyword, which is shorthand for the property `rdf:type`. Note that the tags in front of the colon are used to reference other ontologies. Those have to be defined as prefixes to efficiently reference each foreign type unambiguously later. Finally, the last line, which states `owl:NamedIndividual`, is used to indicate that this subject is an individual and not a class in the ontology.

```
1 prj:P-0C99-0_BP-4 dct:title "Living Room";
2     core:state "Undefined";
3     loc:containsEquipment prj:P-0C99-0_DI-5,
4                           prj:P-0C99-0_DI-12;
5     loc:hasApplicationFunction prj:P-0C99-0_F-18,
6                                prj:P-0C99-0_F-19;
7     tag:hasLocationUsage tag:livingRoom;
8     a loc:Room,
9         owl:NamedIndividual.
```

Listing 2: The exported description of the living room.

Following the ID `prj:P-0C99-0_F-18`, the previously created function related to the heating control inside of the living room can be found. This function is of type *ApplicationFunction*, has the title *HeatingLivingroom*, and multiple *Function Points* assigned to it. These function points represent the communication objects that have been assigned during modeling. In line three of Listing 3, the object used to exchange the current indoor temperature is given, line four is the ID of the object used to set a new actuator state, and in line five, the object containing the current state is linked. Furthermore, there is a comment that this function was created in the ETS with the function type *FT-8*. Searching through the exported ontology, as well as the KNX Information Model [KNX23], no function type with this ID, name, or any other value can be found. This might be due to the current state of the development process of the KNX 3$^{rd}$ Party API.

The easiest way to determine if a room's indoor temperature can be controlled by an agent will be to verify if this room contains a function to heat or cool the room. As

```
prj:P-0C99-0_F-18 rdf:type owl:NamedIndividual ,
    core:ApplicationFunction ;
    knx:hasFunctionPoint prj:P-0C99-0_GA-2 ,
                         prj:P-0C99-0_GA-23 ,
                         prj:P-0C99-0_GA-44 ;
    dct:title "HeatingLivingroom" ;
    core:comment "KIM Application Function created from original ETS
    ↪  Function with Function Type: FT-8" ;
    core:state "Undefined" .
```

Listing 3: The exported application function responsible for the heating inside of the living room.

already seen, each room can be assigned any application function. The KNX Information Model [KNX23] defines all classes and properties introduced by the KNX 3rd Party API. For example, this model specifies what an application function actually is: A partial functionality of a control system. Furthermore, it has multiple function points assigned to it. In this case, the application function corresponds to one of the modeling options given in the ETS, namely the arbitrary functions that can be assigned to each room. In contrast to what one might expect, the five different types do not have any meaningful effect. Furthermore, the function points, also do not show any indication of what the purpose of this function might be. Such a result is to be expected because the designer does not specify any unambiguous functionality for these functions during modeling. Therefore, with the current version of the semantic export feature from the ETS, it is impossible to automatically create an ontology that provides enough information for the proposed system to operate successfully.

Further investigation of the exported model showed that the possibility to model the necessary information exists. A potential remedy is to use the subclasses of `core:ApplicationFunction` as these differentiate between the different purposes a function might have. The subclass *Air Temperature Control* could be used to indicate that a room can be heated with radiators. However, this is still fraught with ambiguity as zones could be equipped with different heating systems that require different controllers. Another possibility is to relate the function points of the application function to the heating system itself. In order to accomplish this, the *points* that the function point groups can be composed by a so-called *Functional Block*. The function block class has a greater variety of subclasses. Those include, for example, *Room Temperature Controller*, *Room Temperature Sensor* or *Outside Temperature Sensor*. When using classes like these, the same problem arises with the subclasses from the application function class. Furthermore, it is not possible to assign subclasses of the application function or the functional block during the modeling process.

To overcome this problem, the exported ontology is modified to query all of the rooms that are heated. To find the rooms in question, a special instance of a function block is created and named *FunctionBlock_Heating*. This block is assigned the ID of the function block *Room Temperature Controller*, which has the ID *fb.399* and is defined in the KNX

Information Model. Finally, the newly created function block is assigned to every data point responsible for controlling the radiators. Querying the ontology for this function block allows finding all rooms equipped with a radiator. Now, the living room has an application function, which itself has multiple function points. As it has been defined in the ETS, these also include one function point used to control the valve in the living room. This specific function point groups multiple data points, which can be used to set a control value for an actuator, for example. Such a data point is now assigned the new function block.

## 3.5 Extension of the Thing Description

Each agent will be a WoT instance, as this comes with multiple benefits: The communication between them is standardized but can be arbitrarily defined by letting a Thing provide different interaction methods. Furthermore, the TD is given in JSON-LD, allowing the previously modeled ontology to be included with its presented classes. Each agent's TD will be extended with the skills it advertises and discovers. As the TD is published into the TDD, other agents can discover the additional skills by reading these descriptions. Once an agent has found another agent that provides a useful skill, it looks for an appropriate function to call. It can do this by inspecting the return types of the properties, actions, and events the discovered Thing provides. Some methods also require inputs to a function, such as a zone ID. This poses another challenge, as these types need to be unambiguous.

Another important piece of information is which agents are implemented in the system. Therefore, Figure 3.11 shows the MAS role model of the implemented system. It also contains a green box labeled *BOPTEST*, which is not an agent or a role, but vital software component some of the agents use it as a source of information. For example, the weather prediction agent will query the weather data from this instance instead of retrieving the information from, e.g., an online weather API. Note that the agent with the role *Building Conditions* does not actively read the current state of the building from BOPTEST, as this information is returned after advancing the simulation. Hence, this data is stored locally at this agent.

The system consists of five different roles that agents must take:

- The *Building Conditions* role is used to provide the current state of the building equipment and apply new control values upon available updates.
- To determine these control values, the role named *Determine Control Values* is performed by an agent. Each zone in the building will have an agent with this role. Agents in this role are called temperature agents. These agents require additional information from the subsequent roles to compute the new outputs.
- To determine if the zone is occupied, an agent with the role *Occupancy* is used. It also allows an agent to predict whether a zone will be occupied in the future.

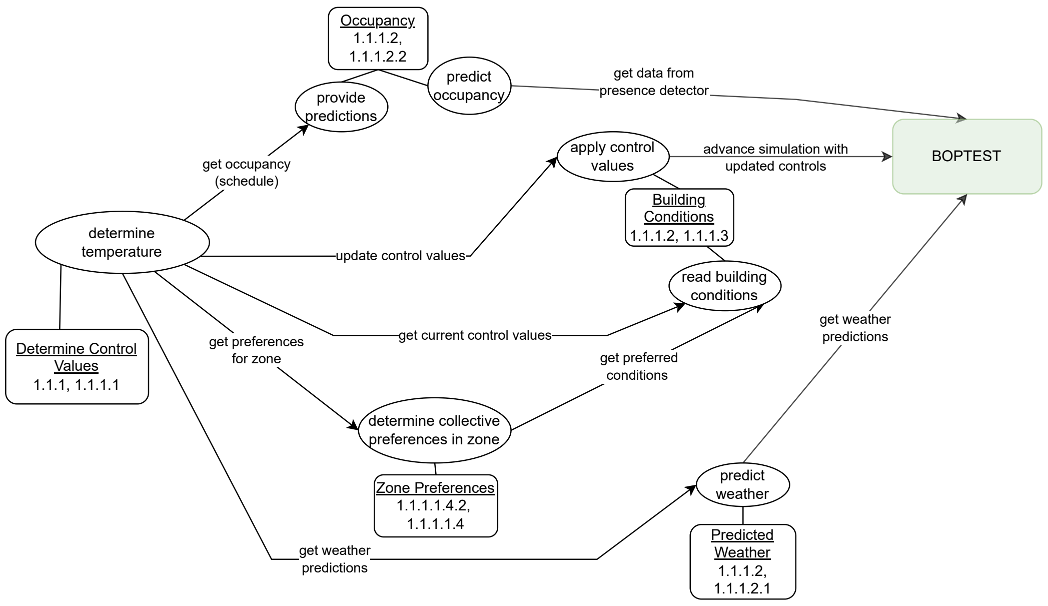


Figure 3.11: The MaSE role model of the implemented system.

- One temperature agent is responsible for controlling the boiler. Computing corresponding control signals require weather data and, therefore, discovers an agent with such a skill. The discovered agent has the role *Predict Weather* and is later referred to as the Weather Prediction Agent
- As the temperature agents need to know which is the optimal setpoint for a zone, another role called *Zone Preferences* is utilized for this purpose.

### 3.5.1 Enriching Things with skills

The first step to use the developed skills ontology is to include the context into the TD. This allows using the vocabulary defined in the ontology in the description of the Things. In Listing 4, the context of the agent responsible for controlling the indoor temperature inside of a zone can be seen. In line three of this listing, the created ontology has been included. The ontology is reachable on a local Apache Jena Fuseki server at port 8000. Other ontologies have also been included, namely Brick, QUDT, SAREF, Schema, and QualityKind. Furthermore, in line six *unit* is included which is defined in the QUDT ontology. It allows the use of a great variety of different units.

The Listing 4 contains additional properties called *advertises* and *discovers*. These are used to list the context-capability pairs that an agent advertises and discovers. For this purpose, the class *ContextCapabilityPair* is included as well in line 11. Finally, the properties *hasContext* and *hasCapabilities* are added to the context to later assign a context and a capability to a pair.

```
"@context": [
  "https://www.w3.org/2022/wot/td/v1.1",
  {"aso": "http://localhost:8000/agent_ontology.ttl#" },
  {"brick": "https://brickschema.org/schema/Brick#" },
  {"qudt": "http://qudt.org/schema/qudt/" },
  {"unit": "http://qudt.org/vocab/unit/"},
  {"saref": "https://w3id.org/saref#" },
  {"schema": "https://schema.org/" },
  {"qualityKind": "https://qudt.org/vocab/quantitykind/" },
  {"advertises": "http://localhost:8000/agent_ontology.ttl#advertises" },
  {"discovers": "http://localhost:8000/agent_ontology.ttl#discovers" },
  {"ContextCapabilityPair":
  ↪  "http://localhost:8000/agent_ontology.ttl#ContextCapabilityPair" },
  {"hasContext": "http://localhost:8000/agent_ontology.ttl#hasContext"},
  {"hasCapabilities":
  ↪  "http://localhost:8000/agent_ontology.ttl#hasCapability" }
]
```

Listing 4: The context field of the temperature agent TD.

The properties defined in previous sections are used to define the skills that the agent advertises and discovers. In Listing 5, the part of the TD of the temperature agent is given to show how an agent advertises different skills. In this case, the context-capability pair *SpaceHeating* and *ExternalSystemControl* is depicted. In the implemented system, there is only one agent that looks for such a pair: the agent responsible for applying control values. This agent is called *BuildingInterfaceAgent* and also looks for agents that provide the pair *AirConditioning* and *ExternalSystemControl*. The temperature agent also computes control values for the air conditioning and, therefore, advertises this pair as well. In Listing 5, it is not explicitly stated as it has the same structure as the other pair it advertises.

```
"advertises": [
    {
      "@type": "ContextCapabilityPair",
      "hasContext": {
        "@type": "aso:SpaceHeating"
      },
      "hasCapabilities": {
        "@type": "aso:ExternalSystemControl"
      }
    },
```

Listing 5: Excerpt of a TD to show how an agent advertises skills.

A TD of an agent also contains the skills it seeks in the same fashion as the skills it advertises. Agents do not actively search for other agents that discover a particular pair, but initiating a conversation with an agent that advertises a matching skill is possible. Nevertheless, to make the matching between the skills easier, the skills an agent discovers have also been added to the TD. The temperature agent discovers a majority of the skills used inside of the system. These include the following:

- $\langle \text{Weather}, \text{Predict} \rangle$: As the agent shall use weather predictions when computing its control values, this pair is discovered by the temperature agent.
- $\langle \text{Occupancy}, \text{Predict} \rangle$: To reduce energy consumption due to heating and cooling, the agent tries to save energy during unoccupied periods. For this reason, it searches for an agent that provides predictions about these.
- $\langle \text{Setpoint}, \text{Negotiation} \rangle$: This skill is discovered as the agent is looking for a setpoint that computes a setpoint that inflicts the least setpoint violation for all users.
- $\langle \text{BuildingControl}, \text{ConditionMonitoring} \rangle$: This skill is given by the building interface and describes the means of monitoring the control variables applied to the system. An agent that advertises these is capable of reading the current conditions of installed building equipment.

### 3.5.2 Return types for Thing-Functions

Once Agents discover another skill they would like to use, they start exchanging messages, but the returned type could significantly vary from one implementation to another. Therefore, it is necessary to define types that are equivocal under every scenario. Assume a simple MPC controller uses a lumped thermal capacitance model to compute the control values. The agent capable of providing information about the thermal capacitance needs to return something of this type, i.e., the returned type given in the TD, shall be defined correctly. These types can be defined unambiguously by specifying the returned type with an ontology.

As ontologies should be reused, it was prioritized to use existing types in well-known ontologies like SAREF or QUDT. In the QUDT ontology, the quality kind called *ThermalCapacitnace* was found. A quality kind is defined as follows [FAI15]:

> *A Quantity Kind is any observable property that can be measured and quantified numerically. Familiar examples include physical properties such as length, mass, time, force, energy, power, electric charge, etc. Less familiar examples include currency, interest rate, price-to-earning ratio, and information capacity.*

So a quality kind is the measurement of something. The quality kind also has a unit, which will be in $J/K$, if a thermal capacitance is returned. The Thing will define its return type as follows:

Note that the prefix `qualityKind` is defined in the context, ensuring it is strongly typed. Also property *qudt:unit* defined by the QUDT ontology has been set to *unit:J-PER-K*. This defines the returned type without any ambiguity.

```
"output": {
    "@type": "qualityKind:ThermalCapacitance",
    "type": "number",
    "qudt:unit": "unit:J-PER-K",
    "description": "The thermal capacitance of the zone."
  }
```

Listing 6: The returned type when querying the thermal capacitance.

This solution seems to fix the problem and increase interoperability. Nevertheless, assume that the controller given in the provided examples is replaced by another controller that no longer uses a lumped model. Instead, it considers different thermal capacitances for each zone. Now, different thermal capacitances are required, e.g., the one from the internal mass, the exterior and interior walls separately, and finally, one for the roof and the floor of each zone. If the same agent is queried for the different thermal capacitances, the MPC-agent has no chance of distinguishing between the different components if no additional information is given. Therefore, an extra input field along with the already given zone ID is required to distinguish between the components. The REC ontology defines a type named *Wall*, which is not specific enough. Other building elements found are *Slab* and *Roof*. To describe the missing types, the REC ontology can be extended by the classes provided in Listing 7. This allows for modeling all the different components of a building.

```
@prefix rec: <https://example.org/rec#> .
@prefix aso: <http://localhost:8000/agent_ontology.ttl#>

aso:ExternalWall a owl:Class ;
    rdfs:subClassOf rec:Wall ;
    rdfs:label "External Wall" .

aso:InternalWall a owl:Class ;
    rdfs:subClassOf rec:Wall ;
    rdfs:label "Internal Wall" .

aso:InternalMass a owl:Class ;
    rdfs:subClassOf rec:BuildingElement ;
    rdfs:label "Internal Mass" .

aso:AirInsideZone a owl:Class ;
    rdfs:subClassOf rec:BuildingElement ;
    rdfs:label "Air Inside Zone" .
```

Listing 7: The extension of the REC ontology.

If the MPC controller now requests a thermal capacitance, it may specify what building element it is looking for, and the agent returning the parameter knows which value shall be returned. Adding additional inputs also helps to use more complex thermal models of the building. For example, with R6C2, multiple resistances and capacitances are given for a single wall. Adding an input parameter defining the position inside the wall will also allow the use of more sophisticated models.

The implemented system will compute the control values based on a closed-loop PID controller, which does not include a thermal building model. Nevertheless, not all of the types that can be found inside additional ontologies make it possible to define meaningful return types. Therefore, the existing ontology is extended with the necessary types.

```json
"setValvePosition": {
  "title": "Set Valve Position for a zone",
  "description": "Set the valve position for a given zone to control
  ↪  heating.",
  "input": {
    "type": "object",
    "properties": {
      "zoneId": {
        "type": "string",
        "description": "The ID of the zone to set the valve position
        ↪  for.",
        "@type": "aso:ZoneId"
      },
      "position": {
        "type": "number",
        "description": "The position of the valve.",
        "qudt:unit": "unit:PERCENT",
        "@type": "aso:RelativeValvePosition"
      }
    },
    "required": ["zoneId", "position"]
  },
  "output": {
    "properties": {
      "result": {
        "type": "boolean",
        "description": "True if the valve position was set successfully,
        ↪  False otherwise."
      }
    }
  },
  "forms": [
    {
      "href": "http://localhost:PORT/buildinginterfacemicroservice/
      ↪  actions/setValvePosition",
      "contentType": "application/json",
      "op": "invokeaction",
      "method": "POST"
    }
  ]
}
```

Listing 8: The method used to update the valve position of a given zone.

To illustrate how such a function is used, inside a TD, the Listing 8 is presented. This method is given in the TD of the building interface and is used to send an updated valve position calculated by the temperature agent. In line 4, the input of this function is defined. It requires a *zoneId* and a *position* to be specified. The property *type* and *@type* do not contradict each other; they have different meanings. When defining the property `type`, it indicates that the input has to be given as a character string. On the other hand, `@type` is used to enrich this input with semantics. By specifying the value *aso:ZoneId*, it is clarified that the given type is defined by the class *ZoneId* in the ontology with namespace *aso*. The second input to the function is the position of the valve. The value is passed as a percentage and of type *aso:RelativeValvePosition*. Line 19 of the listing specifies that these inputs are mandatory.

The function's output is a boolean to indicate if the operation has been successful. Finally, the `forms` property in line 29 is used to specify how this method can be invoked. A URL, the required HTTP method, and the content type format are specified.

## 3.6 Proof of Concept - Multi-Agent System Implementation

The parameters relevant to different control strategies have already been summarized in Section 2.5. To implement this MAS, a PID controller, combined with rules, is used. The reason for this is that some relevant information is not given in the BOPTEST test case, and for proving the concept of a MAS operating on the KNX ontology, it suffices to implement a simple controller. Table 2.1, which presents the parameters, does not specify which are essential for this implementation's operation. Even for more complex controllers, not all of them might be given or are irrelevant for optimal control. Depending on the controller implementation, the used variables differ. The implemented system requires the following parameters:

- The outdoor dry-bulb temperature ($T_{out}$) is used to determine the setpoint of the boiler.
- The presence of occupants ($O_z$) within the zone will also impact the system's functionality. If the zone is predicted to be unoccupied for a longer period, the system will reduce the heating output.
- A preferred setpoint ($T_{set}$) must be queried to control for minimal discomfort.
- Nominal temperatures of the supply water ($T_{sup,n}$) and the return water ($T_{ret,n}$) are relevant to compute the supply water temperature for the radiators.
- An offset temperature ($T_o$) is also required, as it allows influencing the valve position. Given the variability in radiator sizes and the unpredictability of the supply temperature due to potential series connections of radiators, the required temperature control deviates for each radiator. This parameter cannot be known in

advance as it depends on many other variables. An agent responsible for estimating the building parameters could be used to determine this value. However, in this implementation, the value is determined manually.

The control values the MAS computes are the following:

- The boiler load ($U_{boi}$) to supply enough heat to the radiators.
- One agent also controls the supply water temperature ($T_{sup}$). This temperature is not directly proportional to the boiler load, as the supply water temperature is the temperature supplied to the radiators, so the temperature after the mixing valve. No agent determines the control value of the mixing valve. Instead, this variable is used to simplify the calculations. BOPTEST is capable of computing the desired mixing valve value itself.
- The radiator valve ($u_{valve}$) in each zone. This value is given as a percentage.

The implemented agents have already been shown in Figure 3.11. Nevertheless, the picture does not clarify how many temperature agents are used. In fact, there is one temperature agent for each zone to reduce the computational complexity when there are multiple zones. An agent can be in charge of controlling the boiler, if it is instantiated with the corresponding flag. When a temperature agent is run, some parameters are passed to the program. These consist of the following: the location of the TDD, which port should be used, the location of the TD-file, and a flag used to indicate if the agent is responsible for controlling the boiler. Therefore, the system consists of five temperature agents. The agent responsible for the living room will also control the boiler.

Before all agents are instantiated, the exported ontology is queried to determine all heated zones. This allows the system to create the necessary agents automatically. In Listing 9, the query issued to the KNX ontology is shown. It follows the same path described earlier in Section 3.4.2. This query returns all rooms that have a controllable valve. Of course, this is only possible because the exported ontology is modified so that the corresponding data point *composes* the function block with type 399. Once the rooms are known, the temperature agents can be started.

The first task an agent fulfills is to start the WoT server and publish its TD to the TDD. For every temperature agent, the same TD file is used. Therefore, it modifies this file before publishing it to the TDD. One example of such a modification is that the agent specifies the port in each URL.

After the initialization steps are done, the agent starts two tasks. One task repeatedly looks for updates in the TDD to find new agents and detect updates to existing agents. Once an agent's TD containing a matching context-capability pair has been found, the agent discovering agent can include information provided by the new agent in its second task. In this second task, it carries out the main duties it is responsible for, e.g., computing control values.

```
SELECT DISTINCT ?id ?title ?locationUsage ?state
WHERE {
    ?id rdf:type loc:Room ;
    dct:title ?title ;
    tag:hasLocationUsage ?locationUsage .
    ?id loc:hasApplicationFunction ?appFunction .
    ?appFunction knx:hasFunctionPoint ?functionPoint .
    ?functionPoint core:groups ?datapoint .
    ?datapoint knx:composes ?functionBlock .
    ?functionBlock rdf:type knx:fb.399 .
}
```

Listing 9: The method used to update the valve position for a given zone.

### 3.6.1 Zone Preferences Agent

The building interface agent advertises the context-capability pair *Setpoint - Condition Monitoring*. The zone preferences agent discovers this skill, as it intends to keep track of the different preferred setpoints, each inhabitant has and reconciles these to a designated setpoint for each zone. However, BOPTEST cannot distinguish between individual persons in each room. Therefore, the zone preferences agent will simply use the setpoint it receives from the building interface to compute the required setpoint range.

This temperature range defines a comfort span in which the inhabitants will not feel discomfort. The heating setpoint is the lower bound, which is acceptable by the persons inside, while the cooling setpoint is the maximum temperature before feeling uncomfortable. These setpoints define the target temperature for the heating and cooling systems. To determine this range, the zone preferences agent first queries the building interface agent for the user-defined setpoint using the defined method in the TD. In Listing 10, the output of the function given by the building interface agent is depicted. It states that the returned value is a temperature setpoint and given as a number in °C.

```
"output": {
    "type": "object",
    "properties": {
        "setpoint": {
            "@type": "brick:Temperature_Setpoint",
            "type": "number",
            "qudt:unit": "unit:DEG_C",
            "description": "The setpoint for the whole building."
        }
    }
}
```

Listing 10: The output of the *getSetpoint*-method the building interface agent provides.

As depicted in Listing 11, the zone preferences agent uses this setpoint to compute the setpoint range. It requires the input *dateTime*, which is the current time in seconds from the start of the year. This is necessary during simulation as this agent is not in direct contact with BOPTEST, and it has no means of determining the current day and time of the day. Therefore, this parameter is passed to this method. In line nine of Listing 11, the cooling setpoint is increased by two Kelvin during winter, and in line eleven, the heating setpoint is reduced if it is summer. In lines 16-18, the agent checks whether the time is between 10:00 and 19:00, and if the day is a working day. If so, the lower bound of the setpoint range is set to 16°C and the upper bound to 26°C. This is because the simulated building is considered to be unoccupied during this time. Since the benchmark controller reduces its power consumption by following these setpoints, the implemented MAS implements the same approach to allow better comparison.

```
1   thing.setActionHandler("getZonePreferences", async (params) => {
2     const { dateTime } = await params.value();
3     console.info(`Received 'getZonePreferences' request`);
4
5     let cooling_setpoint = setpoint;
6     let heating_setpoint = setpoint;
7
8     // Increase the cooling setpoint during winter and decrease the heating
      ↪  setpoint during summer
9     if (dateTime <= 24*60*60 * 90 || dateTime >= 24*60*60 * 270) {
10      cooling_setpoint += 2;
11    } else if (dateTime >= 24*60*60 * 120 || dateTime <= 24*60*60 * 240) {
12      heating_setpoint -= 2;
13    }
14
15    // Reduce cooling and heating during the unoccupied period
16    if(dateTime % (24*60*60) >= 10*60*60 &&
17       dateTime % (24*60*60) <= 19*60*60 &&
18       dateTime % (24*60*60*7) <= 24*60*60*5) {
19      heating_setpoint = 16;
20      cooling_setpoint = 26;
21    }
22
23    return { "heatingSetpoint": heating_setpoint, "coolingSetpoint":
      ↪  cooling_setpoint };
24  });
```

Listing 11: The action handler of the zone preferences agent, that computes the setpoint range.

The zone preferences agent itself provides a method called *getZonePreferences*, which the temperature agent uses to fetch the setpoint range. The output of this method is given in Listing 12. The returned object consists of two properties: the heating setpoint and the cooling setpoint. Both are given in °C and of the type `brick:Heating_Temperature_Setpoint` and `brick:Cooling_Temperature_Setpoint`, which are defined in the Brick ontology.

```
"output": {
    "type": "object",
    "properties": {
        "heatingSetpoint": {
            "type": "number",
            "qudt:unit": "unit:DEG_C",
            "description": "The heating setpoint of the zone.",
            "@type": "brick:Heating_Temperature_Setpoint"
        },
        "coolingSetpoint": {
            "type": "number",
            "qudt:unit": "unit:DEG_C",
            "description": "The cooling setpoint of the zone.",
            "@type": "brick:Cooling_Temperature_Setpoint"
        }
    }
}
```

Listing 12: The output property of the *getZonePreferences*-method

### 3.6.2 Temperature Agent

As Figure 3.11 depicts, the agents controlling the temperature in each zone communicate with the occupancy agent, the weather prediction agent, the building interface, and the zone preferences agent. Therefore, it searches for the following context-capability pairs:

1. Weather - Predict
2. Occupancy - Predict
3. Setpoint - Negotiation
4. Setpoint - Condition Monitoring
5. Building Control - External System Control

The first three pairs are advertised by the weather prediction agent, the occupancy prediction agent, and the zone preferences agent, respectively. The fourth pair is advertised by the building interface. It indicates that the agent can read the current setpoints inside the building. The final pair is also advertised by the building interface. This pair is used to inform other agents that this agent can apply new control values to the building.

As the task of discovering agents runs in parallel with the primary task responsible for computing new setpoints, the temperature agent will wait until the building interface agent has been discovered. Otherwise, it is not possible to read the current temperature or apply new control values. As soon as the agent discovers the interface, it starts querying the agents for which it has already found a TD.

To compute the control values, a hysteresis loop is first used to determine if the zone requires heating or cooling. If the measured zone temperature is less than the heating setpoint plus one Kelvin, the agent is in heating mode and will compute a valve position. If, on the other hand, the measured zone temperature is above the cooling setpoint minus one Kelvin, it will compute control values for the AC, as it will go into cooling mode. The two different operations are now explained in more detail:

1. An agent in heating mode will compute the valve position based on Equation 2.11. Two parameters must be determined for successful operation: $\alpha_v$ and $T_o$. The first parameter, $\alpha_v$, is used to determine the slope of the sigmoid function, while the second, $T_o$, is used to shift the function on the x-axis. Different configurations can be seen in Figure 3.12. It clearly depicts how the offset $T_o$ shifts the whole function on the x-axis. Furthermore, the function's slope is increased if $\alpha_v$ is also increased. Increasing this value will reduce the band where the valve is not fully opened or closed. For the implementation, $\alpha_v$ has been chosen to be ten while $T_o$ is 0.5 for all rooms except for the living room, which requires $T_o = 0.7$ for better operation.
2. For computing the AC control values, the agent simply uses a PID control strategy to determine the partial load of the air-cooling system. The PID constants have been determined manually but are the same for each zone.

Note that each agent receives the desired setpoint from the zone preferences agent. It will then increase or decrease the setpoint by an additional 0.5 Kelvin, depending on whether the agent is in heating or cooling mode. This allows the discomfort to be reduced further, as slight variations do not fall below the heating setpoint or exceed the cooling setpoint.

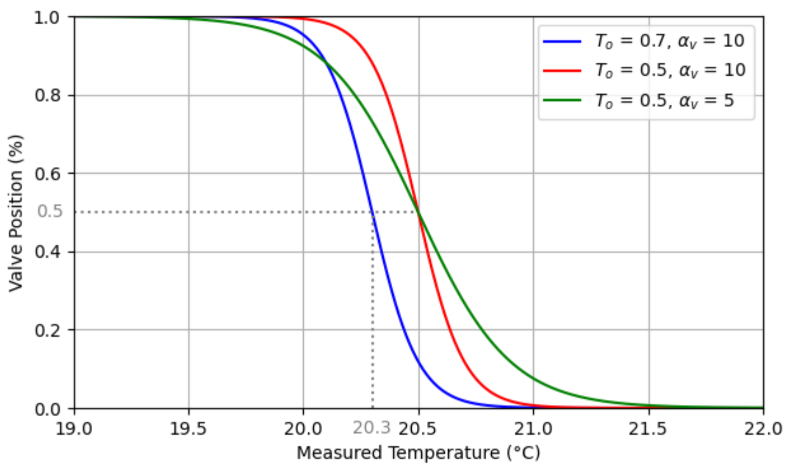


Figure 3.12: The sigmoid function used to control the valve position.

Another important detail is how the system determines the controls for the boiler. As noted before, agents are initialized with a flag indicating if it is responsible for controlling it. The simulated building requires a supply water temperature as input, as well as a boiler load. For computing the supply water temperature, Equation 2.9 is used. The formula requires a nominal value of the supply water temperature and a nominal return water temperature. As these values are not given in BOPTEST, they are estimated to be 62°C and 55°C, respectively. Due to missing necessary data, an alternative formula, as shown in Equation 2.10, has been selected. In [PDKH17] $Q_{rel}$ is expressed as the following:

$$Q_{rel} = \frac{T_{set} - (T_{avg\_out} + \epsilon)}{T_z - (T_{nom\_out} + \epsilon)} \tag{3.1}$$

$T_{set}$, $T_{avg\_out}$, $\epsilon$, $T_z$ and $T_{nom\_out}$ are the setpoint, the average outdoor temperature, a correction term and the nominal outdoor temperature, respectively. This equation shows that the outdoor temperature is necessary to compute the supply water temperature. This is the sole reason why the weather prediction agent is required. Other agents do use predictions about the weather in their computations. The final step is to compute the boiler load, which is done with another PID controller.

Once each agent has computed its control values, it will send the new values to the building interface, where the data is applied to the BOPTEST simulation. As the heating emission system also contains a mixing valve and a pump, but the agent responsible for the control of the boiler will only compute a supply water temperature and the boiler load, the simulation automatically determines the value of the mixing valve. The emission pump is not controlled and is always turned on.

### 3.6.3 Building Interface Agent

This agent has two main tasks: It receives new control inputs and provides an interface for other agents to determine the current settings in the building. For this purpose, it advertises the context-capability pairs *Setpoint - Condition Monitoring*, *Occupancy - Condition Monitoring*, *Setpoint - Condition Monitoring* and *Building Control - External System Control*. The zone preferences agent discovers the pair *Setpoint - Condition Monitoring*, while the occupancy-prediction agent discovers the *Occupancy - Condition Monitoring* pair. In the other main task, this agent advances the simulation once it received new inputs from all agents. In an actual building, this would not be necessary, and updates would be applied instantly.

The agent is also instantiated with a port number, the location of the TD, and the address of the TDD. Also, this agent will start the WoT server first and then publish its TD to the TDD. For starting an agent, the framework *node-wot*[2] is used as it allows to quickly start a Thing and implement its methods, properties, and events.

In Listing 13, the procedure of exposing a Thing is depicted. In line seven of the listing, the Servient-class of node-wot is used to start a new WoT server. The corresponding

[2]https://github.com/eclipse-thingweb/node-wot

TD is passed in line eight. In the following lines, the handlers are implemented for each method specified in the TD. In this listing, only the action to set the boiler load is depicted. In line 16, the input is validated before it is stored in a data structure that holds the new control values in line 20. At the end of the listing, in line 27, theThing is finally exposed. Upon this point, other devices can communicate with this agent, but to find this agent, it has to publish the TD now.

```
1  const Servient = require('@node-wot/core').Servient;
2  const HttpServer = require('@node-wot/binding-http').HttpServer;
3
4  const servient = new Servient();
5  servient.addServer(new HttpServer({ port: port }));
6
7  servient.start().then((WoT) => {
8    WoT.produce(thingDescription).then((thing) => {
9      console.log('Produced Thing:', thing.getThingDescription().title);
10     thingInstance = thing;
11
12     // Action handler for 'setBoilerLoad"
13     thing.setActionHandler("setBoilerLoad", async (params) => {
14       const { percent } = await params.value();
15       console.log(`Received 'setBoilerLoad' request for load:
         ↪  ${percent}`);
16       if (percent < 0 || percent > 100) {
17         return { result: false };
18       }
19       // Update control value in data structure
20       controlValues.global.boi_oveBoi_u = percent / 100;
21       return { result: true };
22     });
23
24     // Other methods follow
25
26     // Expose the Thing
27     thing.expose().then(() => {
28       console.log(`${thing.getThingDescription().title} exposed`);
29     });
30   });
31 });
```

Listing 13: The exported description of the living room.

## 3.6.4 Other agents

Finally, the weather and occupancy prediction agent have a similar operation. In the current implementation, they simply query BOPTEST for the required data once they receive a request. They make use of the `forecast`-endpoint BOPTEST provides to retrieve this data. The weather prediction agent uses a time horizon of six hours and an interval of one hour. Therefore, the agent returns an array with seven data points, the current outdoor temperature, and the hourly predicted weather for the next six hours.

The occupancy agent only uses a three-hour horizon but a 15-minute interval between each prediction. For actual buildings, the weather prediction agent most likely queries a different weather API, which is publicly available. Also, the occupancy agent can no longer request data from BOPTEST. Instead, it might use learning algorithms to predict the inhabitants' behavior.

CHAPTER 4

# Evaluation Results

The evaluation of the proposed system is presented in this chapter. The MAS operates on the simulated building described in Section 3.1. First, the performance of the controller is measured by comparing it to a benchmark controller. The overall discomfort and the energy consumption are highlighted. Additionally, the modularity of the implemented system is tested by adding agents during the runtime of the simulation to see how the controllers will react. Moreover, the scalability with multiple zones will be examined.

## 4.1 Controller Performance

An evaluation of the performance is necessary as the implemented system should be comparable to other controllers, even though it is not the primary objective of the controller to reach the highest possible user comfort with as little energy as possible. The objective of the implemented system is to be adaptable, which is also discussed in later sections. Nevertheless, a fair comparison is only possible if the controller's performance is evaluated.

The controller is implemented for the BOPTEST test case named *multizone residential hydronic*. As noted in the previous chapter, the building consists of five conditioned zones: a living room, a bathroom, and three bedrooms. One additional zone, the hallway, is equipped with a radiator that cannot be controlled. The simulated period has been set to seven days, and the time step at which new inputs are applied is set to one minute. In Figure 4.1, the resulting indoor temperature of the living room (*Liv*) and the first bedroom (*Ro1*) is depicted. Other rooms (*Bth*, *Ro2*, and *Ro3*) are not shown, as the results are quite similar. In the first and second subplots of Figure 4.1, the blue line indicates the temperature inside each zone, while the two grey lines in each plot mark the preferred setpoint range. Additionally, the outdoor temperature and the solar irradiance are depicted in the last subplot. These factors have noticeable influences on the indoor temperature, which is why they are also depicted. Hence, the indoor temperature does not need to follow one setpoint but instead must be within the range given by the heating and cooling setpoint. The heating setpoint is the lower bound at which an inhabitant feels comfortable, while the cooling setpoint defines the upper bound. This range varies due to different reasons. If the zone is unoccupied, the heating and cooling setpoint is reduced and increased, respectively. More explanations about the setpoint ranges will be discussed shortly. The simulation takes place in France during a winter week and is executed for one week. As can be seen from the figure, the controller tries to minimize energy consumption by closely following the heating setpoint.

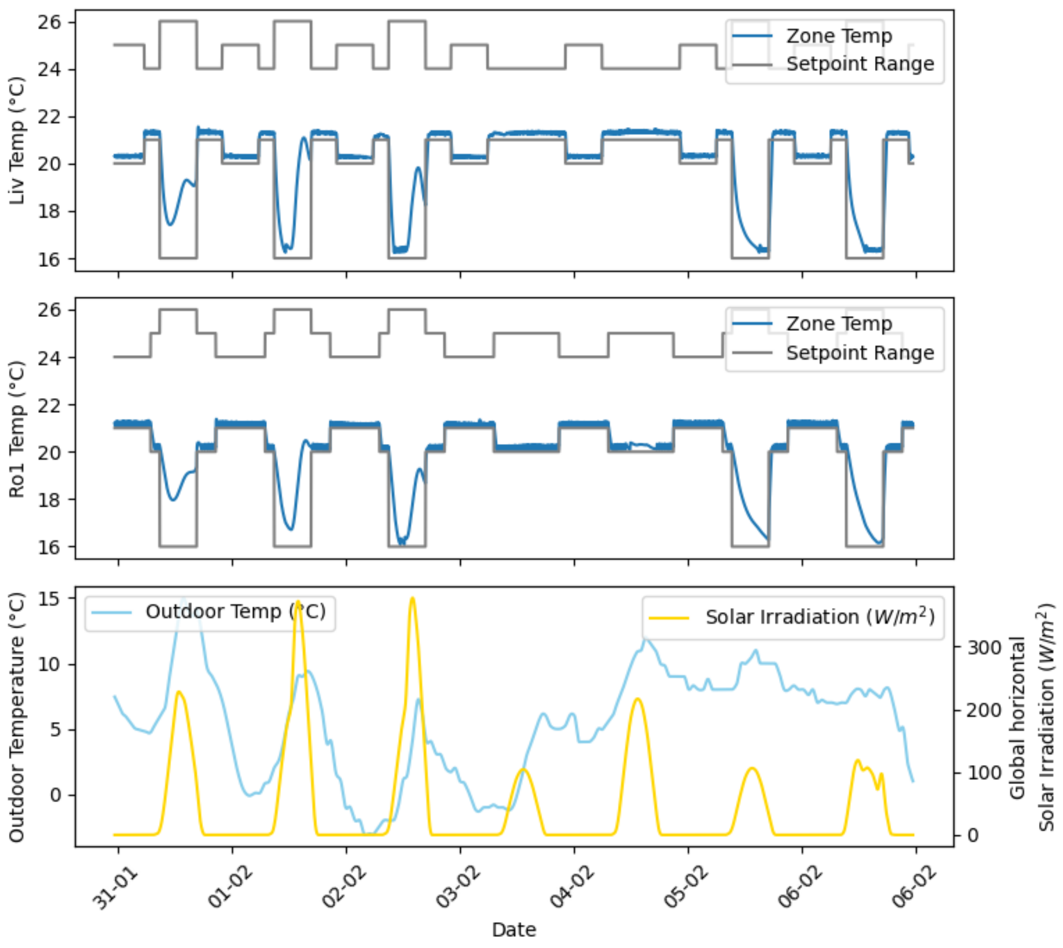


Figure 4.1: The indoor temperature after simulating with all agents.

The setpoint range has great variations, which require additional explanation. Figure 4.2 depicts one day of simulation and highlights the different setpoint ranges. Also, note that the ranges are quite similar for each zone. The different ranges can be classified by three different rules:

1. By definition of the BOPTEST test case, the building is unoccupied on five out of seven days. During this time, the benchmark controller will drastically reduce energy consumption by lowering the heating setpoint. Therefore, the implemented MAS also adheres to this schedule to allow a fair comparison. The whole building is considered unoccupied between 9 a.m. and 5 p.m. on five weekdays. The setpoint range for each zone is adjusted to 16°C and 26°C. This period is highlighted in blue in Figure 4.2.

2. If inhabitants are considered to be inside the building but are not present in the respective zone, the power consumption is slightly reduced. In Figure 4.2, the residents are at home between 00:00 and 06:00 and between 22:30 and 23:59 but are not present in the displayed zone, i.e., the living room. The respective period is marked in yellow. To reduce energy consumption, the heating setpoint is reduced by one Kelvin, while the cooling setpoint is increased by one.

3. If someone is detected to be inside a zone, the controller uses the preferred setpoint range, which is from 21°C to 24°C. The respective periods are colored in red in Figure 4.2.

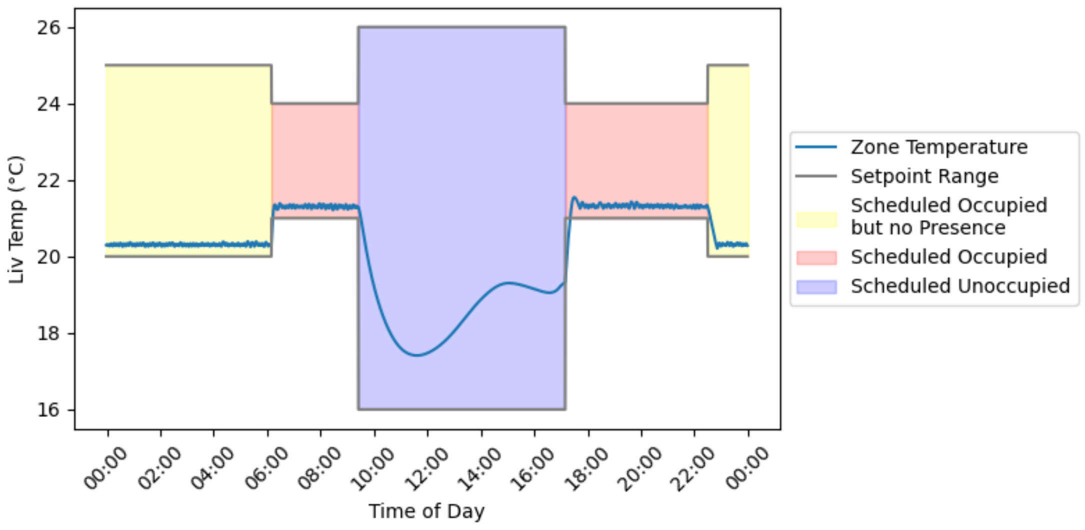


Figure 4.2: The different setpoint ranges during the simulation.

Going back to the performance of the MAS, Figure 4.1 does not point out any apparent violation of the preferred setpoint range. Therefore, the actual deviation from the setpoint range is computed. BOPTEST also has the capability to provide key performance indicators. Nevertheless, these are not utilized here as they only provide a collective value for the whole building and not the individual zones. To calculate the total discomfort, the same formula used by BOPTEST has been applied to each zonal air temperature trajectory. To compute the discomfort for a given zone $z$ in the time interval $[t_0, t_{end}]$, Equation 4.1 was used.

$$disc_z(t_0, t_{end}) = \int_{t_0}^{t_{end}} |e_z(t)| dt \tag{4.1}$$

This equation computes the deviation from the setpoint range, denoted by $e_z(t)$. The resulting value has the unit $K \cdot h$ (Kelvin-hours). This value is calculated for each zone and the whole simulated period. The results are summarized in the second column of Table 4.1. The presented values are the accumulated setpoint deviations during the whole week in comparison to the benchmark controller. For more expressive values, the discomfort per hour can be calculated by dividing the given discomfort by the total number of hours (168 hours). For example, the living room has only an average deviation of about 0.03 Kelvin. The benchmark controller will be discussed in the following section.

| Room | MAS Discomfort (Kh) | Benchmark Discomfort (Kh) |
|---|---:|---:|
| Liv | 4.35 | 211.00 |
| Bth | 1.46 | 29.73 |
| Ro1 | 1.78 | 38.10 |
| Ro2 | 1.69 | 44.07 |
| Ro3 | 2.70 | 69.50 |
| Whole building | 11.98 | 392.41 |

Table 4.1: The discomfort for each zone incurred by the MAS and the benchmark controller.

To compare the results to the benchmark controller of BOPTEST, the same duration, date, and time step have been set to match the MAS. For better comparison, the same setpoint ranges have been applied during the execution. The results can be seen in Figure 4.3. The benchmark implementation controls the setpoint of each zone with a Proportional-Integral (PI) controller. An additional hysteresis function based on the temperature inside the living room controls whether the boiler is on or off. When the boiler is turned on, a PI controller is used to determine the partial load ratio.

There are many differences when comparing the two results. While the MAS implementation follows the lower setpoint quite well, the benchmark controller has trouble accurately controlling the indoor temperature. The most affected room is the living area. The controller does not maintain the provided lower setpoint and fails to meet the user's requirements. A reason for this huge deviation might be an erroneous design of the controller. Also, other rooms have some deviations, especially on the 4th day of February. The reason for this deviation is that the boiler is turned off for some time. One explanation for this is that the living room exceeds the incorrectly set desired indoor temperature due to higher outdoor temperatures and solar gains. When the living room

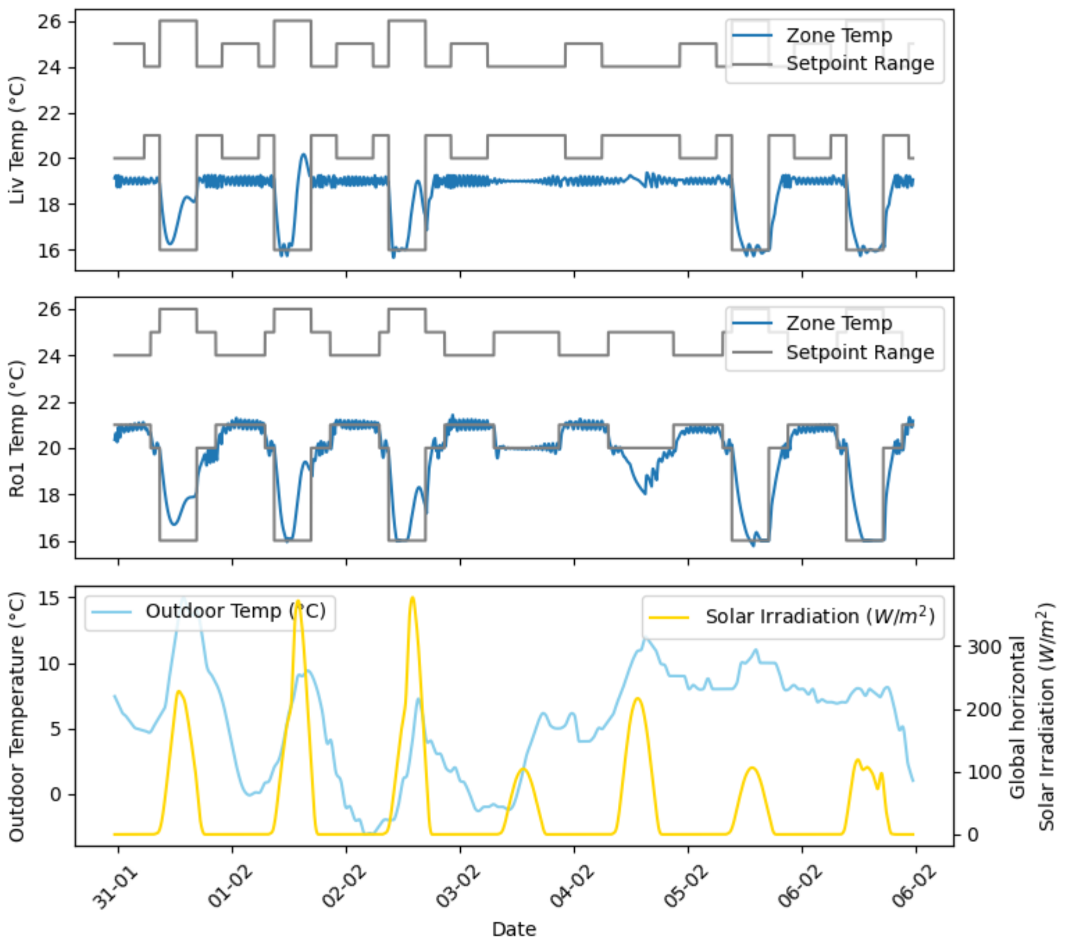


Figure 4.3: The indoor temperature achieved by the benchmark controller.

reaches its desired temperature, the boiler might shut off, as this room's air temperature is used to determine the boiler load.

The computed thermal discomfort that is given on the left side in Table 4.1 also represents these findings. Even if the living room is excluded from the comparison, the benchmark controller reaches a significantly higher discomfort. This is because the benchmark controller is very slow, and the indoor temperature increases only gradually after an unoccupied period. This effect can be observed in Figure 4.3 on the 5th day of February in the zone *Ro1*.

In addition to comparing the indoor temperature, assessing controller performance also requires considering power consumption. As the building heats the water for the emission system with a gas boiler, gas is also the primary consumed resource during this scenario. After finishing the simulation, the energy consumption of each room can be retrieved from BOPTEST directly. The results from the MAS and the benchmark can be seen in

Table 4.2.

| Room | MAS Gas Consumption (kWh) | Benchmark Gas Consumption (kWh) |
|---|---|---|
| Liv | 321.50 | 229.07 |
| Bth | 15.20 | 40.89 |
| Ro1 | 78.07 | 109.60 |
| Ro2 | 83.56 | 103.43 |
| Ro3 | 115.68 | 132.26 |
| Whole building | 614.01 | 615.25 |

Table 4.2: The gas consumption of the MAS and the benchmark controller.

The benchmark consumes only slightly more energy in total. As the living room is heated to a lower temperature, a closer inspection of the consumed energy is necessary. As expected, the living area requires a lot less heat to maintain the faulty heating setpoint. Other rooms do not consume less heat, which might also be the case for the living room if the temperature had been controlled correctly. When only comparing the other zones, the MAS consumed 93.67 kWh less than the benchmark.

## 4.2 Extendability of the Multi-Agent System

Another aspect that has to be considered is whether the extensibility of the developed framework is given and how much interaction from a system engineer is required to fully include new agents into the system. Therefore, different scenarios that might occur during the lifetime of a building are considered next. The scenarios are the following:

- Scenario 1: The building is fitted with an additional solar thermal system to reduce gas consumption by utilizing solar energy.
- Scenario 2: The structure of the building changes, i.e., an additional zone is created.
- Scenario 3: Adding a new agent to the system that provides additional data for better control, e.g., including an agent that is able to predict energy prices. Adding an agent has a similar effort as exchanging one of them, like replacing the rule-based controllers with multiple MPC agents.

Note that the first two scenarios inflict a change in the context of the building, which necessitates a modification of the KNX model. Agents inside the system can only detect a context change if the KNX 3$^{rd}$ Party API includes these updates as well. The last scenario describes the incorporation of a new skill. In the following sections, these scenarios are further evaluated.

### 4.2.1 Scenario 1: Integration of new equipment

Suppose the building is retrofitted with additional equipment. In that case, the existing agents themselves will not be able to detect these changes, as the building interface gathers its data from the KNX 3rd Party server, which does not include such changes until the new devices have been added to the ETS project. Until then, any newly introduced devices and their corresponding agents cannot be used. This is not the case if the newly introduced device is controllable via the Thing itself, i.e., it is controllable via another interface. Once the new device has been added, the corresponding agent can also be instantiated with a port number and the URL of the TDD. Then, the new agent can start interacting with other agents and begin its computation.

As BOPTEST cannot be extended without great effort, another approach has been chosen to show how the system would behave if new equipment is installed: The system is simulated during a summer week, once without cooling the rooms and once with applying inputs to the AC. The steps required to include the new device, namely the AC, are presented in the following.

First, the equipment has to be modeled in the ETS. A corresponding device has to be added, and a function dedicated to cooling is created for each zone. The communication objects are linked with these devices, and the model can be exported. The next step is to shut down the KNX 3rd Party API and include the new export. Once the gateway is up and running, the temperature-controlling agents can apply their computed controls by sending them to the building interface. In this case, the agents are already capable of computing control values for an air conditioning unit. Therefore, no further changes to the system are necessary. When installing the new equipment, the discomfort inside of the living room is reduced from 215.95 Kh to 1.06 Kh by only utilizing 36.36 kWh of electrical energy.

In Figure 4.4, the effects of including the AC are depicted in a different way. Here, the simulation is only run for three days without cooling, and then the agents start controlling the AC for the rest of the week. The dotted line on June 29th indicates the start. It is visible that the setpoint does not exceed the upper bound after this day.

### 4.2.2 Scenario 2: Scaling the number of zones

The second scenario considers structural changes in the building, e.g., a change in the number of zones. Again, the agents themselves will not be able to detect these changes because the KNX 3rd Party API will not update unless these changes are present in the ETS project. Once these updates are visible in the gateway, a new agent can be created for the corresponding zone. This requires creating a new Node.js server that consumes the TD typically used by temperature-controlling agents. Apart from assigning a new port to this agent, no further changes are necessary.

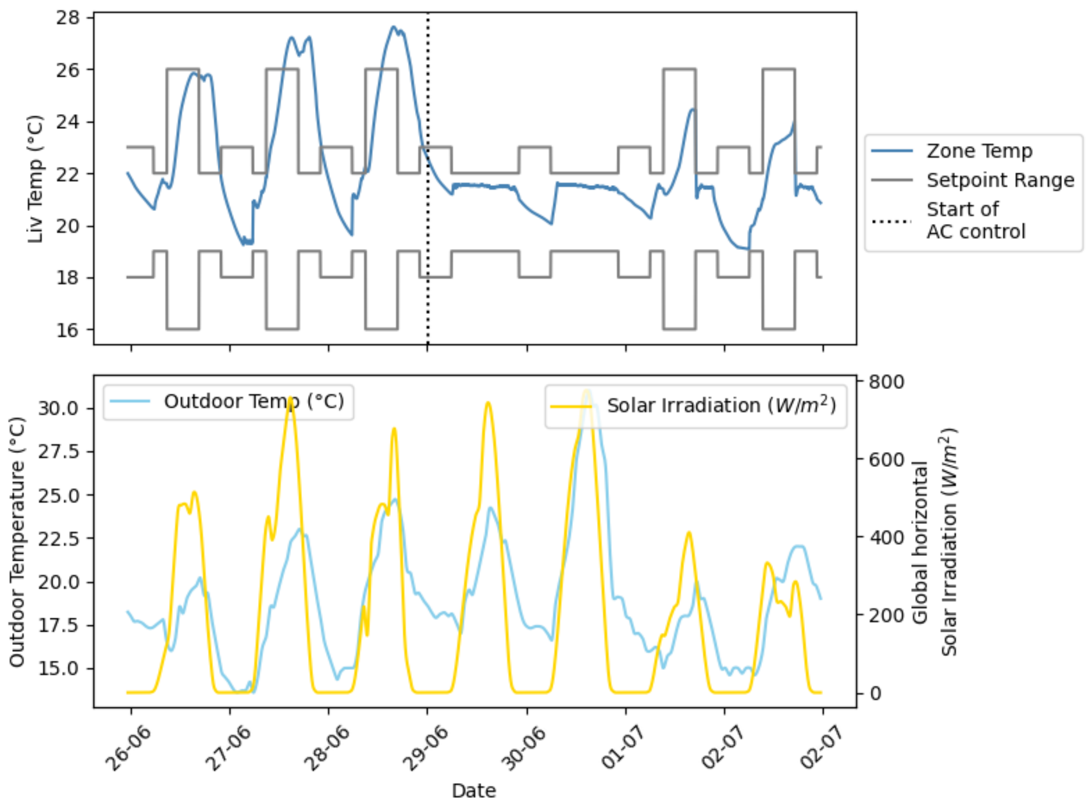


Figure 4.4: The indoor temperature in the living room when AC is included.

### 4.2.3 Scenario 3: Including additional agents

Finally, the case is considered where a new skill is being added to the system, such as exchanging the current rule-based agents with MPC agents. Of course, this would require a more complex control mechanism implemented in each agent, as well as the inclusion of other agents that provide potentially missing data. Inputs to a MPC agent include additional data, such as parameters of the building, like the thermal model of a zone. Therefore, an additional agent might also be introduced if the information is not hard-coded within the MPC agent. To discover an agent providing this information, the MPC agent looks for an agent with the context-capability pair *BuildingParameter - Predict* to request information about the RC components relevant to the zone. Weather data, occupancy, preferences, and current building conditions are already present in the current implementation and can, therefore, also be utilized for computation.

Similarly, when other agents are included in the system that would make it possible to improve the performance of a controller, there has to be an agent that can include the data of the newly introduced agent in its computations. Assume the implemented system was extended by an agent that is able to predict the energy price. This agent would not change the behavior of the MAS as no agent, especially the thermal comfort agent, is capable of including the energy price in its computation.

Nevertheless, an attempt has been made to show how the system would behave if an additional agent were added, allowing other agents to use the information it provides. In Figure 4.5, the MAS was run without the occupancy agent for 3.5 days. Then, the occupancy agent was added to the system so that the temperature agent was able to save energy during longer periods of absence for the rest of the simulation. The figure only depicts the temperature in the living room but clearly shows how the setpoint range is adjusted after the 3rd of February due to an occupied period without any presence in the living room; hence, the new agent was included seamlessly.

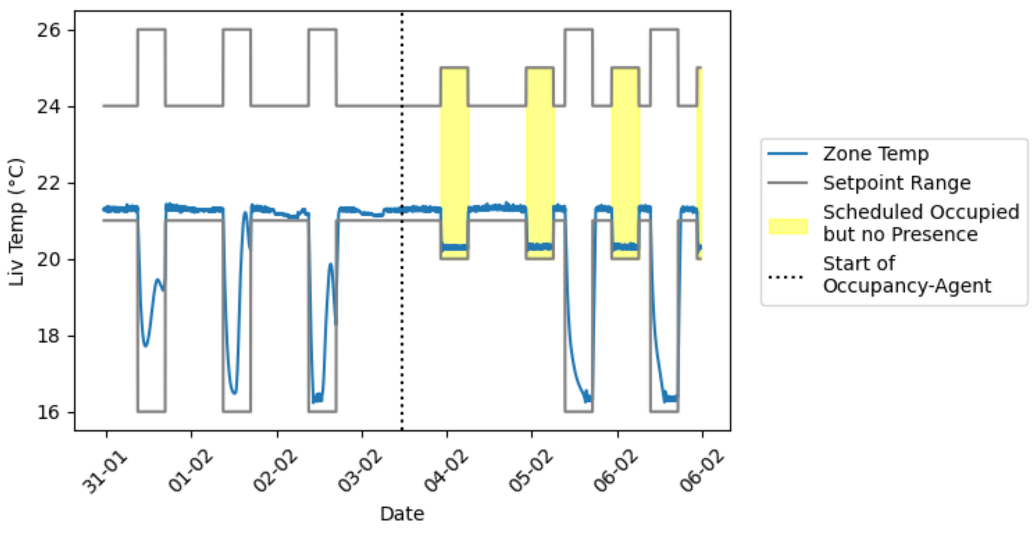


Figure 4.5: The setpoint ranges if the occupancy agent has been added during the simulation.

To evaluate the effects of including a new agent, the energy consumption of the same period without the occupancy agent is compared to the performance of the complete MAS, presented in Section 4.1. The original controller tries to reduce power consumption by lowering the setpoint during unoccupied periods. This is the main difference between the two simulations. The required heating power is summarized in Table 4.3.

The controller only consumed 29.03 kWh more during this experiment, compared to the simulation that includes the occupancy agent right from the start. Apart from lowered energy consumption, the main reason for this experiment was to show the automatic integration of new agents, which has been successful. As the agents continuously poll data from the TDD, they can include the data upon the point in time where the agent publishes its TD.

| Room | Power Consumption kWh |
|---|---:|
| Liv | 333.25 |
| Bth | 26.44 |
| Ro1 | 80.17 |
| Ro2 | 85.94 |
| Ro3 | 117.24 |
| Whole building | 643.04 |

Table 4.3: The consumed energy when running the simulation without occupancy predictions.

CHAPTER 5

# Discussion

In this chapter, the results that were achieved are discussed. First, it is analyzed if the implemented system is comparable to other controllers, such as the benchmark, and what additional considerations must be made. Next, the effectiveness of the KNX IoT 3$^{rd}$ Party API as an interface to the building will be discussed. Finally, it is evaluated if the MAS fulfills the purpose of creating an extendable energy management solution.

## 5.1 Analysis of the Proof of Concept Implementation

As previously presented in Section 4.1, the controller achieves relatively low occupancy discomfort and decreased power consumption in comparison to the benchmark controller. The discomfort has been dramatically reduced, which is also caused by fine-tuning the controller for this building. Obviously, it will behave very differently in other buildings. For example, the heating setpoint is adjusted to be 0.5 Kelvin above the actual preferred value to reduce accumulated discomfort due to slight variations. The benchmark controller does not use this knowledge. Therefore, even slight deviations from the setpoint might inflict discomfort. Another crucial parameter that was estimated is $T_o$, which is used to adjust the output of the sigmoid function that controls the valve position. These values have been determined manually. An agent could have done this work during a dedicated phase where the parameters of the controllers are estimated. Nevertheless, this parameter is used to ensure that the valve position corresponds to the actual heat requirement of this room. Different radiators will also require different offsets. Parameters that are way more important for the system's overall functionality are the PID terms, which also have to be computed before the MAS can properly compute control values.

Furthermore, it is worth noting that the BOPTEST forecasts are always correct. It is not possible for residents to show up in a zone earlier than predicted. This scenario is unrealistic because residents might change rooms as they prefer, which may not align with their typical behavior. If these predictions were not perfect, the results, especially the occupant discomfort, could differ.

Even though some effects do not occur in an actual building, the implemented controller shows good performance. The discomfort is drastically reduced by also utilizing slightly less energy. Some parameters, especially the PID terms, have to be estimated to allow a successful operation of the system. Other controllers, such as the benchmark controller or MPC controllers, also require this step. Nevertheless, the goal of the system is not to define a mechanism that allows skipping the process of tuning the controller. However, such behavior could be implemented with an additional agent. The following sections analyze the extendability of the MAS.

## 5.2 Discussion of Research Question 1

This section analyzes whether the first research question has been answered. The question states the following:

> **Research question 1**:
> *What information has to be additionally provided along the KNX-IoT information model in order to be capable of controlling and optimizing the thermal comfort of a building with a multi-agent system?*

The main interface for any control system, not only the one introduced here, will interact with the newly developed interface from KNX to read and write data. As already mentioned in Section 2.7, the modeling options in ETS are very limited, making it difficult for any sophisticated controller to operate. A thermal model, which is required for many controllers, cannot be queried from the KNX interface. Therefore, this information has to be provided somewhere else. Such information can be provided somewhere else inside of the system, but a matching between the zones extracted from the KNX 3$^{\text{rd}}$ Party API and the other source is necessary. The ETS designer and the person creating the thermal model have to agree on a common vocabulary to automatically interlink the models later.

Furthermore, the modeling options inside of the ETS have to be standardized to infer information automatically. A device can be assigned to a room, but it is unclear whether this refers to the device's physical location or the room where it operates. Consequently, it is impossible to fully automatically infer which heating system is used or even which rooms are heated and air-conditioned. It might be the case that such information is only currently not possible to be modeled as the KNX 3$^{\text{rd}}$ Party API is still under development. Therefore, it is recommended that upon release of the API, the designer is also able to make use of the subclasses from the KNX Information Model. Moreover, allowing the designer to define custom functional blocks or application functions would also reduce manual intervention after modeling the building.

Although there exist some challenges when modeling the building inside of the ETS, it is possible to create basic rule-based controllers assuming that the full potential of the

KNX ontology is used and it can be determined which rooms are heated and cooled by querying the ontology.

With this information, the first research question can be answered: The ETS developer has to use the optional modeling methods and create functions for heating and cooling a zone. These functions must be of a respective function block class in order to query the ontology later. Also, the corresponding devices must be added to these functions to apply control values subsequently. With the semantic export of the ETS project, it is then possible to use basic controllers. For more sophisticated ones, there must be more modeling options, or there has to be a way to match other building models to the KNX model. The presented parameters in Table 2.1 present typical values that are used in such control systems, and control engineers would greatly benefit if these were also added to the system. Nevertheless, KNX is not likely to add these additional modeling options because the primary goal of using the ETS is not to create a BIM but to program devices inside of a building using a standardized communication specified by them. Therefore, developing a way to combine the information given in a BIM and the model exported from the ETS is part of future work.

## 5.3 Discussion of Research Question 2

Before analyzing the findings, the second research question is restated:

> **Research question 2**:
> *What is an appropriate architectural design within a multi-agent building energy management system to facilitate a semi-automatic integration by using existing information from the KNX IoT interface to provide a scalable and extendable energy optimization solution?*

The plausible scenarios that cause a change in the MAS are described in the previous section 4.2. The three different scenarios are now analyzed, and it is determined whether the second research question has been answered using this approach.

### Scenario 1: The integration of new equipment.

The first scenario describes the integration of new devices into the system. As the device must be added to the ETS, a new semantic export has to be created, and the KNX IoT 3$^{rd}$ Party API has to be restarted. Fully automatic integration is not possible. Nevertheless, once the new device is visible inside the KNX gateway, everything else will work automatically. Note that integrating a new device requires a corresponding agent that can control it. Furthermore, other agents must be capable of including data of added agents. If not, these agents must also be updated as well. For instance, in the implemented system, it would not be possible to include a solar thermal system, as the agent controlling the boiler does not support the functionality of considering other heating methods since BOPTEST does not include this option either. Nevertheless, a more sophisticated implementation could delegate the control of the boiler to another agent that also communicates with other devices with context-capability pairs of relevance, e.g., *SolarThermalCollector - ExternalSystemControl*. Including new devices might also inflict an overhead on the rest of the system. Existing agents might require vast changes if no control logic is implemented to include the new device. Moreover, the new agents might also have to be fine-tuned by an engineer as, e.g., PID terms vary from building to building. Nevertheless, there could be agents instantiated inside of the system that help determine such parameters, which reduces the effort of integrating new agents. Another advantage of the proposed system is that downtimes are minimized. As the system is very modular, only specific agents can be exchanged or tuned while the rest of the system stays operational.

**Scenario 2: Scaling the number of zones.**

Adding new zones will require similar steps as for including new equipment. Once the changes are visible in the KNX IoT 3rd Party API, an agent is instantiated and starts controlling the corresponding zone. Checking the building interface for new zones and creating new agents could be done automatically. Nevertheless, scaling the number of zones will also affect the number of servers, as each agent is a Node.js server. For larger buildings with many zones, there are many ports to manage, as well as the infrastructure on which each agent runs. Despite the fact that such WoT servers are capable of running on devices with restricted computational power, the overhead each server causes is higher compared to simpler implementations that might not be decentralized. The advantage of this decentralized system is that the integration of new zones is in a semi-automatic fashion.

**Scenario 3: Including additional agents.**

As described in the previous chapter, introducing a new agent might also require additional information-providing agents. This can be done efficiently, as only an agent has to be created. It is also possible to extend the existing system by adding an agent. This agent will provide information that can be used by existing agents. This has been demonstrated in Section 4.2.3. For the implemented MAS, a replaced agent must have the same outputs as the previous one, as the building interface only accepts a restricted set of control values, e.g., a valve position. If the building interface agent provides additional methods that allow setting different control values, the same outputs are not required. This allows the owner of the building to choose between different controllers without changing other agents. This increases the flexibility of the system. Moreover, the system can be easily extended by additional agents.

One goal was to find an appropriate architectural design of a MAS to allow semi-automatic integration and scalability. The developed system shows how the developed ontology can be used to create such an energy management system. The agents-skills-ontology can be used to describe which information an agent requires and which data it can provide to other agents. By considering multiple aspects of BMS as well as different control strategies, an ontology has been created that allows modeling complex MASs. By describing the skills of the agents, the resulting system is extendable as well as scalable. Depending on which changes are made to the system, more or less interaction by a system designer is required. Therefore, the proposed system also meets the desired semi-automatic integration. Furthermore, utilizing WoT devices to create an extendable system has also proven effective, as these provided the means of including semantics, reducing ambiguity, and increasing interoperability. Therefore, the provided proof of concept implementations shows a possible solution as answer to the second research question.

CHAPTER 6

# Conclusion and Future Work

In this thesis, a MAS has been proposed that provides a remedy for the centralized control strategies with its complex optimization used in many building controllers. With an ontology describing each agent's skills, the system can be extended and scaled easily. Furthermore, the system builds upon the existing standard KNX IoT 3$^{\text{rd}}$ Party API and utilizes the concept of WoT to reduce ambiguity during communication.

The research has demonstrated that the use of the skills-ontology allows the system designer to model a great variety of different systems. It can include different heating and cooling systems, light control, access control, heating distribution, weather and energy price predictions, user preferences, and many other important skills. By including these skills in an agent's TD and publishing it, agents can automatically find each other and start communicating. With the use of other ontologies in the description of Thing's methods, the agents can also determine which inputs are expected and what data is returned.

Nevertheless, despite the promising outcomes, a fully automatic integration of new components is not possible. Changes can only be detected once these have been modeled inside of the ETS. Also, the tedious process of tuning a controller is not done automatically, even though this would significantly reduce the integration effort.

The KNX IoT 3$^{\text{rd}}$ Party API has the potential of supporting the use of a basic MAS if the full modeling options are available in the ETS. However, despite the promising capabilities of the KNX Information Model, it is not possible to utilize these in the current state of the KNX IoT 3$^{\text{rd}}$ Party API. Furthermore, some modeling options, like parameters of the building envelope, will most likely never be modeled inside of the ETS.

Based on the findings in this thesis, there are some directions for future research. As already noted previously, a way of combining the exported ontology from the ETS and a BIM would help to infer more information automatically. Further investigations are also necessary to see how the implemented system behaves on other buildings, with a greater number of controllable assets, like photovoltaics or other RES. Integrating more advanced control strategies would also achieve better results, especially if predictions are no longer always true. Finally, in-depth research can be carried out on how the system can also include other standards like BACnet or Modbus, as these technologies are also commonly used in building automation.

Overall, this thesis has provided the fundamentals for a skill-based MAS in the BMS domain that provides a decentralized and extendable energy management solution and opens up multiple directions for future research topics.

# Overview of Tools Used

**DeepL** to improve complex sentences in order to achieve better readability.

**Grammarly** to correct errors while writing the thesis.

# List of Figures

# List of Tables

# List of source codes

# Acronyms

**AC** Air Conditioning. 24, 59, 69, 70, 83

**ACL** Agent Communication Language. 13, 14

**API** Application Programming Interfaces. 3, 9, 15, 24, 27–32, 46–48, 68, 69, 73–76, 79

**BIM** Building Information Model. 30, 75, 79

**BMS** Building Management Systems. 2, 3, 7, 34, 76, 80

**ETS** Engineering Tool Software. ix, 3, 4, 28, 29, 31, 45–48, 69, 74, 75, 79

**HTTP** Hypertext Transfer Protocol. 15, 20, 21, 28, 40, 54

**HVAC** Heating, Ventilation and Air Conditioning. 24, 26, 34, 42

**IoT** Internet of Things. 1, 19

**JSON** JavaScript Object Notation. 19

**JSON-LD** JSON for Linked Data. 19, 20, 48

**MAS** Multi-Agent System. ix, 2–5, 7, 11–16, 18, 19, 21, 28, 29, 31–33, 35, 38, 42, 45, 48, 54, 55, 57, 63, 65–68, 71, 73–77, 79, 80, 83, 85

**MaSE** Multiagent Systems Engineering. 4, 16–19, 31, 33, 34, 36–38, 40, 42, 49, 83

**MPC** Model Predictive Control. 3, 11, 21, 22, 24, 26, 27, 51, 52, 68, 70, 74

**MSS** Microservice System. 14, 15

**OOP** Object-Oriented Programming. 7, 8

**OWL** Web Ontology Language. 8

**PASSI** Process for Agent Societies Specification and Implementation. 18, 19, 33

| | |
|---|---|
| **PI** | Proportional-Integral. 66 |
| **PID** | Proportional-Integral-Derivative. 45, 53, 54, 59, 60, 73, 74, 76 |
| **RC** | Resistor–Capacitor. 21–24 |
| **RDF** | Resource Description Framework. 8, 19, 21, 46 |
| **REC** | Real Estate Core. 9, 52, 87 |
| **RES** | Renewable Energy Sources. 1, 11, 34, 80 |
| **SPARQL** | SPARQL Protocol and RDF Query Language. 21 |
| **SRF** | Solar Response Factor. 23 |
| **TD** | Thing Description. ix, 19–21, 31, 32, 34, 48–51, 54–56, 58, 60, 61, 69, 71, 79, 87 |
| **TDD** | Thing Description Directory. 19–21, 31, 34, 48, 55, 60, 69, 71 |
| **UML** | Unified Modeling Language. 18 |
| **URI** | Unique Resource Identification. 8, 19 |
| **URL** | Uniform Resource Locator. 19, 54, 55, 69 |
| **WoT** | Web of Things. vii, ix, 2, 3, 7, 19–21, 40, 48, 55, 60, 76, 77, 79 |

# Bibliography

[AAC$^+$15] Fatima Amara, Kodjo Agbossou, Alben Cárdenas, Yves Dubé, and Sousso Kelouwani. Comparison and simulation of building thermal models for effective energy management. *Smart Grid and Renewable Energy*, 06:95–112, 2015.

[AH11] Dean Allemang and James A Hendler. *Semantic web for the working ontologist : effective modeling in RDFS and OWL*. Morgan Kaufmann, Waltham, MA, 2. ed. edition, 2011.

[BB14] Jonathan Brooks and Prabir Barooah. Energy-efficient control of under-actuated hvac zones in buildings. In *2014 American Control Conference*, pages 424–429, 2014.

[BBF$^+$16] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, BuildSys '16, page 41–50, New York, NY, USA, 2016. Association for Computing Machinery.

[BULA$^+$19] Hector Bastida, Carlos E. Ugalde-Loo, Muditha Abeysekera, Meysam Qadrdan, and Jianzhong Wu. Thermal dynamic modelling and temperature controller design for a house. *Energy Procedia*, 158:2800–2805, 2019. Innovative Solutions for Energy Transitions.

[CFM$^+$19] Borui Cui, Cheng Fan, Jeffrey Munk, Ning Mao, Fu Xiao, Jin Dong, and Teja Kuruganti. A hybrid building thermal modeling approach for predicting temperatures in typical, detached, two-story houses. *Applied Energy*, 236:101–116, 2019.

[Con] World Wide Web Consortium. Home - web of things (wot) - w3.org. `https://www.w3.org/WoT/`. [Accessed 09-07-2024].

[CS14] Massimo Cossentino and Valeria Seidita. *PASSI: Process for Agent Societies Specification and Implementation*, pages 287–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[DAC+20] Ján Drgoňa, Javier Arroyo, Iago Cupeiro Figueroa, David Blum, Krzysztof Arendt, Donghun Kim, Enric Perarnau Ollé, Juraj Oravec, Michael Wetter, Draguna L. Vrabie, and Lieve Helsen. All you need to know about model predictive control for buildings. *Annual Reviews in Control*, 50:190–232, 2020.

[DB00] P. Davidsson and M. Boman. A multi-agent system for controlling intelligent buildings. In *Proceedings Fourth International Conference on MultiAgent Systems*, pages 377–378, 2000.

[DJS+21] Blum David, Arroyo Javier, Huang Sen, Drgoňa Ján, Jorissen Filip, Taxt Walnum Harald, Chen Yan, Benne Kyle, Vrabie Draguna, Wetter Michael, and Helsen Lieve. Building optimization testing framework (boptest) for simulation-based benchmarking of control strategies in buildings. *Journal of Building Performance Simulation*, 14(5):586–610, 2021.

[DKJ18] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018.

[Dor20] Kirill Dorofeev. Skill-based engineering in industrial automation domain: Skills modeling and orchestration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 158–161, 2020.

[DWS01] Scott Deloach, Mark Wood, and Clint Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11:231–258, 06 2001.

[EM15] G. Evola and L. Marletta. The solar response factor to calculate the cooling load induced by solar gains. *Applied Energy*, 160:431–441, 2015.

[FAI15] FAIRsharing Team. Fairsharing record for: Quantities, units, dimensions and types, 2015.

[FIP] FIPA. FIPA Communicative Act Library Specification. `http://www.fipa.org/specs/fipa00037/SC00037J.html`. [Accessed 17-03-2025].

[FLZ+22] Moritz Frahm, Felix Langner, Philipp Zwickel, Jörg Matthes, Ralf Mikut, and Veit Hagenmeyer. How to derive and implement a minimalistic RC model from thermodynamics for the control of thermal parameters for assuring thermal comfort in buildings. In *2022 Open Source Modelling and Simulation of Energy Systems (OSMSES)*, pages 1–6, 2022.

[Frü21] Thomas Frühwirth. *Dependability in multi-agent systems for smart grid applications*. Wien, 2021.

[GCLPVD23] Raúl García-Castro, Maxime Lefrançois, María Poveda-Villalón, and Laura Daniele. The etsi saref ontology for smart applications: a long path of development and evolution. *Energy Smart Appliances: Applications, Methodologies, and Challenges*, pages 183–215, 2023.

[GLW23] Christoph Gehbauer, Eleanor S. Lee, and Taoning Wang. An evaluation of the demand response potential of integrated dynamic window and hvac systems. *Energy and Buildings*, 298:113481, 2023.

[GTPLC+22] M. González-Torres, L. Pérez-Lombard, Juan F. Coronel, Ismael R. Maestre, and Da Yan. A review on buildings energy information: Trends, end-uses, fuels and drivers. *Energy Reports*, 8:626–637, 2022.

[HWKH19] Karl Hammar, Erik Oskar Wallin, Per Karlberg, and David Hälleberg. The realestatecore ontology. In *The Semantic Web – ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II*, page 130–145, Berlin, Heidelberg, 2019. Springer-Verlag.

[IEA23] IEA. Global CO2 emissions from the operation of buildings in the net zero scenario, 2010-2030, 2023. Licence: CC BY 4.0.

[INC08] Victoria Iordan, Antoanela Naaji, and Alexandru Cicortas. Deriving ontologies using multi-agent systems. *W. Trans. on Comp.*, 7(6):814–826, June 2008.

[JWC96] S.E.G. Jayamaha, N.E. Wijeysundera, and S.K. Chou. Measurement of the heat transfer coefficient for walls. *Building and Environment*, 31(5):399–407, 1996.

[KKM23] Sebastian Käbisch, Ege Korkan, and Michael McCool. Web of things (wot) thing description 1.1. W3C recommendation, W3C, 12 2023. https://www.w3.org/TR/2023/REC-wot-thing-description11-20231205/.

[KNX23] KNX. KNX Information Model. `https://schema.knx.org/`, 2023. [Accessed 23-10-2024].

[LBW+22] Manuel Lämmle, Constanze Bongs, Jeannette Wapler, Danny Günther, Stefan Hess, Michael Kropp, and Sebastian Herkel. Performance of air and ground source heat pumps retrofitted to radiator heating systems and measures to reduce space heating temperatures in existing buildings. *Energy*, 242:122952, 2022.

[MBW+18] Somayeh Malakuti, Jürgen Bock, Michael Weser, Pierre Venet, Patrick Zimmermann, Mathias Wiegand, Julian Grothoff, Constantin Wagner, and Andreas Bayha. Challenges in skill-based engineering of industrial automation systems. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 67–74, 2018.

[MKA+23] Miriam Zawadi Muchika, Oudom Kem, Sarra Ben Abbes, Lynda Temal, and Rim Hantach. Achieving interoperability in energy systems through multi-agent systems and semantic web. In *2023 IEEE/WIC International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 441–448, 2023.

[MMK24] Panagiotis Michailidis, Iakovos Michailidis, and Elias Kosmatopoulos. Review and evaluation of multi-agent control applications for energy management in buildings. *Energies*, 17(19), 2024.

[MSC+19] Zheng Ma, Mette Jessen Schultz, Kristoffer Christensen, Magnus Værbak, Yves Demazeau, and Bo Nørregaard Jørgensen. The application of ontologies in multi-agent systems in the energy sector: A scoping review. *Energies*, 12(16), 2019.

[MTCT23] Michael McCool, Kunihiko Toumura, Andrea Cimmino, and Farshid Tavakolizadeh. Web of things (wot) discovery. W3C recommendation, W3C, December 2023. https://www.w3.org/TR/2023/REC-wot-discovery-20231205/.

[PDKH17] Damien Picard, Ján Drgoňa, Michal Kvasnica, and Lieve Helsen. Impact of the controller model complexity on model predictive control performance for buildings. *Energy and Buildings*, 152:739–751, 2017.

[PEFST13] Katia Potiron, Amal El Fallah Seghrouchni, and Patrick Taillibert. *Multi-Agent System Properties*, pages 5–10. Springer London, London, 2013.

[RCR17] Alberto Rivas, Pablo Chamoso, and Sara Rodríguez. An agent-based internet of things platform for distributed real time machine control. In *2017 IEEE 17th International Conference on Ubiquitous Wireless Broadband (ICUWB)*, pages 1–5, 2017.

[RWG+24] Lasse M. Reinpold, Lukas P. Wagner, Felix Gehlhoff, Malte Ramonat, Maximilian Kilthau, Milapji S. Gill, Jonathan T. Reif, Vincent Henkel, Lena Scholz, and Alexander Fay. Systematic comparison of software agents and digital twins: differences, similarities, and synergies in industrial production. *Journal of Intelligent Manufacturing*, Jan 2024.

[SH13] Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[SMK24] Christina Sander, Dominik Meyer, and Bernd Klauer. A scalable agent architecture. In *2024 4th International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, pages 1–6, 2024.

[TBB+21] Christian Ankerstjerne Thilker, Peder Bacher, Hjörleifur G. Bergsteinsson, Rune Grønborg Junker, Davide Cali, and Henrik Madsen. Non-linear grey-box modelling for heat dynamics of buildings. *Energy and Buildings*, 252:111457, 2021.

[vHM04] Frank van Harmelen and Deborah McGuinness. OWL web ontology language overview. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-owl-features-20040210/.

[WCOLO19] Rem W. Collier, Eoin O'Neill, David Lillis, and Gregory O'Hare. Mams: Multi-agent microservices. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 655–662, New York, NY, USA, 2019. Association for Computing Machinery.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, Sep 2000.

[WXHG07] Song Wei, Wang Xiangnan, Cao Houji, and Pan Guowei. Multi-agent architecture of energy management system based on iec 61970 cim. In *2007 International Power Engineering Conference (IPEC 2007)*, pages 1366–1370, 2007.

[YSX+21] Liang Yu, Yi Sun, Zhanbo Xu, Chao Shen, Dong Yue, Tao Jiang, and Xiaohong Guan. Multi-agent deep reinforcement learning for hvac control in commercial buildings. *IEEE Transactions on Smart Grid*, 12(1):407–419, 2021.