A Metamodel for Mechanical, Electrical, and Plumbing Systems: Enabling Interoperability and Control Integration in Building Management

Peter Yefi

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Software Engineering) at

Concordia University

Montréal, Québec, Canada

May 2026

© Peter Yefi, 2026

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Peter Yefi**

Entitled: **A Metamodel for Mechanical, Electrical, and Plumbing Systems: Enabling Interoperability and Control Integration in Building Management**

and submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________ Chair
Dr. Lata Narayanan

______________________ External Examiner
Dr. Burcin Becerik-Gerber

______________________ Examiner
Dr. Sandra Cespedes

______________________ Examiner
Dr. Weiyi Shang

______________________ Examiner
Dr. Mazdak Nikbakht

______________________ Supervisor
Dr. Yann-Gaël Guéhéneuc

______________________ Co-supervisor
Dr. Ursula Eicker

Approved by ______________________
Dr. Paquet, Joey, Chair
Department of Computer Science and Software Engineering

__________ 2026

____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

A Metamodel for Mechanical, Electrical, and Plumbing Systems: Enabling Interoperability and Control Integration in Building Management

Peter Yefi, Ph.D.

Concordia University, 2026

Buildings are complex systems that integrate mechanical, electrical, and plumbing (MEP) subsystems—such as HVAC, lighting, and fire safety—to optimise resource use while ensuring occupant comfort. These systems are managed by Building Management Systems (BMS), yet many buildings remain energy inefficient, contributing significantly to carbon emissions. A key barrier to scalable energy solutions is the heterogeneous, proprietary nature of BMS, which severely hinders data interoperability and the implementation of advanced control strategies.

While researchers use traditional RDF/OWL-based ontologies to model building systems and address this heterogeneity, these ontologies typically lack the mechanisms necessary to enforce constraints, model dynamic behaviour, and integrate executable control logic.

This thesis addresses these gaps by introducing **MetamEnTh**, an object-oriented metamodel for modelling the operational phase of MEP subsystems in buildings. We adopt a grounded-theory approach, informed by surveys and interviews with researchers and industry practitioners, to develop **MetamEnTh**. It employs structured classes with predefined relationships, methods, and validation rules to ensure accurate, dynamic models, complemented by interfaces that enable the integration of user-defined control logic.

We also evaluate the APIs of common BMS platforms, assessing their capabilities for exposing system data and integrating control logic. We validate **MetamEnTh** through real-world use cases and a comparative evaluation against existing ontologies, demonstrating improved accuracy, constraint enforcement, and enhanced error prevention. **MetamEnTh** provides a practical, extensible metamodel that supports data-driven, energy-efficient building management by unifying data

semantics and operational control.

Future work will broaden **MetamEnTh** by incorporating classes for occupancy modelling and utility methods for energy-efficiency tasks. We will extend our BMS API evaluation to cover predictive control and digital twin integration. To support practitioners, we will develop user-friendly editors while aligning **MetamEnTh** with community initiatives such as ASHRAE, Brick, and Project Haystack. **MetamEnTh** opens new avenues for validated and executable representations of building systems, providing a foundation for future research on model-driven digital twins, self-adaptive control, and cross-domain interoperability in cyber-physical infrastructures. By unifying data and control logic, it enables researchers to experiment with novel AI- and simulation-driven methods for optimising building energy use.

Acknowledgments

I thank God for His grace, strength, and guidance throughout this journey. I am deeply grateful to my supervisor, Dr Yann-Gaël Guéhéneuc, and my co-supervisor, Dr Ursula Eicker, for their invaluable support and mentorship. I also sincerely thank Dr Ramanunni Parakkal Menon for his insightful feedback and encouragement. To my wife, thank you for your love, patience, and unwavering support. I am also grateful to my siblings for their unwavering support and to my lab colleagues for their collaboration and friendship.

This thesis is dedicated to the memory of my beloved sister, Mrs Elizabeth Awuntuba. Your kindness and support continue to inspire me.

Contents

List of Figures **x**

List of Tables **xiii**

1 Introduction **1**

1.1 Context 1

1.2 Thesis Statement 5

1.3 Problem Statement 5

1.4 Research Questions 6

1.5 List of Publications 8

1.6 Outline 9

2 Literature Review **10**

2.1 Object-oriented Modelling of MEP Systems 10

2.1.1 Project Haystack and Brick 14

2.2 Interoperable Data Access for MEP Systems 23

2.3 Integrating Control Logic in Object-Oriented MEP Models 26

3 Object-oriented Modelling of MEP Systems **29**

3.1 Introduction 29

3.1.1 Metamodel, Model, and Ontology 30

3.2 Approach 31

3.2.1 Needs Assessment 32

3.2.2 Design of MetamEnTh 37
3.3 Evaluation 47
3.3.1 Case Study: Evaluation of UML Designs 47
3.3.2 Case Study: Practical and Empirical Evaluation 53
3.4 Discussions 66
3.4.1 Approach 66
3.4.2 Evaluation 67
3.5 Threats to Validity 68
3.5.1 Internal Validity 68
3.5.2 External Validity 68
3.5.3 Construct Validity 69
3.5.4 Conclusion Validity 69
3.6 Conclusion 69

4 Interoperable Data Access for MEP Systems 77
4.1 Introduction 77
4.2 Approach 78
4.2.1 Needs Assessment 79
4.2.2 Interoperable Data Access of Building Systems 79
4.2.3 BMS APIs Capabilities Enabling Interoperable Access to BMS 86
4.3 Evaluation 112
4.3.1 Use Case: Varennes and Webster Libraries 112
4.3.2 Use Case: BMS API Real-time Data Access 114
4.4 Discussions 121
4.4.1 Approach 121
4.4.2 Evaluation 123
4.5 Threats to Validity 124
4.5.1 Construct Validity 124
4.5.2 Internal Validity 124

4.5.3 External Validity . . . 125
4.5.4 Conclusion Validity . . . 125
4.6 Conclusion . . . 125

5 Integrating Control Logic in Object-Oriented MEP Models 127
5.1 Introduction . . . 127
5.2 Approach . . . 129
5.2.1 Needs Assessment . . . 130
5.2.2 Interfaces for Control Logic Integration . . . 131
5.2.3 MetamEnTh Classes and Attributes Enabling Control Logic Integration . . . 134
5.3 Evaluation . . . 140
5.3.1 Use Case: Webster Library . . . 140
5.3.2 Use Case: Ludwigsburg Residential and Commercial Building . . . 145
5.4 Discussions . . . 150
5.4.1 Approach . . . 151
5.4.2 Evaluation . . . 152
5.5 Threats to Validity . . . 153
5.5.1 Construct validity . . . 153
5.5.2 Internal validity . . . 153
5.5.3 External validity . . . 153
5.5.4 Conclusion validity . . . 153
5.6 Conclusion . . . 153

6 Conclusion 155
6.1 Summary of Contributions . . . 157
6.1.1 RQ1 & RQ2 . . . 157
6.1.2 RQ3 . . . 158
6.1.3 RQ4 . . . 159
6.2 Answer . . . 159
6.3 Future Work . . . 160

6.3.1 Short-Term 160
6.3.2 Mid-Term 161
6.3.3 Long-Term 161

Bibliography **163**

Appendix A **181**

List of Figures

Figure 2.1 A Project Haystack sample project definition for a building in Gaithersburg 16

Figure 2.2 Brick information concepts and representations 17

Figure 2.3 Thermal zones of the tenth floor of Concordia University's EV buildings as captured by the installed BMS 20

Figure 2.4 The Brick class hierarchy illustrating the five levels of inheritance leading to the *Return_Air_CO2_Sensor* class 21

Figure 3.1 **Demographic information of the survey respondents.** The figure shows that the majority of respondents have more than three years of experience, with 50% having over five years of experience. Additionally, 50% of respondents identify as having advanced expertise in their respective fields. 35

Figure 3.2 **Familiarity of respondents with built environment modelling tools.** The figure indicates that respondents have experience with a range of built environment modelling tools, with four respondents reporting familiarity with Brick, gbXML, IFC, and BEDES. 36

Figure 3.3 **Modal Responses to Building Model Feature Survey.** The figure illustrates the distribution of the most frequently selected responses (mode) for each Survey Question (SQ) on a 5-point Likert scale, where 1 represents the lowest perceived importance/difficulty and 5 represents the highest. **Key Finding:** 50% of respondents assigned the highest rating (5) to SQ2, SQ4, and SQ7, underscoring the perceived critical nature (importance or difficulty) of these three concerns. 37

Figure 3.4 **Respondents' ratings of building model features.** The first plot shows respondents' selections of building systems and entities they consider relevant for energy efficiency. The second plot summarises the most important concerns when modelling buildings for energy-efficient operations, concerning accuracy, flexibility, and ease of use. 38
Figure 3.5 A high-level meta-metamodel for built environments showing some classes and their relationships 40
Figure 3.6 An instance of the high-level meta-metamodel illustrating a valid but inaccurate relationship between a damper and a fan 41
Figure 3.9 **MetamEnTh** `Appliance`, illustrating the `Appliance` class and its relationships with `AbstractTransducer` and spatial entities (`AbstractSpace`) 44
Figure 3.12 Interconnected through a complex network of pipes, the boilers, chillers, and cooling towers form the principal ventilation systems of the MB/JMSB Building. . 50
Figure 3.13 A **MetamEnTh** modelling of the building in Figure 2.1 on Page 16 51
Figure 3.16 Observed versus reported responses of participants on their ability to create various relationships between entities in Task Two 57
Figure 3.17 A section of the architectural drawings of the LB Building showing labelled (e.g., RT_D_P) block covers with their dimensions and further information (below the block covers) pointing to details of the layers. 60
Figure 3.18 **MetamEnTh** representation of the CO_2 sensor as illustrated in the Brick class hierarchy in Figure 2.4 62
Figure 3.7 **MetamEnTh**, a complex low-level model instance showing domain-specific classes, their relationships and packages to which they belong 71
Figure 3.8 **MetamEnTh** Structure, showing the various classes, their properties, and relationships 72

Figure 3.10 **MetamEnTh** HVAC control system represented as a network of HVAC components connected through pipes and ducts. The design decision was taken in consultation with an expert to ensure that **MetamEnTh** can model the network of HVAC entities through ducts and pipes. 73
Figure 3.11 **MetamEnTh** Energy System showing the different energy systems, how they relate with themselves and with spatial entities (**AbstractFloorSpace**), **AbstractSubSystem** and **AbstractTransducer** 74
Figure 3.14 An HVAC system showing ventilation ducts, HVAC components, and airflow to rooms and open spaces, highlighting the distribution of conditioned air throughout the building for optimal climate control. 75
Figure 3.15 Demographic distribution of experiment participants by job title, years of programming, experience rating, level of education, country and age group 76
Figure 4.1 A five-layered architecture showing how different clients can access transmutable data from buildings and their systems 81
Figure 4.2 Time of day temperature variability analysis based on models for the Varennes Library (Enteliweb BMS) and LB Building (Desigo BMS) 115
Figure 5.1 **MetamEnTh** classes that enable the integration of user-implemented control logic 134
Figure 5.2 The IoT setup in office Unit 3 of the residential and commercial building in Ludwigsburg shows how various devices use LoRaWAN for communication and expose their modelled entities through FIWARE REST APIs 146
Figure A.1 A UML instance of selected sections of the EV building showing various entities like building, floors, rooms, open spaces, meter, weather station, weather data, sensors, zones, and ventilation ducts 193
Figure A.2 A UML instance model of selected sections of the MB/JMSB Building, highlighting additional entities—such as dampers, variable air volume boxes, baseboard heaters, actuators, solar PV panels, setpoints, and duct connections—not modelled in the EV Building use case 194

List of Tables

Table 2.1 Mapping of research questions to gaps and relevant studies: Summary 28
Table 3.1 Summary statistics for questionnaire responses comparing Brick and **MetamEnTh** 56
Table 3.2 Results of statistical tests comparing Brick and **MetamEnTh** responses . . . 56
Table 3.3 Layer composition of cover RT-B1 of the LB Building 59
Table 4.1 BMS and their integration APIs 85
Table 4.2 APIs of five (5) BMS vendors and their capabilities grouped into 12 categories 91
Table 4.3 Selected endpoints of the mocked BMS APIs, their HTTP verbs, responses, and HTTP status codes. The responses and status codes are limited to successful HTTP requests. 102
Table 4.4 Common Building Applications, Building Entities and their Relationships, and API Requirements 107
Table 4.5 Comparison of Brick and Project Haystack Entities for Representing BMS Data Acquired via APIs Across 12 Functional Categories 109
Table A.1 Review of literature, identified gaps and their relations with the four research questions 181

Chapter 1

Introduction

1.1 Context

Buildings are complex systems composed of interconnected mechanical, electrical, and plumbing (MEP) subsystems that manage resources, ensure occupant safety, and maintain optimal indoor environmental conditions (IECs). Building management systems (BMS), which centrally monitor and control building subsystems, are indispensable for their role in monitoring and controlling these subsystems—particularly HVAC, lighting, electrical power, renewable energy systems, shading, access control, fire and life safety, elevators, and sometimes water and waste management systems—to improve energy efficiency while preserving comfort and safety (Manic, Wijayasekara, Amarasinghe, & Rodriguez-Andina, 2016). A typical BMS consists of three layers: (1) the *field device level* (e.g., dampers, VAV boxes, fans, chillers, sensors, thermostats), (2) the *automation level* (controllers and routers), and (3) the *management level*, where servers and software with graphical interfaces allow facility managers to interact with the system (Hamza, 2018). At the management layer, BMSs represent building systems using internal data models derived from protocol specifications—for example, BACnet object types and properties (Bill Swan, 2022).

Despite the deployment of BMS, buildings continue to exhibit substantial inefficiencies, with up to 30% of energy usage wasted (Y. Wang, Kuckelkorn, & Liu, 2017). According to the US Energy Information Administration (EIA), commercial buildings consumed 6.8 quadrillion British thermal units (Btu) and incurred $141 billion in energy costs in 2018 (US EIA, 2018). Similarly,

the United Nations Climate Technology Centre and Network (CTCN) attributes 25–40% of energy consumption in OECD countries to buildings (UN CTCN, 2001).

HVAC systems, which constitute a major portion of MEP systems, consume approximately 30% of a building's energy and contribute significantly to carbon dioxide emissions (Wikipedia contributors, 2025). Due to the devastating nature of carbon emissions to the environment, the European Union Energy Performance of Buildings Directive (Energy Performance of Buildings Directive, 2023), reinforced by the European Climate Law (European Parliament and Council of the European Union, 2021), aims to reduce greenhouse gas emissions by 55% by 2030. Canada's Net-Zero Emissions Accountability Act outlines a legislative roadmap to achieve net-zero emissions by 2050 (Government of Canada, 2022).

Facility managers, the professionals responsible for operating and maintaining building systems, often configure BMSs in static and suboptimal ways (DMA Group, n.d.). Such configurations lead to considerable energy waste and further increase carbon emissions. The inefficiency resulting from these configurations, coupled with the potential for significant energy savings, has encouraged researchers to explore new approaches to building control and data management. However, the lack of standardised access to building data (Bhattacharya, 2016) hampers these efforts. Current BMSs utilise vendor-specific data representations, which are often opaque to both humans and machines, thereby limiting interoperability (Chamari, Petrova, & Pauwels, 2022).

Moreover, modern building management systems lack programmability because there are no abstraction layers between the control logic and the physical systems. Applications directly interface with low-level sensors and actuators, which results in tightly coupled, non-portable implementations (Dawson-Haggerty et al., 2013).

The increasing digitalisation of the Architecture, Engineering, and Construction (AEC) industry has led to the development of Building Information Modelling (BIM) as an approach for representing the physical and functional characteristics of buildings (Becerik-Gerber & Kensek, 2010). While BIM has advanced design coordination and construction workflows, it remains limited in representing operational data and control logic required to manage buildings post-commissioning.

To bridge these gaps, researchers have turned to semantic Web technologies and ontologies based on the Resource Description Framework (RDF) (W3C RDF Working Group, 2023) and Web

Ontology Language (OWL) (W3C Semantic Web Wiki, 2025). These technologies offer flexible mechanisms for representing MEP subsystems as RDF triples and OWL constructs. Their flexibility lies in their capacity to express domain knowledge through subject–predicate–object triples, enabling consistent, machine-interpretable descriptions of entities, attributes, and interconnections. However, this flexibility often leads to inconsistencies and inaccuracies in the models created, as these ontologies lack built-in validation mechanisms. Modellers must conduct validation separately because the modelling process does not integrate it, which complicates workflows, especially when these models evolve. Furthermore, ontologies do not inherently model the dynamic nature of systems, and they typically output static files, limiting seamless integration with control systems.

In this thesis, we address eight limitations of current building modelling approaches that focus on the operational phase of buildings. We specifically examine Project Haystack and Brick due to their wide adoption for modelling operational data and relationships. The identified limitations include: L_1 ambiguity in defining entities; L_2 potential for composing flawed entity relationships; L_3 absence of entities for ducts, pipes, and detailed spatial context; L_4 difficulty in validating entities and relationships; L_5 inconsistent data and structural semantics; L_6 complex and deep inheritance hierarchies; L_7 inability to represent system behaviours; and L_8 static artefacts and limited interactivity. We discuss these limitations in detail in Chapter 2. To address them, we propose and implement an object-oriented metamodel—*Metamodel for Energy Things (**MetamEnTh**)*—for modelling building systems, their relationships, and spatial contexts.

We employed a mixed-methods approach, combining surveys and semi-structured interviews with both researchers and practitioners, to inform the design of the proposed modelling approach and to elicit practical requirements relevant to building operation.

The limited scale of the empirical study—comprising 10 survey respondents, 2 interview participants, and 10 experimental participants—limits the generalisability of the results. Nevertheless, the study was designed as an exploratory investigation intended to validate the feasibility and relevance of the proposed modelling concepts rather than to establish statistically generalisable conclusions. Future work will extend this study to include a larger and more diverse group of practitioners and researchers to enhance the generalisability of the conclusions.

While the use of interviews and surveys for needs assessment in built environment modelling is

not new (Chen, Das, Chen, & Cheng, 2020; Gispert, Yitmen, Sadri, & Taheri, 2025), these methods have predominantly been applied to capture high-level domain knowledge or structural requirements. To the best of our knowledge, this work is the first to systematically employ surveys and interviews with both researchers and practitioners as part of the needs assessment and empirical evaluation of a built environment modelling system that explicitly targets the operational aspects of buildings, particularly mechanical, electrical, and plumbing (MEP) systems, in the context of building energy efficiency.

Building on these findings, **MetamEnTh** supports the creation of modular, validated, and executable models of building systems, thereby enhancing interoperability and enabling the integration of operational control logic across heterogeneous Building Management Systems.

To enable the deployment of dynamic **MetamEnTh** models that integrate with MEP subsystems within buildings, we propose a five-layer architecture for accessing standardised BMS data. Within this architecture, we evaluate the APIs of leading BMS vendors at the third layer to assess their effectiveness in supporting model instantiation and facilitating access to interoperable, standardised building data. Based on this evaluation, we develop interfaces to implement control strategies that enable the seamless integration between building models and BMSs. Through three different use cases, we demonstrate **MetamEnTh's** ability to generate models that integrate seamlessly with real-world systems. Comparative empirical studies against Brick—a leading ontology in the domain—reveal that **MetamEnTh** improves fault tolerance, model correctness, and level of details in modelling entities, providing an improved metamodel for modelling and managing energy-efficient buildings.

Overall, this thesis targets both industrial practitioners (e.g., facility managers) and researchers concerned with building energy efficiency, offering a novel object-oriented approach for modelling the operational aspects of the built environment. By enabling accurate, validated, and executable representations that integrate directly with building systems, this work supports the deployment of dynamic control strategies and advances the state of practice in operational building energy management.

1.2 Thesis Statement

As the building industry increasingly explores digitalisation to meet energy efficiency, interoperability, and automation goals, there is growing interest in using structured digital models to represent MEP subsystems (Mars BIM International, 2024; Nilesh Mayani, 2025; Walczyk & Ożadowicz, 2024). However, existing approaches remain constrained by the limitations outlined earlier in this chapter, which restrict model reuse and extensibility across applications. This motivates the central thesis of this work:

> **Thesis Statement**
>
> The current complexity of representing mechanical, electrical, and plumbing (MEP) systems hinders data interoperability and dynamic control deployment. Our thesis posits that an object-oriented modelling approach resolves these issues by providing consistent, standardised access to MEP data and enabling the integration of control strategies via structured class definitions and interfaces and validation mechanisms compatible with Building Management Systems (BMS).

We adopt an approach that supports semantic accuracy, dynamic behavioural representation, and deployment of control logic to create digital twins through defined interfaces. By combining structural validation, behavioural abstraction, and dynamic memory-resident models with operational building data, this work demonstrates the potential of object-oriented metamodels as a foundation for next-generation representations of building systems.

1.3 Problem Statement

To overcome interoperability limitations arising from vendor-specific representations of building systems through their BMS, practitioners and researchers rely on ontologies to provide a standardised representation of such systems. However, these ontologies are primarily for domain knowledge representation and are limited in their ability to support dynamic behaviour (Daniele, 2025), enforce validation rules, integrate operational data, and facilitate the seamless deployment of control strategies—challenges that persist in the practical applications of building models.

Semantic ontologies, such as Brick and Project Haystack, standardise building data through RDF/OWL-based representations for modelling the operational phase of buildings. However, their limitations (discussed above) reduce their effectiveness in real-world deployments, which require models that combine semantic representations with operational logic.

Consequently, there is a pressing need for a modelling approach that enforces constraints during model creation, provides semantic clarity, supports behaviour modelling, and integrates control interfaces. Such an approach would enable the deployment of intelligent, energy-aware solutions across diverse BMS, bridging the gap between semantic representation and actionable control to enable interoperable MEP system management.

1.4 Research Questions

To address the thesis statement of this thesis, we formulate the following research questions, which are explained below.

Object-oriented Modelling of MEP Systems. Mechanical, electrical, and plumbing (MEP) systems are fundamental components of buildings, playing a critical role in their operational performance. Efficient management of MEP systems can significantly improve energy efficiency and ensure optimal environmental conditions for occupants (Moghadam, Morales, Zambrano, Bruton, & O'Sullivan, 2023). Thus, accurately representing MEP entities is key to enabling energy-efficient solutions and transferring applications across buildings with heterogeneous BMSs.

To address this challenge, we conducted a comprehensive review of existing building representation systems to understand their limitations and build on their strengths. We adopted a practical approach by engaging experts through surveys and interviews and by examining complex HVAC systems in detail. This grounding enables us to propose an object-oriented modelling approach that reflects the practical needs of researchers and practitioners.

Two key research questions and subquestions guide this approach.

(1) **RQ1**: How does an object-oriented metamodel enable the creation of accurate, inherently validated models compared to existing ontology-based approaches?

- **RQ1.1**: What are the key limitations of current built environment modelling approaches, particularly ontology-based systems?
- **RQ1.2**: How can an object-oriented modelling paradigm address these limitations and offer practical advantages for representing and validating MEP systems?

(2) **RQ2**: How can an object-oriented metamodel represent mechanical, electrical, and plumbing (MEP) systems and their complex interactions?

By answering these questions, we design and implement a novel object-oriented metamodel. We evaluate the proposed metamodel through practical use cases and empirical experiments.

Interoperable Data Access for MEP Systems. Creating abstractions of MEP systems in buildings to enable a homogeneous representation of the heterogeneous semantic data provided by different BMS is crucial for deploying intelligent and energy-efficient solutions, such as control algorithms. Having designed and implemented an object-oriented metamodel to abstract the complexity of semantic representations across building systems, this thesis investigates the following question:

(3) **RQ3**: How can practitioners and researchers access interoperable data of mechanical, electrical, and plumbing (MEP) systems in buildings?

- **RQ3.1**: What capabilities do BMS APIs provide to interact with building systems and the data they generate?
- **RQ3.2**: How can practitioners and researchers interact with building systems and their data in a standardised way?

To support this investigation, we propose a five-layered architecture in which clients (researchers and practitioners) at the first layer can access and interact with building systems at the fifth layer in an interoperable manner. The third layer of this architecture encompasses APIs provided by different BMS vendors. We conduct a comprehensive evaluation of these APIs to assess their capability to support the instantiation of models and the deployment of energy-efficient solutions.

Integrating Control Logic in Object-Oriented MEP Models. Having developed an object-oriented metamodel and evaluated BMS APIs to assess their capabilities, we integrate both to deploy control strategies that enhance building operations and enable the creation of digital twins that represent the dynamic aspects of building subsystems through real-time interactions. Researchers and practitioners frequently create abstractions of reality to simulate and test solutions in controlled environments, and building models are similarly used in energy-efficiency research (Han & Kim, 2021) for tasks such as envelope retrofitting and deployment of control algorithms. In many commercial buildings, ventilation loads account for a greater share of energy consumption than envelope-related factors (Moe & Gibbs, 2023). While this thesis includes a practical use case demonstrating detailed envelope modelling at Concordia University's Webster Library, it places particular emphasis on integrating control strategies into models, given their potential to improve building energy efficiency.

RQ4 guides this section of our thesis.

(4) **RQ4**: How does an object-oriented modelling approach support the integration of control logic and promote broader adoption in practical energy-efficient applications?

Our proposed metamodel defines interfaces that researchers and practitioners can implement to embed control logic into homogeneous building representations. A control logic integration use case, along with multiple other examples, illustrates the practical applicability of our approach, especially within the context of the Webster Library.

1.5 List of Publications

This doctoral thesis led to the submission and publication of the following research papers.

Yefi, P., Ejaz, S., Menon, R. P., Eicker, U., & Guéhéneuc, Y.-G. (2024). An Architectural Approach for Enhanced Data Interoperability Across Building Systems. In *2024 7th conference on cloud and internet of things (ciot)* (p. 1-8). doi: 10.1109/CIoT63799.2024.10757021

Yefi, P., Menon, R., & Eicker, U. (2023). Building IoT systems modeling: A Object-oriented Metamodelling Approach. In *2023 ieee/acm 5th international workshop on software engineering*

research and practices for the iot (serp4iot) (p. 1-8). doi: 10.1109/SERP4IoT59158.2023.00006

Yefi, P., Menon, R., & Eicker, U. (2024). Evaluation of APIs for Data Exchange with Building Management Systems. In *Proceedings of the acm/ieee 6th international workshop on software engineering research & practices for the internet of things* (pp. 1–6).

Yefi, P., Menon, R. P., Eicker, U., & Guéhéneuc, Y.-G. (2024). Metamenth: A Object-Oriented Metamodel for IoT Systems in Buildings. *IEEE Internet of Things Journal*, 1-1. doi: 10.1109/JIOT.2024.3373330

Yefi, P., Menon, R. P., Eicker, U., & Guéhéneuc, Y.-G. (2025a). *Implementation of an Object-Oriented Metamodel for Modelling Building Systems.* (Manuscript submitted for publication)

Yefi, P., Menon, R. P., Eicker, U., & Guéhéneuc, Y.-G. (2025b). *A Study of the Capabilities of Building Management System APIs and the Limitations of Their Practical Usage.* (Manuscript submitted for publication)

1.6 Outline

We organise the remainder of this thesis as follows. Chapter 2 presents a review of existing literature and identifies key limitations in current representation systems for modelling buildings and their subsystems. Chapter 3 introduces the design principles and implementation of **MetamEnTh**, outlining its object-oriented metamodel and validation mechanisms. In Chapter 4, we evaluate BMS APIs, categorising their functionalities into twelve groups and assessing their capability to expose the necessary endpoints for seven common building applications. Chapter 5 demonstrates how **MetamEnTh** supports the seamless integration of user-implemented control logic through well-defined interfaces. Finally, Chapter 6 concludes the thesis and outlines a roadmap for future research and development.

Chapter 2

Literature Review

In this chapter, we review existing literature to identify key research gaps aligned with the research questions introduced in Chapter 1. The review is structured into three main sections, each corresponding to one or more of the research questions:

(1) Section 2.1 examines existing literature related to **RQ1** and **RQ2**, focusing on current modelling approaches for MEP systems and their limitations.

(2) Section 2.2 explores literature related to **RQ3**, with emphasis on data access and API capabilities in Building Management Systems (BMS).

(3) Section 2.3 reviews studies relevant to **RQ4**, particularly on the integration of modelling approaches with control strategies in energy-efficient applications.

2.1 Object-oriented Modelling of MEP Systems

This section explores the literature relevant to the first two research questions:

(1) **RQ1**: How does an object-oriented metamodel enable the creation of accurate, inherently validated models compared to existing ontology-based approaches?

(2) **RQ2**: How can an object-oriented metamodel represent mechanical, electrical, and plumbing (MEP) systems and their complex interactions?

Ongoing commissioning (OCx) is a continuous process that verifies and maintains building performance across a facility's life cycle. Smart and ongoing commissioning (SOCx) further leverages real-time sensor data and analytics to optimise operations under changing conditions. Within this context, ontology-based approaches have emerged as key enablers for structuring building metadata. Gilani, Quinn, and McArthur (2020) conducted a systematic review of ontologies published since 2014 and categorised SOCx data as static (physical attributes) or dynamic (time-varying measurements). They identified commonly reused classes—indoor conditions, physical building information, and BMS-related data—and three ontology development approaches: (i) new ontologies; (ii) synthesis of existing ones; and (iii) comparative analyses. Although Brick was found to be the most comprehensive for SOCx, its real-world implementations remain limited, and significant coverage and validation gaps persist.

These limitations align with the broader findings of Butzin, Golatowski, and Timmermann (2017), who reviewed semantic information models in BMS and highlighted the insufficient representation of the contextual and functional characteristics of devices. For example, BACnet deployments typically describe devices with minimal semantic information (e.g., location, description). They evaluated models such as QUDT (QUDT, 2014), Unit of Measure (Rijgersberg, Van Assem, & Top, 2013), BIM (Kensek, Karen, 2014), Green Building XML (gbXML.org, 2000), IFC (Laakso, Kiviniemi, et al., 2012), ThinkHome (Reinisch, Kofler, & Kastner, 2010), BOnSAI (Stavropoulos, Vrakas, Vlachava, & Bassiliades, 2012), Project Haystack (Haystack, 2011), SAREF (ETSI SmartM2M Technical Committee, 2015), and Brick (Balaji et al., 2016), and identified limited spatial semantics (often restricted to device location) and incomplete support for operational subsystems such as transportation and security (RQ1.1).

In response, our work introduces an object-oriented metamodel that clearly separates static and dynamic data, embeds validation within the model, and is validated across multiple real-world BMS case studies. It models logical groupings such as HVAC zones (RQ1.2), supports diverse IoT and building subsystems (RQ2), and provides complete implementation and documentation to ensure reproducibility and practical deployment.

A recurring challenge in ontology-driven modelling is the lack of built-in constraint enforcement. For example, Stolk and McGlinn (2020) showed that OWL's open-world assumption and

limited cardinality enforcement hinder the validation of ifcOWL datasets. They proposed SHACL (Simon Steyskal and Dimitris Kontokostas, 2017) to detect inconsistencies and improve IFC interoperability. Extending this, C. Wang, Zhang, and Yan (2024) used SHACL to enforce IFC's WHERE rules, previously impossible to represent in OWL. Although effective, SHACL introduces additional modelling overhead and depends on correct rule definitions. *By contrast, our metamodel enforces constraints inherently at the class and relationship level (RQ1.2), removing the need for post hoc validation.*

Beyond building ontologies, research in cyber-physical systems (CPS) also reveals gaps in expressiveness. Fitz, Theiler, and Smarsly (2019) studied modelling approaches for CPS in structural health monitoring and found that existing ontologies inadequately represent communication infrastructures connecting system components. Their proposed metamodel integrates communication-related information using EXPRESS, SWE, UML/OMG, and IFC. Complementing this structural focus, Rasmussen, Pauwels, Hviid, and Karlshoj (2017) introduced the Building Topology Ontology (BOT), a minimal yet extensible framework centred on *building*, *story*, *space*, and *element*. BOT provides a reusable topological foundation but does not inherently capture subsystem dynamics or operational behaviour. *Our metamodel advances beyond CPS and BOT by explicitly modelling MEP subsystems (HVAC, lighting, plumbing) and their operational roles, supporting performance-driven control and model validation not addressed by prior work (RQ1.2).*

Tag-based metadata approaches offer flexibility but introduce semantic ambiguity. The Haystack Tagging Ontology (HTO) (Project Haystack Organization, 2025) uses controlled vocabularies to describe heterogeneous BMS elements. While enabling interoperability, its lack of strict enforcement can lead to inconsistent modelling (RQ1.1). Likewise, Bhattacharya, Ploennigs, and Culler (2015) found that Project Haystack, IFC, and SSW insufficiently capture contextual, spatial, and functional relationships in sensor-rich buildings, limiting their suitability for complex applications.

Brick (Balaji et al., 2018) and its extension Brick+ (Fierro, Koh, Agarwal, Gupta, & Culler, 2019) address some of these issues by offering a unified, hierarchical schema for building metadata. Although Brick achieved high coverage across six commercial buildings, it inherits RDF/OWL limitations, including the absence of built-in validation, limited behavioural modelling, and

weak integration with control strategies. Synthesising these shortcomings, Pritoni et al. (2021) reviewed five prominent ontologies—BOT, SSN/SOSA (Compton et al., 2012), SAREF, RealEstateCore (Hammar, Wallin, Karlberg, & Hälleberg, 2019), and Brick—and concluded that none fully support energy-oriented tasks such as AFDD, audits, or advanced HVAC optimisation, often requiring multi-ontology integration (RQ1.1).

Our metamodel addresses these fragmentation and validation gaps by defining coherent classes and relationships for MEP systems, enabling extensibility and behavioural modelling while maintaining internal model validity (RQ1.1, RQ2).

Cross-domain frameworks such as NGSI-LD (ETSI, 2023) also inform building modelling. NGSI-LD uses a property graph approach to represent structural, functional, and administrative aspects of entities and enhances the semantic expressiveness of RDF triples (W3C RDF Working Group, 2023). However, property graphs lack standardised formal semantics, making interoperability with RDF and SPARQL (W3C SPARQL Working Group, 2013) challenging and requiring reification strategies that increase complexity. *Our object-oriented metamodel avoids these issues by using UML class diagrams to represent structure, behaviour, and constraints within a unified modelling framework (RQ1.1).*

Industry initiatives further illustrate the need for integrated semantics and validation. Google's *Digital Buildings* project (Google, 2023) proposes semantically rich models, a configuration language, and validation tools, although validation remains external to the ontology (Google, 2023). Similarly, SAREF (Bhattacharya et al., 2015; ETSI SmartM2M Technical Committee, 2015) and Eclipse Vorto (Eclipse Foundation, 2014) provide reusable abstractions and DSLs for device capabilities but rely on external tooling for consistency and offer limited support for modelling dynamic subsystem behaviour. *In contrast, our metamodel embeds validation within class structures and supports dynamic interactions and control logic (RQ1.2, RQ2), offering a more suitable foundation for modelling operational building systems.*

Because **MetamEnTh** focuses on modelling the operational phase of buildings, the remaining of our review for **RQ1** and **RQ2** concentrates on two prominent ontologies widely adopted for this purpose—Project Haystack and Brick (Balaji et al., 2018; Bhattacharya et al., 2015; Gilani et al., 2020; Google, 2023; Haystack, 2011; Pritoni et al., 2021). We examine them, highlighting key

limitations that hinder their effectiveness in representing building operations. In Chapter 3, we introduce **MetamEnTh** and demonstrate how it addresses these limitations through an integrated, object-oriented modelling approach.

2.1.1 Project Haystack and Brick

In this subsection, we highlight the limitations of prior approaches and motivate **MetamEnTh** as an alternative by situating it within the broader body of work.

The metadata approach to describing buildings adopts a bottom-up strategy, in which data labels (metadata) combine to define entities and their relationships. In this context, metadata refers to descriptive information—tags—that characterise building components, their types, and their associations. This approach forms the foundation of many widely used ontologies in the built environment domain, such as Project Haystack and Brick, which rely on flexible tagging systems to encode knowledge in a machine-readable form.

Figure 2.1 illustrates an example project from the Project Haystack Reference Implementation Guide (Project Haystack, 2025a). Each box in the figure represents an entity described using metadata tags. The first box (at the top) defines a site (building), the second describes an air handling unit (AHU), and the third defines a sensor. The `siteRef` tag in the second box links the AHU to the building, indicating that the unit is installed at the site labelled *Gaithersburg*. Similarly, the sensor is associated with the AHU through the `equipRef` tag. The `rooftop` tag further specifies that the AHU is located on the building's roof.

While this tagging-based metadata representation offers flexibility, it also introduces significant challenges. Because Project Haystack lacks built-in validation, users could mistakenly associate the sensor directly with the site rather than the AHU without triggering any errors (Figure 2.1). Similarly, including both `air` and `water` tags in the sensor's definition creates ambiguity about whether the sensor measures air temperature, water temperature, or both. Even though Project Haystack recommends compatible tag combinations, *tag compatibility* is not enforced, and the flexible tagging approach does not prevent users from creating conflicting or incomplete definitions. As a result, validating entity relationships and ensuring semantic consistency remain difficult in practice.

The Brick ontology attempts to introduce more structure by defining relationships within high-level classes, which are then inherited by child classes. For example, an `Equipment` can `hasPart` another `Equipment`. Because both a camera and an AHU are subclasses of `Equipment`, the ontology technically allows a model in which a camera `hasPart` an AHU. Although this is formally valid, it is not accurate or meaningful in real-world contexts.

To address these issues, Brick+ introduces an inference engine that automatically derives entity classes from Haystack tags. However, this engine is not fully accurate (Fierro et al., 2019), and maintaining it requires significant effort whenever users modify tags or introduce new ones. Relying on dynamic verification to compensate for the lack of explicit structure increases complexity and maintenance costs. More broadly, metadata- and ontology-driven approaches share a fundamental limitation: they lack strict, declarative rules that specify which entities can exist, what properties they must or may have, and how they can relate to one another. This absence of an enforced schema enables users to build multiple models that are formally valid but not necessarily accurate. Other existing metamodels, such as Eclipse Vorto, tend to remain generic and high-level, thereby increasing the risk of inconsistent representations.

In the following paragraphs, we highlight eight limitations of Project Haystack and Brick.

L_1**: Possible Ambiguity in Defining an Entity.** The flexibility to freely assemble tags when defining entities can lead to ambiguity if users select conflicting tags. As illustrated in Figure 2.1, combining incompatible tags creates uncertainty about what a sensor measures. While the Haystack ontology specifies *supertypes*, *subtypes*, and recommended tag combinations for concepts such as *point* (Haystack, 2025), it does not enforce any constraints on which other tags users can combine when defining an entity. This absence of restrictions can lead to conflicting or unclear definitions. Moreover, the ability to introduce custom tags to extend the vocabulary further increases the risk of ambiguity and inconsistent interpretations across implementations, which ultimately undermines the notion of standardisation that ontologies are intended to promote.

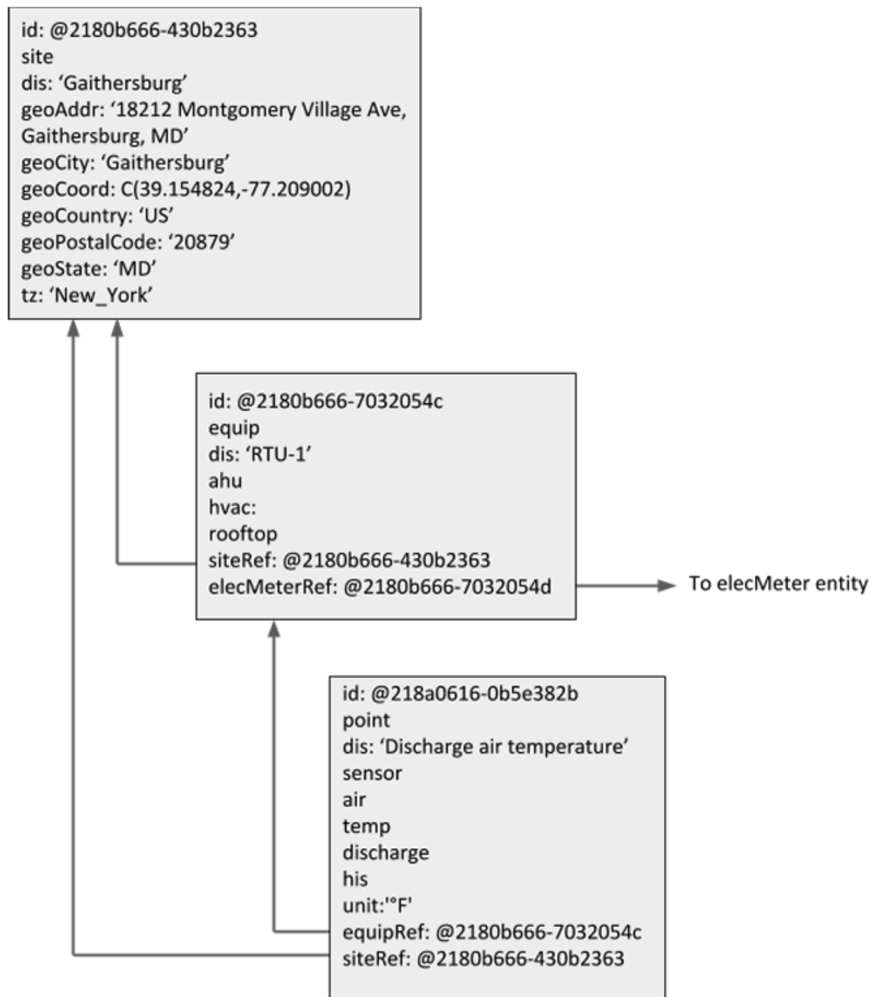
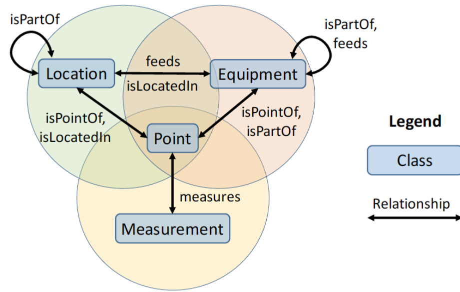
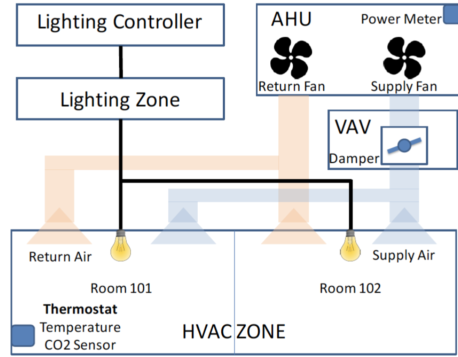


Figure 2.1: A Project Haystack sample project definition for a building in Gaithersburg
Source: (Project Haystack, 2025a)

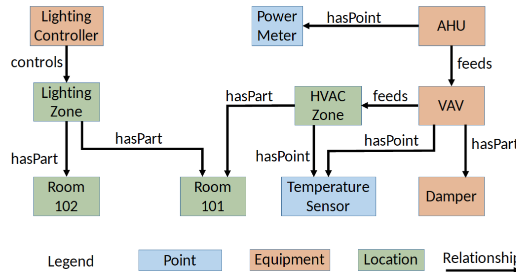
L_2**: Possibility of Composing Flawed Entity Relationships.** The information concepts and their relationships to data points in Brick (Balaji et al., 2016) are illustrated in Figure 2.2a. The relationship constraints defined by Brick specify that an `Equipment` can be part of either a `Location` or another `Equipment` (Fierro et al., 2019). Because these constraints are established at the top level of the class hierarchy (shown in Figure 2.2d), all subclasses inherit them without additional contextual refinement. For example, both `Lighting System` and `AHU` are subclasses of `Equipment`, which means that Brick allows a `Lighting System` to be part of an `AHU` and vice versa. In



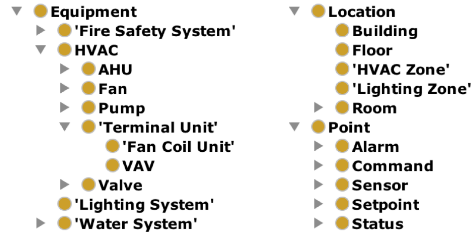
(a) Information concepts and their relationship to a point



(b) A simple example building highlighting some components



(c) An example brick model of the building in b)



(d) A subset of brick class hierarchy

Figure 2.2: Brick information concepts and representations
Source: (Balaji et al., 2016)

practice, this bidirectional relationship is inaccurate, as a lighting system cannot logically be a component of an air handling unit.

These examples illustrate how overly general relationship constraints in the ontology can lead to formally valid but inaccurate compositions of entities and relationships. Such ambiguities complicate automated validation and reduce the reliability of generated models.

L_3**: Lack of Entities for Ducts, Pipes, and Detailed Spatial Context.** Ventilation ducts and pipes are essential components of HVAC systems because they contain entities such as dampers, filters, and heat exchangers that regulate substance flow. These ducts often include sensors that measure phenomena like temperature, pressure, and carbon dioxide concentration. Additionally, entities within ventilation ducts connect to other equipment, such as chillers, cooling towers, and

boilers, that form part of the HVAC system.

Figure 2.2b shows an example where a `VAV` (variable air volume) box sits inside a ventilation duct supplying air to the `HVAC ZONE`. The `VAV` box controls the volume of air flowing through the duct and may incorporate auxiliary heaters to warm the air. `VAV` boxes installed at different positions within a duct can have distinct damper configurations and setpoints to regulate indoor environmental quality. For example, two `VAV` boxes with different configurations could be installed to serve Rooms 101 and 102, delivering distinct temperatures and airflow, even though both are connected to the same `AHU`.

In Brick, these `VAV` boxes can be associated with high-level locations, such as `Floor`, `HVAC Zone`, `Lighting Zone`, or `Room`. However, modelling their location within the ventilation system remains a challenge. Brick lacks dedicated entities to represent ducts and pipes as first-class objects, making it difficult to accurately associate internal components, such as dampers and sensors, with their specific spatial context. For example, distinguishing between sensors that measure supply air and those that measure return air becomes problematic without explicit duct entities.

The possibility of combining multiple ontologies, e.g., Brick with RealEstateCore, Brick with TUBES (Pauen, Schlütter, Siwiecki, Frisch, & van Treeck, 2020), to further enhance coverage complicates validation, as heterogeneous update cycles, overlapping vocabularies, and differing modelling conventions introduce inconsistencies that are difficult to detect. Moreover, current validation approaches provide limited support for continuous or dynamic validation during model construction, leaving users to manually verify correctness as models evolve.

L_4**: Difficulty Validating Entities and Relationships.** Validating entities and their relationships in metadata-driven approaches is inherently challenging because users define both by assembling tags without a strictly enforced schema. As explained earlier, for example, an `AHU` can be easily associated with a `Camera` via a `hasPart` relationship, with the `AHU` as the subject. Such configurations are syntactically permitted but semantically invalid. Similarly, when conflicting tags are combined to define an entity, it becomes difficult to determine the intended meaning and validate whether the definition is correct.

This difficulty stems from the lack of a structure that prescribes which entities are valid, which

relationships they can form, and how their properties should be composed. Brick+ attempts to mitigate these issues by modelling concepts as formal combinations of their properties and deducing Brick classes from Haystack tags (Fierro et al., 2019). However, this approach relies heavily on the accuracy of the inference engine. Whenever new tags are introduced or existing tags are updated, the inference engine must be retrained or reconfigured, making this a costly and labour-intensive method of introducing structure.

More broadly, ontologies based on RDF, RDFS, and OWL lack built-in validation mechanisms that can enforce completeness, consistency, and correctness. Although OWL supports some structural constraints, its reliance on the open-world assumption and the absence of the unique name assumption make it difficult to guarantee that a model fully and uniquely represents the domain (Stolk & McGlinn, 2020). While SHACL provides mechanisms for validating RDF graphs against a set of constraints, this requires maintaining a separate validation model, which increases implementation complexity and leaves open the possibility that users can bypass or neglect validation altogether.

L_5: **Inconsistent Data and Structure Semantics.** Data semantics are essential for enabling stakeholders to interpret and relate digital representations to the physical and logical aspects of a building. In Project Haystack, for example, a `Zone` is defined as a kind of `Space` (Project Haystack, 2025e). According to the Haystack documentation, a `Space` represents a three-dimensional volume within a building. However, a `Zone` is more accurately understood as a logical grouping of spatial entities—such as rooms and corridors—that share common characteristics, like heating or cooling requirements.

As illustrated in Figure 2.3 from the Siemens BMS of Concordia University's Engineering and Visual Arts (EV) Building, the thermal zone labelled *VC2-010-186* is not itself a three-dimensional volume but rather a logical division on the tenth floor, grouping spatial entities to deliver consistent indoor environmental conditions. Semantically, treating a zone as a subtype of space conflates logical and physical concepts, suggesting that rooms, floors, and zones are all equivalent three-dimensional volumes. This *design* misalignment undermines the model's clarity and interpretability for practitioners who need to distinguish between spatial extents and functional groupings.

Similarly, *by design*, the Brick ontology exhibits semantic inconsistencies in its class hierarchy,

shown in Figure 2.2d. Brick defines `Sensor`, `Command`, and `Setpoint` as subclasses of the parent class `Point`, which it broadly describes as any physical or virtual entity that generates time series data (Balaji et al., 2018). However, this grouping obscures important distinctions: a `Sensor` is a physical device that measures phenomena such as temperature or humidity. In contrast, a `Command` represents a control signal that actively alters the state of equipment (Fierro, 2019). Placing both under the same parent class implies structural and functional equivalence where none exists, weakening the semantic precision necessary for consistent modelling across applications.

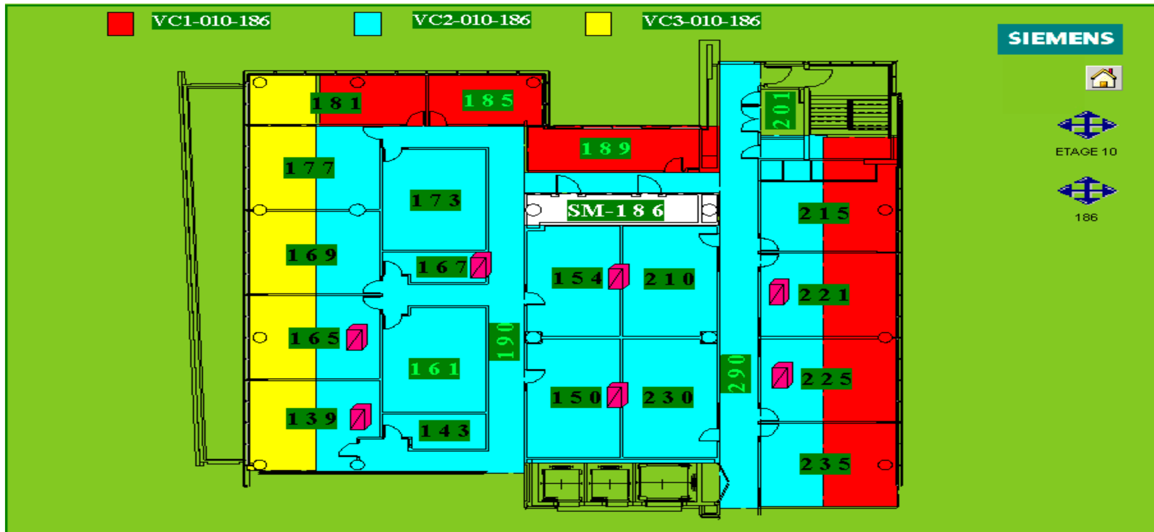


Figure 2.3: Thermal zones of the tenth floor of Concordia University's EV buildings as captured by the installed BMS

L_6**: Complex and Deep Inheritance Hierarchies.** In Brick, tagsets define the valid types used to classify entities. These tagsets are organised into a structured class hierarchy that specifies both subclass relationships and inheritance patterns across entity types (Fierro, 2019). As illustrated in the Brick documentation and shown in Figure 2.4, the class `Return_Air_C02_Sensor` is a subclass of `CO2_Sensor`, which in turn inherits from `Air_Quality_Sensor`. This hierarchy continues up to the `Sensor` and ultimately the `Point` classes.

This type-driven approach to classification builds specificity through increasingly granular subclassing. Consequently, by design, more specific entities reside deeper within the hierarchy, creating extended inheritance chains that capture increasingly specialised characteristics. Furthermore,

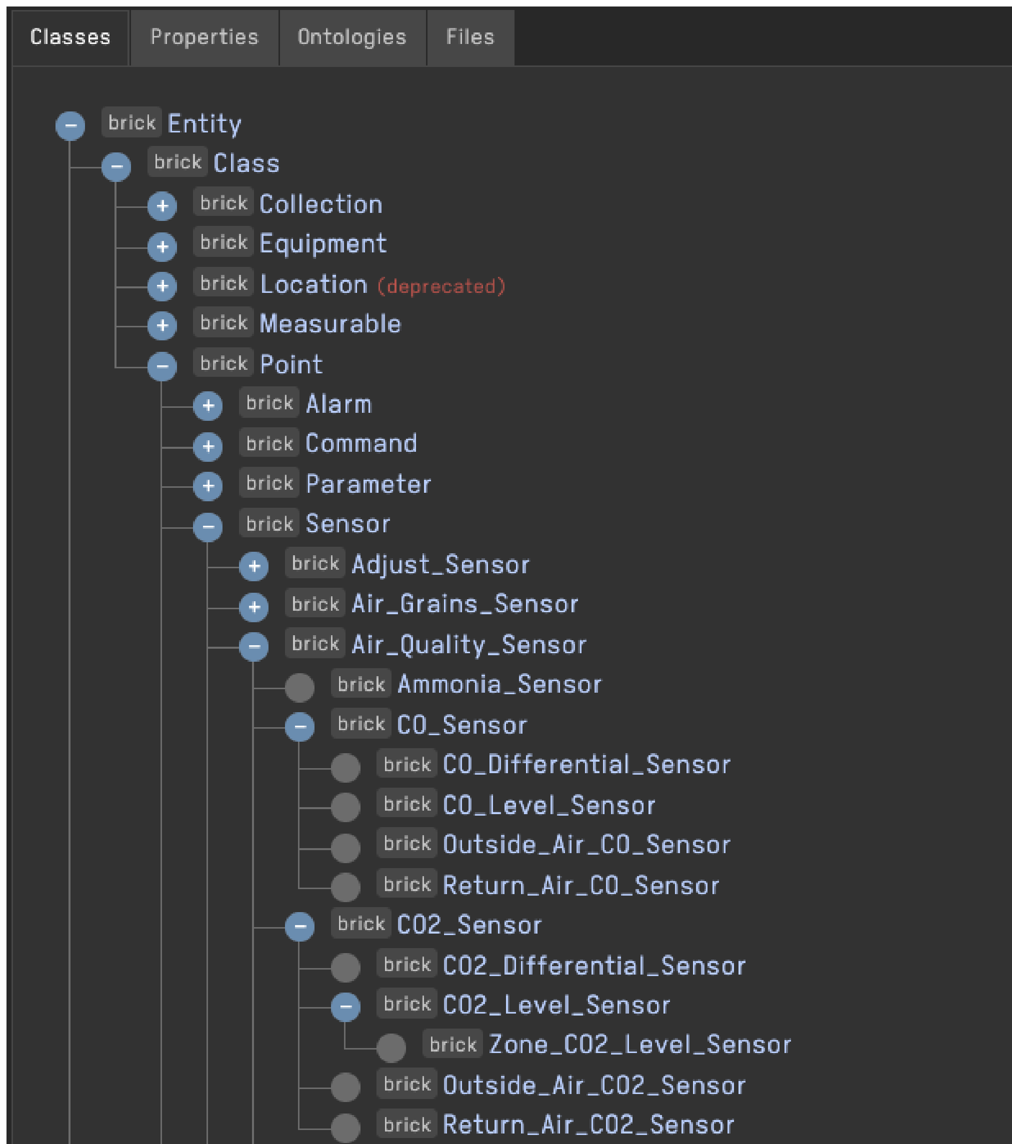


Figure 2.4: The Brick class hierarchy illustrating the five levels of inheritance leading to the *Return_Air_CO2_Sensor* class

Brick supports multiple inheritance, allowing a class to derive from more than one parent. While this provides flexibility, it also introduces complexity and potential ambiguity, particularly when entity types are defined based on what they measure (e.g., temperature, pressure, carbon dioxide concentration, or discharge).

Long inheritance paths can obscure an entity's intended semantics, and small changes to upper-level classes or tag combinations may propagate unintended consequences throughout the hierarchy.

As the number of tags and entity types increases, so too does the risk of inconsistencies and misclassification, which is further exacerbated when users introduce new, domain-specific tags.

Ultimately, while Brick's inheritance model aims to introduce structure and reuse into sensor and equipment definitions, the reliance on deep, sometimes ambiguous hierarchies complicates interpretation, maintenance, and automated reasoning, especially for complex MEP systems.

L_7: **Inability to Represent System Behaviours.** RDF-based ontologies primarily focus on representing static knowledge—descriptions of entities, their attributes, and their relationships. While this is valuable for building a shared understanding of system structure, it fundamentally limits the ability to capture or encode system behaviours (Gil, Oakes, Gomes, Frasheri, & Larsen, 2024). As a result, ontologies struggle to support operational logic or to be used in true digital twins capable of representing and driving real-time system behaviour.

This limitation means that tasks such as comparing sensor readings to control setpoints, triggering actuator responses, or executing control sequences cannot be natively defined within an RDF or OWL representation. Instead, such logic must be implemented externally in separate tools or applications, which increases integration complexity and reduces consistency between the data model and control processes.

L_8: **Static Artefacts and Limited Interactivity.** Ontology-based models typically produce static artefacts, such as RDF or OWL files, which serve as snapshots of domain knowledge. These files provide a standardised, machine-readable description of entities and their relationships, but have limitations. Static RDF/OWL representations have limited interactivity. They primarily function as data repositories rather than as live models that can actively participate in system operations. While they can be queried or parsed, they do not inherently expose programmatic interfaces for executing dynamic behaviours or interacting with real-time system states. Any integration with operational processes requires external applications to read, interpret, and act upon the data, adding complexity and increasing the risk of inconsistencies between the model and the system it describes.

Ontologies describe the properties of a subject area and their relationships by defining a set of

categories that represent the subject (Gray & Rumpe, 2022). They serve well for domain knowledge representation, offering flexibility to model complex entity relationships and adapt to varying system configurations. However, their inherent static nature limits their suitability for modelling the dynamic aspects of building subsystems and their interactions (Daniele, 2025). They generally lack built-in constraint enforcement and behavioural semantics, making it difficult to represent real-time state changes, control logic, or temporal dependencies.

Motivated by these limitations, this thesis adopts an alternative object-oriented metamodelling approach rather than extending an existing ontology. This approach avoids the structural and behavioural constraints of ontology-based approaches while still leveraging their strengths, including clear entity definitions and shared domain vocabularies. In this regard, **MetamEnTh** is inspired by Brick and reuses a substantial portion of its class definitions to ensure conceptual continuity and familiarity. However, by grounding these concepts in object-oriented modelling principles, **MetamEnTh** mitigates Brick's limitations by enabling built-in validation, explicit behavioural representation, and executable control logic, thereby providing a more suitable foundation for modelling the operational aspects of MEP systems in buildings.

2.2 Interoperable Data Access for MEP Systems

This section reviews literature related to **RQ3**: How can practitioners and researchers access interoperable data of mechanical, electrical, and plumbing (MEP) systems in buildings?

Our review of the literature on BMS and ontology-related APIs, particularly their role in enabling integration with external systems, reveals a significant gap. While many studies explore aspects of building energy management, semantic ontologies, or individual BMS architectures, few directly evaluate or compare the functional capabilities of the APIs offered by commercial BMS platforms. Moreover, we found no prior research assessing how semantic models, such as Brick or Project Haystack, represent and interact with real-time BMS data when accessed via these APIs.

A substantial body of work addresses interoperability challenges in building systems through semantic modelling and ontology alignment. Early contributions such as the Alignment API by Euzenat (2004) and the ontology extraction approach by Ratiu, Feilkas, and Jurjens (2008) focus on

resolving heterogeneity across conceptual schemas by discovering structural and naming correspondences between ontologies or API descriptions. These techniques support semantic harmonisation, but primarily target static schemas rather than live building data.

More domain-focused ontological efforts extend these principles to energy and building management contexts. Schachinger, Kastner, and Gaida (2016) developed an OWL 2 ontology to facilitate communication between smart grids and BMS, following the Ontology Development 101 methodology, while Horridge and Bechhofer (2011) provided the OWL API, an extensible software framework for programmatic creation, manipulation, and reasoning over OWL ontologies. Both support structured interoperability across heterogeneous systems but operate at the schema level.

Recent work extends semantic integration into operational BMS environments. Kučera and Pitner (2018) expanded the SSN ontology to integrate automation data with BIM and facility management systems through a semantic middleware layer that supports complex cross-system queries. Similarly, Bourreau et al. (2019) embedded the OntoH2G ontology within the BEMServer platform to provide a semantic abstraction layer for heterogeneous time-series and contextual data. These systems illustrate the value of semantic standardisation for improving data accessibility and interoperability (RQ3.2), though most remain limited in scope, focusing on benchmarking, smart grids, or static modelling rather than enabling real-time, dynamic interactions required for operational building management.

Complementing semantic approaches, a parallel stream of research explores APIs, Web services, and middleware as mechanisms for integrating heterogeneous building systems. Anthony Jnr, Abbas Petersen, Ahlers, and Krogstie (2020) proposed a multi-layered API architecture for managing diverse, high-velocity energy data streams, using HTTP and MQTT protocols to enable flexible exchange. Although not evaluated with real-world deployments, the architecture highlights requirements for scalable, interoperable data flows.

At the building level, Jianbo, Yuzhe, and Guochang (2011) demonstrated how BMS subsystems can be exposed through Web services by wrapping BACnet objects (Bill Swan, 2022) in a service-oriented architecture. This provides real-time access to building data and illustrates how standardised service interfaces can improve interoperability (RQ3.1). Extending this line, Bandara, Yashiro, Koshizuka, and Sakamura (2016) identified the lack of standardised IoT API specifications

as a major barrier to developing building applications and proposed device and context APIs that distinguish between static, dynamic, and spatial attributes of devices. Their deployment on more than 300 devices demonstrates the practical value of well-documented, standardised APIs (RQ3.2).

API-driven integration also appears in broader data platforms. Dave, Buda, Nurminen, and Främling (2018) presented a BIM-IoT integration framework that exposes structured building data to third-party applications, enabling services such as space reservation systems. However, they highlight the persistent issue of data heterogeneity across BMS. Finally, Marinakis, Doukas, Karakosta, and Psarras (2013) illustrated how simulation-based optimisation platforms can interface with building automation systems, though their tightly coupled architecture underscores the need for open, standardised interfaces to improve portability and reuse across buildings.

Efforts to integrate BIM with IoT data provide a complementary perspective on interoperability. The review by Tang, Shelden, Eastman, Pishdad-Bozorgi, and Gao (2019) synthesises common strategies, including direct APIs to BIM tools, relational databases for real-time ingestion, and hybrid RDF-relational approaches that separate static and time-series data. While effective, these solutions often produce fragmented architectures requiring significant integration effort, reflecting challenges central to RQ3.2.

Focusing on operational workflows, Quinn et al. (2020) demonstrated two methods for linking IoT data with FM-BIM: direct name-based mappings and ontology-based alignments using Dynamo. Although the ontology-based method offers richer semantic depth, the authors recommend the direct mapping approach due to its lower complexity—illustrating the ongoing trade-offs between semantic expressivity and practical implementability in building data integration.

Overall, research on API-driven integration, semantic modelling, and BIM-IoT interoperability reveals significant progress but remains fragmented. API-based systems (Anthony Jnr et al., 2020; Jianbo et al., 2011) highlight pathways for exposing building data but are often conceptual or limited in scope. Semantic approaches (Bourreau et al., 2019; Kučera & Pitner, 2018) advance interoperability through structured models yet typically address narrow domains and do not fully engage with real-time operational needs. Schema alignment methods (Euzenat, 2004; Ratiu et al., 2008) provide foundational tools but stop short of addressing live BMS data streams. Persistent barriers, especially cross-vendor heterogeneity and the lack of standardised API descriptions (Bandara

et al., 2016), continue to limit scalable, generalisable integration across building systems, directly motivating the need addressed in RQ3.

This thesis directly addresses these limitations by providing a foundational and empirical evaluation of how BMS APIs expose system data, with a focus on mechanical, electrical, and plumbing (MEP) systems. It identifies what functional capabilities these APIs currently support (RQ3.1) and critically examines the extent to which these capabilities enable standardised, semantic interaction with building data (RQ3.2).

2.3 Integrating Control Logic in Object-Oriented MEP Models

This section addresses literature related to **RQ4**: How does an object-oriented modelling approach support the integration of control logic and promote broader adoption in practical energy-efficient applications?

A broad body of research has sought to improve interoperability, data integration, and operational intelligence in the built environment through ontologies, middleware platforms, and digital-twin frameworks. For example, Alshammari, Beach, Rezgui, and Alelwani (2023) developed an ontology-based framework for access management that integrates IoT devices, cyber-physical systems, BIM, and digital twins. Their ontology formalises roles, permissions, and resource hierarchies to support automated reasoning and policy enforcement. While this advances semantic interoperability, its focus remains on access-control semantics rather than on modelling building-system behaviours or enabling executable control logic.

Complementing this semantic perspective, several middleware-oriented efforts address fragmentation in building management systems. BOSS, introduced by Dawson-Haggerty et al. (2013), provides standardised interfaces for device interaction, control, data access, and authorisation through components such as a Hardware Abstraction Layer and a Control Transaction Manager. Similarly, OpenBMS (McGibney, Rea, & Ploennigs, 2016) employs IoT technologies to reduce vendor lock-in and streamline interoperability, demonstrated across multiple pilot sites. These platforms enable more uniform access to heterogeneous systems, but they operate primarily at the interface level and do not model spatial or system hierarchies. As such, they rely on external abstractions rather than

offering integrated representations of building structure, behaviour, and control.

Parallel work in digital construction and building ontologies provides richer semantic models of building entities, workflows, and domain knowledge. The Digital Construction Ontologies (DiCon) by Zheng, Törmä, and Seppänen (2021), underpinned by the Basic Formal Ontology (Alan Ruttenberg, 2002), standardise information across construction processes via six interconnected modules. Likewise, CTRLont (Schneider, Pauwels, & Steiger, 2017) extends ontology-based modelling to represent the semantics of control logic in building automation systems. Although these efforts introduce more comprehensive knowledge representations, their reliance on static RDF serialisations (e.g., `.ttl` outputs) limits the ability to represent runtime behaviours or to interact directly with operational control systems.

Digital-twin and CPS-oriented research further advances real-time data utilisation. Lu et al. (2020) proposed a digital twin integrating BIM, IoT, and energy simulation to support performance monitoring. In contrast, Zhang, Kang, Lin, Zhang, and Zhang (2020) introduced a scalable BIM–CPS platform employing sensing, cloud-based storage, RESTful services, and advanced analytics. These systems effectively combine static and dynamic data for monitoring, forecasting, and decision support. However, in most cases, behavioural models remain tied to simulation tools or data-analysis pipelines, and the frameworks lack mechanisms to embed and execute real-time control logic via BMS APIs.

Collectively, the literature reveals three recurring limitations:

- Ontology-based systems offer strong semantic structure but limited support for dynamic behaviours or executable control logic.
- Middleware and interoperability platforms unify access to heterogeneous systems but do not represent building hierarchies or control behaviours within a modelling framework.
- Digital twins and CPS platforms emphasise data acquisition and analytics, but often remain decoupled from real-time actuation or behavioural modelling.

In response to RQ4, our research addresses these gaps through a metamodel-driven, object-oriented approach that embeds control logic directly into memory-resident representations of building systems. Unlike ontology-based or middleware-driven approaches, our metamodel captures

structural, behavioural, and interactive aspects of buildings while supporting real-time reasoning and actuation through BMS APIs. This enables practitioners to move beyond passive monitoring and analytics toward active, automated building management grounded in an extensible and computationally executable modelling foundation.

Table 2.1 presents a summary of the reviewed literature, mapping our four main research questions to key references and the gaps they reveal. A comprehensive version is provided in Table A.1 in the Appendix, which details all reviewed references, their associated gaps, and corresponding research question mappings.

Table 2.1: Mapping of research questions to gaps and relevant studies: Summary

RQ	**Main Focus**	**Identified Gap**	**Relevant Studies**
RQ1	Creating accurate and validated models with an object-oriented modelling approach	Lack of built-in validation mechanism. Post-hoc validation allows inaccurate models, and it is difficult to maintain	Stolk and McGlinn (2020), C. Wang et al. (2024), Bhattacharya et al. (2015), Balaji et al. (2018), Fierro et al. (2019), Pritoni et al. (2021)
RQ2	Modelling MEP entities with an object-oriented metamodel	Approaches limited to the use of ontologies. Existing metamodels and high-level, and they do not focus on the operational aspects of buildings	Butzin et al. (2017), Bhattacharya et al. (2015), Balaji et al. (2018), Pritoni et al. (2021), ETSI (2023)
RQ3	Accessing interoperable data of building systems	Absence of research on the evaluation of BMS APIs capabilities for interoperable data access through standardised models	Anthony Jnr et al. (2020), Euzenat (2004), Ratiu et al. (2008), Schachinger et al. (2016), Horridge and Bechhofer (2011), Jianbo et al. (2011)
RQ4	Integration of control logic through object-oriented modelling approach	Existing approaches lack behavioural expressiveness, and interfaces for real-time control, or actuation logic integration compared to runtime metamodels	Alshammari et al. (2023), Dawson-Haggerty et al. (2013), McGibney et al. (2016), Zheng et al. (2021), Lu et al. (2020), Zhang et al. (2020), Schneider et al. (2017)

Chapter 3

Object-oriented Modelling of MEP Systems

This chapter presents the first set of core contributions of the thesis by addressing **RQ1** and **RQ2**. The following sections address these research questions, providing an overview of the research method, results, discussion, limitations, future work, and a concluding summary.

3.1 Introduction

Researchers and practitioners have widely adopted ontologies for formal knowledge representation across multiple domains. Ontologies provide a shared repository of well-defined vocabularies and the relationships among them, fostering a common understanding among diverse stakeholders. Their flexibility lies in their capacity to express domain knowledge through subject–predicate–object triples, enabling consistent, machine-interpretable descriptions of entities, attributes, and interconnections. In Chapter 2, Section 2.1, we reviewed existing literature on the use of ontologies for modelling building subsystems and highlighted their limitations, thereby answering **RQ1.1**. In this section, we introduce an object-oriented modelling approach, the Metamodel for Energy Things (**MetamEnTh**), to address these limitations and enable richer, more structured representations of MEP systems and their operational logic. This discussion leads us to the two research questions addressed in this section:

- **RQ1**: How does an object-oriented metamodel enable the creation of accurate, inherently validated models compared to existing ontology-based approaches?
  - **RQ1.1**: What are the key limitations of current built environment modelling approaches, particularly ontology-based systems? (answered in Chapter 2, Section 2.1)
  - **RQ1.2**: How can an object-oriented modelling paradigm address these limitations and offer practical advantages for representing and validating MEP systems?
- **RQ2**: How can an object-oriented metamodel represent mechanical, electrical, and plumbing (MEP) systems and their complex interactions?

To contextualise our approach, we first clarify the distinctions among an **ontology**, a **metamodel**, and a **model**, highlighting how each concept underpins different aspects of representation and implementation.

3.1.1 Metamodel, Model, and Ontology

A model is an illustration of a specific aspect of reality, tailored for a particular intention, and can represent diverse types of realities (Yonglin, Zhi, & Qun, 2020). It is a specification of a system and its environment for some specific purpose (Henderson-Sellers, 2011). On the contrary, a metamodel is a model that represents other models, and every model must have a corresponding metamodel (Paige, Kolovos, & Polack, 2014; Thomas, 2003). It is a "model of a set of models" (Favre, 2005). A model that aligns with the metamodel represents truth, thereby establishing metamodels as models with guiding principles (Yonglin et al., 2020). The expressiveness of a metamodel determines what an instantiated model can describe. Ontology and metamodels are closely related yet independent concepts. Ontologies are formal constructions agreed upon by domain experts (Henderson-Sellers, 2011). They are maps that show different ideas and how they connect, while a metamodel is a map that shows concepts and connections and instructs how to arrange them (Yonglin et al., 2020). Metamodels define semantics for modelling languages to express models within a specific domain.

3.2 Approach

To address the research questions, this study adopted a multi-method approach combining grounded theory, iterative design, and engineering research principles. The literature review, together with an evaluation of existing ontologies and metamodels, as well as empirical engagements with researchers and facility managers, provided a comprehensive understanding of the current state of the art—highlighting both the strengths and limitations of existing approaches. Building on these insights, we iteratively designed and refined **MetamEnTh** through a feedback loop involving practitioners and researchers across multiple case studies. This process informed the development of **MetamEnTh** to integrate the strengths of ontology-based modelling while incorporating practitioner feedback and mitigating identified limitations. Below, we summarise the steps undertaken to address the research questions, excluding the literature review presented in Chapter 2.

- Examined existing built environment representation systems to understand how they model buildings, what entities they encompass, and what their strengths and shortcomings are.
- Conducted a needs assessment with practitioners and researchers to gather requirements for an alternative object-oriented metamodel.
- Toured the HVAC system of Concordia University's buildings to develop a deeper understanding of the different entities and how they connect and function as a complete system.
- Designed an initial UML-based metamodel that synthesised findings from the needs assessment, HVAC site visit, and the review of existing ontologies
- Conducted case studies using data from three buildings and iteratively refined the UML-based metamodel based on feedback and insights obtained from these studies.
- Implemented the metamodel programmatically in Python.
- Developed and documented practical unit tests to ensure correctness and demonstrate how the implementation can be applied in practice.
- Evaluated the applicability of the implementation through four use cases, including Concordia's EV Building, LB Building, MB/JMSB Building, and the Varennes Library.

- Conducted an empirical experiment to compare the metamodel with the Brick Ontology in terms of expressiveness, usability, and adaptability.

3.2.1 Needs Assessment

After identifying the limitations of existing built environment representation systems, we conducted a targeted needs assessment with practitioners and researchers. This assessment was informed by both our review of existing ontologies (see Chapter 2) and the specific challenges uncovered in the preceding analysis. The purpose of the needs assessment was to ensure that any new representation system would not only mitigate the identified shortcomings but also meet the practical requirements of the professionals who rely on such models in real-world settings. We contacted prospective respondents via email and LinkedIn, resulting in 12 researchers and practitioners participating. We developed our questions based on insights gained from the literature review. A domain expert reviewed these to ensure clarity and relevance.

To systematically capture stakeholder needs, we conducted both interviews and surveys. We conducted in-depth interviews with two experienced facility managers (FMs), who provided insights into operational workflows, pain points, and expectations for modelling tools. In addition, we distributed online surveys to researchers and industry practitioners, and received 10 responses.

3.2.1.1 Interviews

The primary objective of the interviews was to obtain an in-depth understanding of BMS and identify the critical components necessary to advance research on building energy efficiency. Each interview was conducted individually and lasted approximately 30–45 minutes, allowing for detailed exploration of participants' experiences and perspectives.

Both participants are professionals based in Canada, with 18 and 16 years of experience, managing government, institutional, and public commercial buildings. They also have backgrounds in mechanical and civil engineering. The following questions guided our needs assessment:

- **Question 1**: If you were creating building models for simulation to improve energy efficiency, which building systems would you prioritise?

- **Question 2**: Based on your experience, is it acceptable to make minor errors in building models intended for energy simulation?
- **Question 3**: Is it important for a user to determine which time-series (sensor) data to include in a model, and the required data duration?
- **Question 4**: Could you explain why modelling ventilation ducts, the components within them (e.g., fans, dampers), and their connections (e.g., to chillers) is important?
- **Question 5**: Have you worked with BMS data outside the BMS itself? If so, what data did you use, for what purpose, and in what format?

We summarise the insights from the interviews below:

- **Modelling Priorities**: The two FMs emphasised the importance of modelling fresh air systems (ventilation ducts and pipes) and mass flow of air and water to accurately represent components necessary for calculating conductive and convective heat losses.
- **Tolerance for Modelling Errors**: While minor modelling errors may be acceptable, the FMs agree that models should avoid inaccuracies in the configuration of ventilation ducts or piping systems, as such errors can propagate through analyses and lead to incorrect conclusions about system performance. One FM noted that BMS visualisations can be misleading, and such modelling errors can significantly distort energy analysis. Expert validation is recommended for airflow-related modelling.
- **Data Inclusion and Scope**: Extensive data can increase model size and detail, potentially complicating interpretation and analysis. Setting thresholds—such as specifying the types and amount of sensor data or selecting systems based on building size and modelling objectives—helps maintain focus and ensures the model remains manageable and relevant.
- **Use of BMS Data**: Trend log data from BMS are often exported in heterogeneous formats, such as CSV, XML, proprietary .dat files, or even via email. Facility managers typically import these logs into Excel or other analytical tools to perform calculations on energy flux,

entropy changes in heating systems, or power consumption in electrical systems. This workflow is error-prone, time-consuming, and limits the ability to efficiently analyse and act on building performance data.

3.2.1.2 Survey

We conducted a survey to understand the needs of both researchers and practitioners regarding an object-oriented approach to modelling the building subsystems. In total, we received 10 responses: six from student researchers, two from facility managers, and the remaining two from a professor and an industrial researcher. To provide transparency regarding the respondents' backgrounds, we report basic demographic information, including professional roles, years of experience, and countries in which respondents have worked. Collectively, the respondents have worked in thirteen countries, including Canada, Lebanon, Iraq, Turkey, Syria, the United States, Australia, Chile, Colombia, Ecuador, Germany, India, Mexico, and Spain. Figure 3.1 presents additional details about the demographic distribution of the respondents. Given the exploratory nature of the study and the limited sample size, no statistical analysis or substantive conclusions are drawn from the demographic data.

Figure 3.2 illustrates the respondents' familiarity with multiple built environment modelling tools. The respondents employ these tools for a range of purposes, including energy and operations simulations, occupant comfort analysis, thermal and energy modelling, and energy demand forecasting, particularly in the context of building management and retrofitting.

The respondents answered a series of questions using a 5-point Likert scale. Below are the survey questions (SQ) posed to the participants:

- SQ1: In your experience, how difficult is it to create building models from existing ontologies and metamodels?
- SQ2: How important is it for a building model to accurately represent the relationships between building systems and entities relevant to your research or work?
- SQ3: How important is it for a metamodel to model the behaviour of building entities?

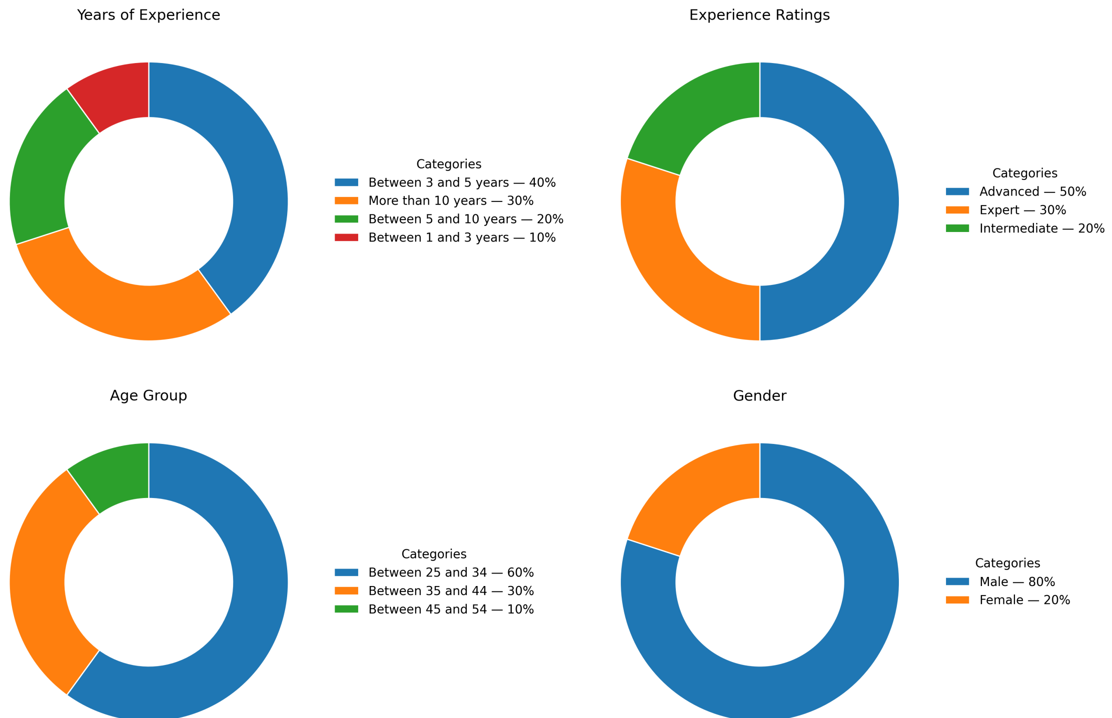


Figure 3.1: **Demographic information of the survey respondents.** The figure shows that the majority of respondents have more than three years of experience, with 50% having over five years of experience. Additionally, 50% of respondents identify as having advanced expertise in their respective fields.

- SQ4: How important is it for a building model to distinguish between static data (e.g., sensor definitions or building envelope) and dynamic data (e.g., time-series data such as temperature recorded every 15 minutes)?
- SQ5: How important is it to specify the amount of historical sensor data included in a building model (e.g., three months of occupancy data)?
- SQ6: How important is it for a metamodel or ontology to explicitly model ductwork and its associated systems (e.g., ventilation ducts, connected spaces, and air handling units)?
- SQ7: How important is semantic organisation (e.g., grouping rooms and floors into spaces and relating them to zones) for the comprehension and usability of a building model?

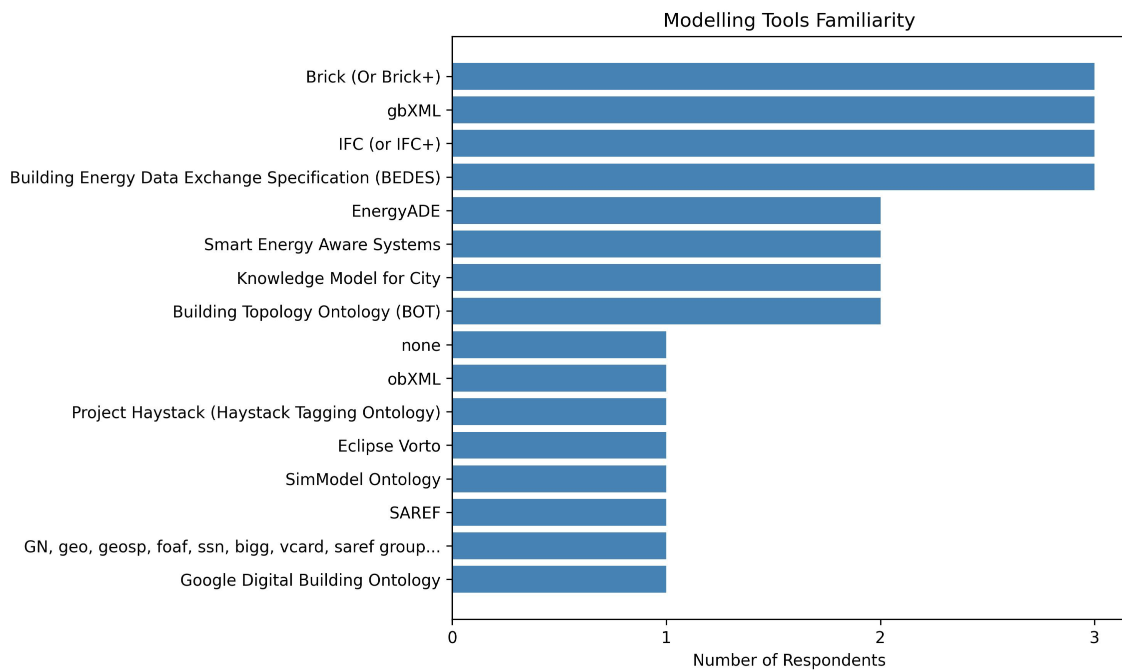


Figure 3.2: **Familiarity of respondents with built environment modelling tools.** The figure indicates that respondents have experience with a range of built environment modelling tools, with four respondents reporting familiarity with Brick, gbXML, IFC, and BEDES.

Figure 3.3 displays the distribution of the most frequently selected responses, highlighting the central tendencies across the survey questions (SQs). For SQ2, SQ4, and SQ7, **50% of respondents assigned a rating of 5**, indicating strong agreement on the importance of these proposed modelling aspects. **In contrast**, the modal response for SQ1 was a rating of 3 (60% of respondents), while the modal response for SQ5 was a rating of 4 (60% of respondents). Crucially, **no question received a rating below 3**, indicating that all proposed modelling aspects were perceived as, at minimum, moderately relevant, important, or difficult.

Given that **MetamEnTh** focuses on modelling the operational aspects of building systems towards energy efficiency, the study sought to identify the specific priorities of researchers and practitioners. Therefore, Survey Question 8 (SQ8) directly asked: "Which building systems (and entities) are crucial for your energy efficiency research or work?" Figure 3.4 presents the responses.

From Figure 3.4, the survey results highlight the built environment entities considered essential for energy efficiency. Eight respondents identified the building envelope as a key entity, followed

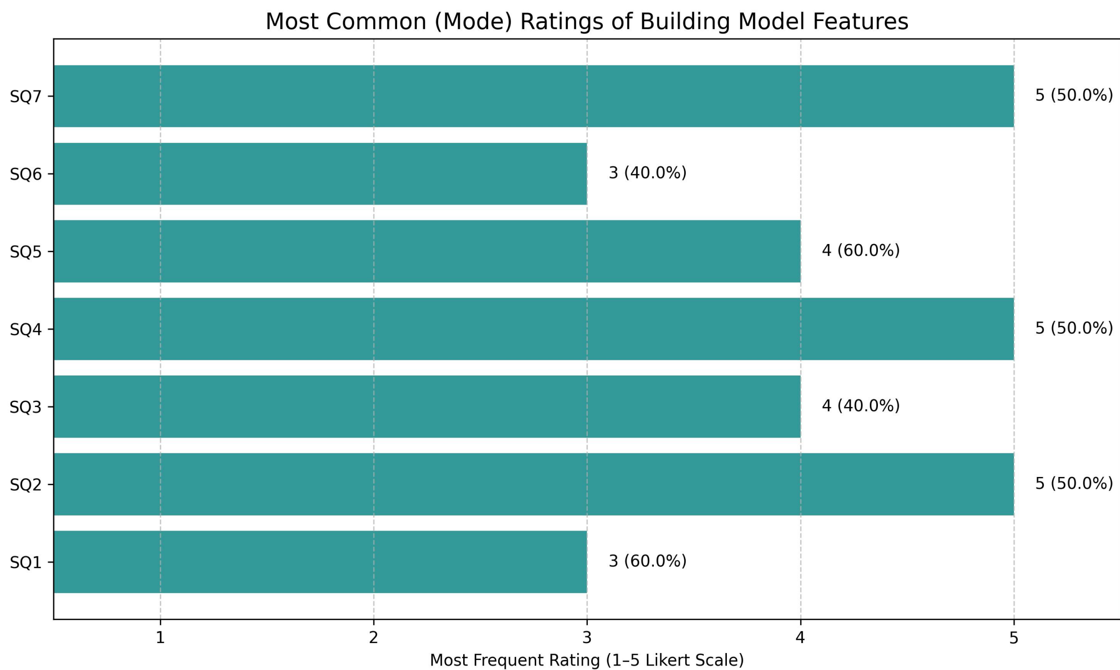


Figure 3.3: **Modal Responses to Building Model Feature Survey.** The figure illustrates the distribution of the most frequently selected responses (mode) for each Survey Question (SQ) on a 5-point Likert scale, where 1 represents the lowest perceived importance/difficulty and 5 represents the highest. **Key Finding:** 50% of respondents assigned the highest rating (5) to SQ2, SQ4, and SQ7, underscoring the perceived critical nature (importance or difficulty) of these three concerns.

by sensors and their data (5 respondents), solar photovoltaic (4), meters (4), heat pumps (4), and air handling units (3). Also, model accuracy emerged as the most frequently cited concern when modelling buildings for energy efficiency, with eight respondents selecting it as a critical factor.

3.2.2 Design of MetamEnTh

With the insights gained from our review of existing ontologies, the needs assessment, and our tour of the HVAC systems at Concordia University, we proceed to design an alternative object-oriented metamodel for representing MEP entities in the built environment. Using UML notation, we define three fundamental relationship types (Barbier, Henderson-Sellers, Opdahl, & Gogolla, 2001) for modelling the relationships among MEP subsystems:

- Association: This approach is commonly used when two classes or entities need to communicate, as denoted by a line connecting them. An arrow illustrates the direction of the

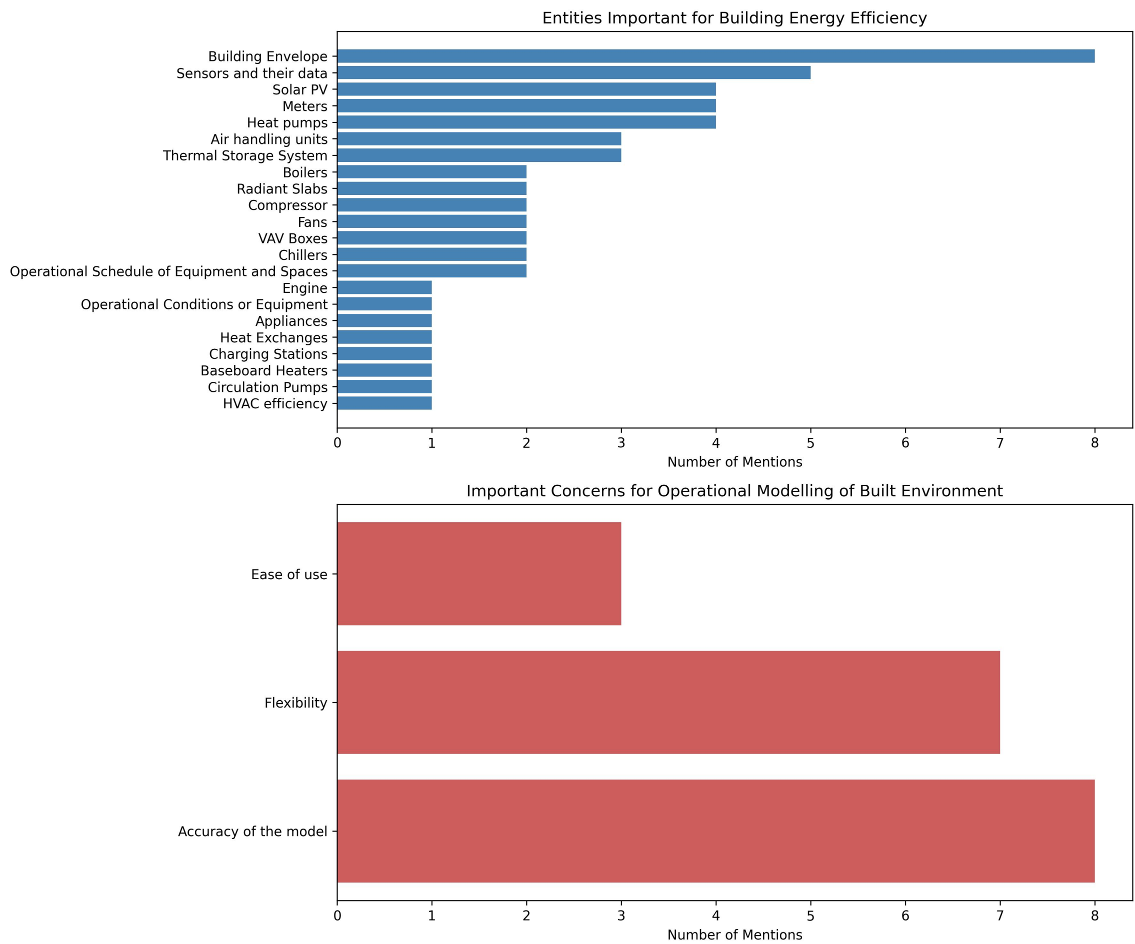


Figure 3.4: **Respondents' ratings of building model features.** The first plot shows respondents' selections of building systems and entities they consider relevant for energy efficiency. The second plot summarises the most important concerns when modelling buildings for energy-efficient operations, concerning accuracy, flexibility, and ease of use.

relationship. For example, in Figure 3.8, there exists an association relationship between the building and meter classes, where the building class is aware of the meter class (as depicted by the arrow), but the meter class does not possess knowledge of the building class.

- Aggregation: This is an association relationship between a child and parent class, where the child class *can* exist independently without the parent. A hollow diamond symbol attached to the parent class illustrates this relationship. For example, in Figure 3.9, there is an aggregation relationship between the appliance class and space (floor, room, and open space) where the spatial classes (floor, room, and open space) are the parents. The implication is that appliances

can exist independently of the rooms they associate with. The appliances can be moved to different rooms if the rooms they are associated with get decommissioned.

- Composition: This is an association relationship between a child and parent class, where the child class *can not* exist independently of the parent. A filled diamond symbol attached to the parent class illustrates this relationship. For example, in Figure 3.8, a room cannot exist autonomously without being associated with a specific floor. Similarly, a floor cannot exist independently without being associated with a particular building.

3.2.2.1 MetamEnTh: Metamodel for Energy Things

We propose a high-level meta-metamodel that represents various entities in the built environment along with their relationships. As illustrated in Figure 3.5, the meta-metamodel defines key classes, their associated properties and behaviours, and the interrelationships among these classes. However, a key limitation of this high-level abstraction—common to many ontology-based models—is that it permits the creation of syntactically valid instances that may not be semantically correct. For example, in Figure 3.5, a `Component` may be linked to one or more other `Component` instances. While `Fan` and `Damper` are valid HVAC components, a model that directly relates them may be valid in terms of the metamodel, yet incorrect from a domain-specific perspective (Figure 3.6), as fans and dampers do not interact directly within an HVAC system.

To address the limitations of the high-level metamodel, we introduced a **low-level metamodel** derived from the proposed meta-metamodel. This low-level metamodel defines specific classes with predefined properties and relationships, and serves as a foundation for enforcing domain constraints and validation rules. Figure 3.7, although complex, illustrates selected classes, their interrelationships, and the packages to which they belong.

3.2.2.2 Packages of MetamEnTh

We organised the **MetamEnTh** classes into six packages, informed by our scope of interest, review of existing systems and insights from the needs assessment. Figure 3.7 illustrates the overall structure of these packages and their constituent classes, which we describe in detail below. Given

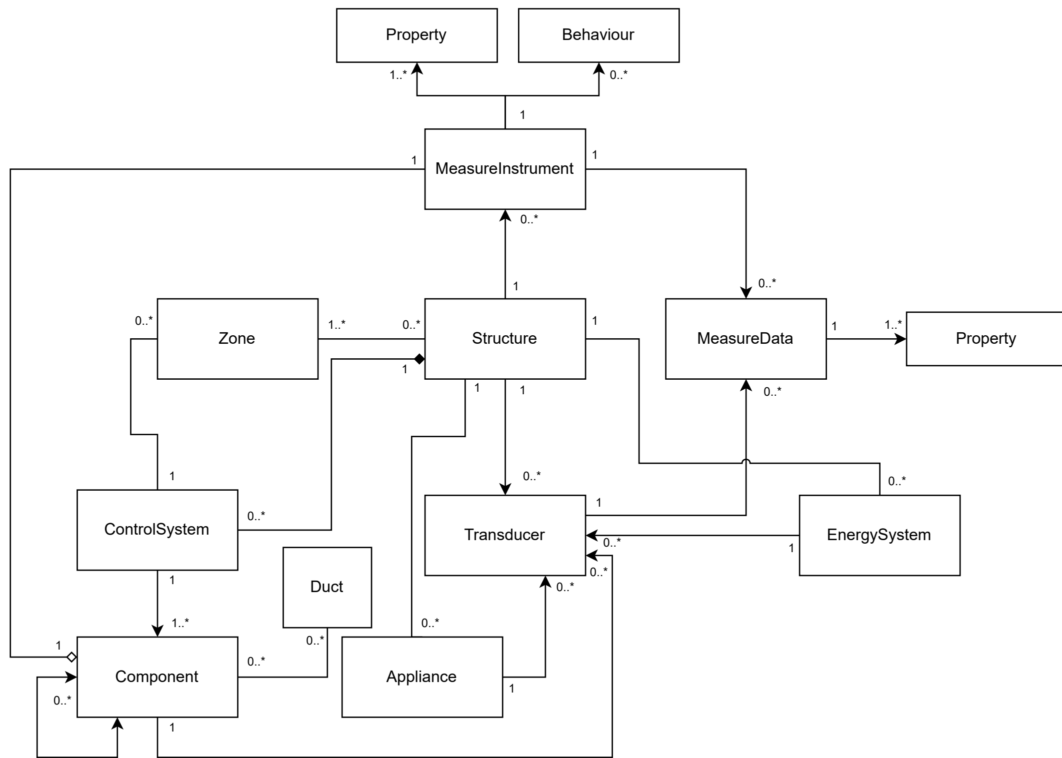


Figure 3.5: A high-level meta-metamodel for built environments showing some classes and their relationships

the inherent complexity and breadth of MEP systems, it is impractical to comprehensively model every possible configuration. Therefore, we designed **MetamEnTh** with extensibility as a core principle, enabling stakeholders to define additional classes, attributes, and relationships to represent new entities and evolving requirements.

Practitioners and researchers could therefore extend **MetamEnTh** to improve its comprehensiveness in modelling specific MEP subsystems. For example, the `BuildingControlSystem` class within the `Subsystem` package currently aggregates various control systems but explicitly implements only the `HVACSystem`. Future work could expand this to include additional control systems such as `LightingControlSystem`, `AccessControlSystem`, `FireSafety-ControlSystem`, and `WaterManagementControlSystem`.

- **Structure**: This package defines the spatial entities of a building, such as rooms, floors, and open spaces, and their relationships

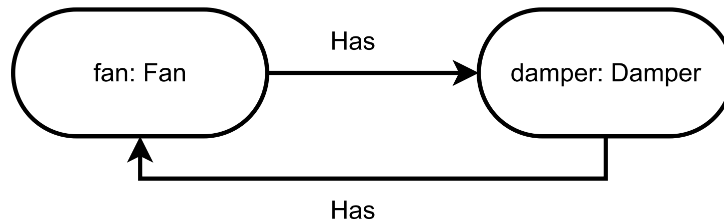


Figure 3.6: An instance of the high-level meta-metamodel illustrating a valid but inaccurate relationship between a damper and a fan

- **MeasureInstrument**: This package represents a building's weather station as well as meters that measure various environmental phenomena. It also includes the data tables used to store measurements collected by meters and sensors
- **Subsystem**: The Subsystem package contains two main components that model *energy-consuming* entities within buildings:
    - **BuildingControlSystem**: Currently, this contains a single `ControlSystem` dedicated to HVAC systems, including their entities, relationships, and associations with other components such as `Transducer`
    - **Appliance**: This class represents appliances and smart devices, along with their relationships to transducers and spatial elements
- **Transducer**: This package models `Sensor` and `Actuator` entities and their associations with other system components
- **EnergySystem**: This package describes the energy systems, organised into four subclasses focusing on renewable sources, non-renewable sources, and energy storage systems:
    - **RenewableEnergySystem**: Models renewable energy sources, including two primary subclasses: `SolarPV` and `WindMill`.
    - **NonRenewableEnergySystem**: Represents non-renewable energy sources, with a single subclass: `Engine`.

- **ImmobileEnergyStorageSystem**: Models *stationary* energy storage systems with two main subclasses: `Battery` and `SuperCapacitor`
- **MobileEnergyStorageSystem**: Models *mobile* energy storage systems, currently represented by one subclass: `ElectricVehicle`

- **Virtual**: This package models logical divisions within buildings

Structure. The **Structure** package (Peter Yefi, 2024b) of **MetamEnTh** defines spatial entities—including `Building`, `Floor`, `Room`, and `OpenSpace`—and their relationships. As illustrated in Figure 3.8, the `AbstractSpace` class captures properties common to floors, rooms, and open spaces, while `AbstractFloorSpace` refines these to define additional characteristics specific to rooms and open spaces. Through this inheritance, rooms and open spaces share relationships such as `adjacentSpaces`.

`AbstractFloorSpace` (encompassing rooms and open spaces) can contain zero or more `Meter` instances. Because `AbstractSpace` extends `AbstractZonalSpace`, all floors, rooms, and open spaces can host multiple zones, including *overlappingZones* and *adjacentZones* such as *LightingZone* or *HVACZone*. These HVAC zones may be classified as interior or perimeter zones via the *HVACType* property of the `Zone` class. Rooms and open spaces can also contain multiple `AbstractTransducer` instances (representing sensors and actuators) through inheritance from `AbstractFloorSpace`, which is derived from `AbstractDynamicEntity`.

A `Building` can have one or more `Envelope`, enabling users to represent buildings with multiple towers by specifying separate envelopes for each tower. Each `Envelope` aggregates multiple `Cover` elements, which comprise one or more `Layer` instances made of different *Material*.

Additionally, a `Building` contains exactly one `SchedulableEntity` with multiple `OperationalSchedule` entries. This design allows users to define a building's operational schedules—including name, start date and time, end date and time, setpoint, and whether the schedule recurs. Rooms, floors, and open spaces can define their schedules via inheritance from `AbstractSpace`.

MetamEnTh explicitly defines the relationships and cardinalities between spatial entities and other components. For example:

- A `Building` can contain multiple `Floor` instances, and each `Floor` belongs to exactly one `Building`.

- A `Floor` can include zero or more `Room` and `OpenSpace` instances, but must have at least one `OpenSpace` or `Room` by default.

- A `Building` can have more than one `WeatherStation`, which is useful in large or complex buildings where distinct microclimates may exist across different areas (Sharpe, 1987).

MetamEnTh also enforces property types and enumerations that constrain permissible values. For example, each property is bound to a defined data type:

- The *constructionYear* property requires a four-digit number.

- The *height* property expects a `BinaryMeasure`, which encapsulates both a numeric value and its unit in a single object.

Similarly, the *address* property is of type *Address*, which includes attributes such as *city*, *street*, *state*, *zipCode*, *country*, *geoCoordinates*, *what3words*, and *northOrientation*. The *geoCoordinates* attribute is itself a *Point* object containing latitude and longitude. The *what3words* (Arthur, 2023) property encodes a unique three-word string identifying a precise $3\,\mathrm{m} \times 3\,\mathrm{m}$ square, offering an alternative geolocation reference.

***MetamEnTh** enforce these constraints to prevent users from creating invalid or incomplete models. For example, each `Building` must instantiate at least one `Floor`, and each `Floor` must contain at least one `Room` or `OpenSpace`. Likewise, the setter method for constructionYear ensures that only four-digit numbers are accepted.*

Subsystem, MeasureInstrument, and Transducer. The **Subsystem** package (Peter Yefi, 2024b) has two main classes: `BuildingControlSystem` and `Appliance`, with `BuildingControlSystem` having one control system class, `HVACSystem`, in a composition (a whole-part relationship where `BuildingControlSystem` is the whole and `HVACSystem`, the part (Barbier et al., 2001)) relationship. This design allows the extension and inclusion of other control systems in **MetamEnTh** in the future.

Figure 3.9: **MetamEnTh** `Appliance`, illustrating the `Appliance` class and its relationships with `AbstractTransducer` and spatial entities (`AbstractSpace`)

Figure 3.9 presents the `Appliance` class from the **Subsystem** package and its relationships with other entities. It also illustrates the `AbstractTransducer` class, which has two subclasses—`Sensor` and `Actuator`—along with their associated data classes: `SensorData` and `TriggerHistory`, which represent data generated by sensors and actuators, respectively.

A one-to-many aggregation relationship exists between `AbstractTransducer` in the **Transducer** package, and `AbstractSpace`, indicating that multiple transducers can be located within a given space (e.g., room, floor, or open space). This relationship is essential for modelling transducers that are directly attached or installed in spatial entities.

MetamEnTh separates dynamic data from the entities that produce them. This design accommodates varying sampling strategies. For example, some sensors may collect raw data at 500-millisecond intervals but report averaged values every five seconds for storage. In addition to periodic recording, BMS may use change-of-value (COV) strategies to store only data points that differ

from the last recorded value. The *changeOfValue* boolean property in the Sensor class models this behaviour, clarifying whether the stored frequency reflects the actual sampling rate or a filtered dataset based on value changes.

MetamEnTh's MeasureInstrument package models various meters and weather stations equipped with sensors that capture different phenomena. These measurements are represented by the SensorData, StatusMeasure, MeterMeasure, TriggerHistory, and WeatherData classes. **MetamEnTh** defines explicit relationships between meters and other entities—including buildings, floors, rooms, open spaces, and HVAC components. These associations enable the measurement of metrics such as power consumption and energy usage for specific parts of the building.

MetamEnTh models the HVAC control system, part of the **BuildingControlSystem**, as comprising various AbstractHVACComponent, Duct, and Engine, as illustrated in Figure 3.10. The Duct class inherits from AbstractDynamicEntity, enabling ducts to include multiple sensors and to be associated with one or more Zone instances. The *connections* property of type *DuctConnection* allows users to specify the permitted AbstractHVACComponent entities—and rooms or open spaces—that serve as the source and destination of airflow within a duct.

Additional properties such as *fans*, *heatExchangers*, *filter*, *connectedAirVolumeBoxes*, and *damper* capture the components commonly installed inside ventilation ducts or air handling units.

An HVACSystem can include multiple VentilationSystem instances, each of which has a designated principalDuct. Other AbstractHVACComponents can be connected to this principal duct through additional ducts and pipes, represented as *association classes*. This design supports the flexible modelling of HVAC systems and their complex interactions. **MetamEnTh** also enables each AbstractHVACComponent to define its own *operationalSchedule*, in addition to specifying the spatial entities where the components are installed.

EnergySystem. The **EnergySystem** package models the diverse energy sources in buildings. It is organised into four main categories: RenewableEnergySystem, NonRenewableEnergySystem, MobileStorageEnergySystem, and ImmobileStorageEnergySystem.

As in other parts of **MetamEnTh**, parent classes define common properties that are inherited

by specialised subclasses. For example, AbstractCommonEnergySystem provides shared attributes for both AbstractElectrical and AbstractEnergySystem. The classes UPS, ATS, and Alternator inherit from AbstractElectrical, while MobileStorageEnergySystem, ImmobileStorageEnergySystem, RenewableEnergySystem, and NonRenewableEnergySystem are all subclasses of AbstractEnergySystem.

ElectricVehicle is an example of a MobileStorageEnergySystem, reflecting its ability to charge or discharge at building charging stations. To support this, **MetamEnTh** models connectivity data using the VehicleConnectivity class, which records periods during which electric vehicles are charging or discharging.

The RenewableEnergySystem class has two main subclasses: SolarPV and Wind. The *solarPVType* property of the SolarPV class can take one of three values: *BIPV* (Building-Integrated PhotoVoltaic), *BIPVT* (Building-Integrated PhotoVoltaic/Thermal), or *Other*—to specify the photovoltaic technology employed. Owing to **MetamEnTh's** extensible design, practitioners and researchers can readily introduce additional subclasses under RenewableEnergySystem to represent emerging or alternative renewable sources, such as geothermal, biomass, hydro, or hydrogen-based systems. Relationships between energy systems and other entities, including spatial elements and transducers, are illustrated in Figure 3.11.

The NonRenewableEnergySystem class currently has a single subclass, Engine, which defines four key properties: *engineType*, *engineSubType*, *mode* and *polyGeneration*. The *engineType* property can have one of seven values (*InternalCombustion*, *FuelCell*, *Stirling*, *MicroTurbine*, *Electrolyser*, Steam, and *Other*), while *engineSubType* can be set to one of six values (Diesel, NaturalGas, *Hydrogen*, *Petrol*, *Coal*, and *Other*), as shown in Figure 3.11.

MetamEnTh's EnergySystem package provides renewable and non-renewable energy sources to Subsystem entities, as depicted in Figures 3.7 and Figure 3.11. Notably, the Engine class also inherits from AbstractHVACComponent, allowing it to serve as an energy-consuming component of the HVAC control system. This dual role enables engines to support cogeneration—simultaneous production of electricity and useful heat (Wikipedia, 2023)—as illustrated in Figures 3.10 and 3.11. This design decision was informed by the feedback and guidance of an energy and built environment expert consulted during the development of the metamodel.

Virtual. The **Virtual** package of **MetamEnTh** contains the `Zone` class, which models a building's *Lighting* and *HVAC* zones. The *zoneType* property distinguishes between *HVAC* and *Lighting* zones, representing groups of spatial entities with adjustable environmental conditions. For HVAC zones, the *hvacType* property specifies whether the zone is *Perimeter* or *Interior*, reflecting the thermal characteristics of the space. Perimeter zones are crucial for maintaining distinct indoor conditions around the building's perimeter.

Figure 2.3 on Page 20, taken from the BMS of Concordia's EV building, illustrates three thermal zones on the north side of the tenth floor: two perimeter zones (yellow and red) and one interior zone (cyan). Multiple spatial entities can belong to the same HVAC or lighting zone, and a single spatial entity—such as a room—can be part of multiple zones.

Figure 3.8 shows how the `Zone` class relates to spatial entities in **MetamEnTh**. Specifically, the `AbstractZonalEntity` defines the *zones* property—a list of associated zones—that is inherited by `AbstractFloorSpace`. This design allows users to accurately represent overlapping or adjacent zones in complex spaces.

3.3 Evaluation

In Chapter 2 we discussed the limitations of existing approaches for modelling the operational phase of MEP subsystems. Specifically, we presented eight limitations of Project Haystack and Brick in modelling MEP subsystems. In this section, we highlight how **MetamEnTh** addresses those challenges through case studies and practical, empirical evaluations.

3.3.1 Case Study: Evaluation of UML Designs

3.3.1.1 Introduction

The objective of these case studies is to validate **MetamEnTh** against real-world BMS data to ensure it can comprehensively represent MEP entities. We selected three buildings in consultation with an expert: two institutional buildings at Concordia University in Montreal and the Varennes Library in the municipality of Varennes. Two buildings are managed by Regulvar BMS (Regulvar, 2025), and one by Siemens (Siemens, 2025c) BMS.

- The Engineering, Computer Science, and Visual Arts Integrated Complex (EV) features two towers, dual weather stations, and a sophisticated, custom-built HVAC system. We anticipated that if **MetamEnTh** could successfully model this complexity, it would be able to handle less intricate systems.

- The John Molson School of Business (MB/JMSB) building, connected to the EV building, is a LEED-certified facility. The MB Building presents a challenging environment for modelling because of its complex HVAC system and the presence of several unique building subsystems not found in the EV Building.

- The Varennes Library is Canada's first institutional net-zero building, introducing elements such as heat pumps and radiant slabs that are not found in the other two buildings.

The first case study focused extensively on the EV building to confirm **MetamEnTh's** ability to model intricate relationships and entities. Subsequent case studies concentrated on modelling elements absent from the first study, avoiding unnecessary duplication. These additional cases helped refine **MetamEnTh** by incorporating entities previously underrepresented. **MetamEnTh** models for all use cases are available online (Peter Yefi, 2024a).

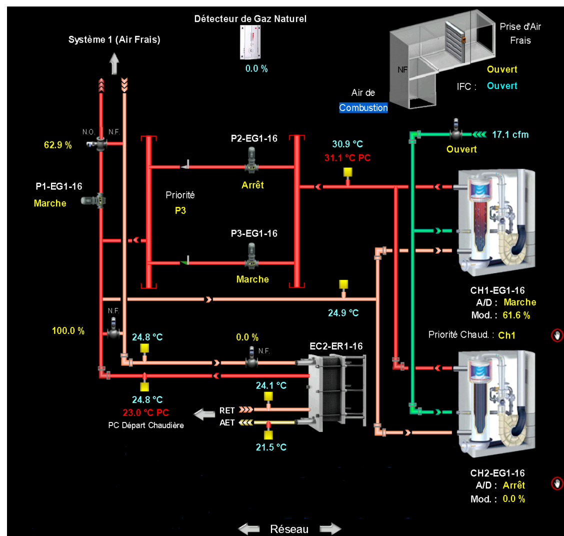
Summary of Case Studies In the EV Building, we selected the 10th, 12th, and 17th floors. The 17th floor hosts the main ventilation systems and contains most of the central HVAC components, while the other floors demonstrate how **MetamEnTh** models connections between the principal ventilation system and floor-level equipment. Using BMS data from the three buildings, we validated **MetamEnTh's** ability to represent structural entities (buildings, floors, rooms, and open spaces), measuring instruments (weather stations and meters), virtual entities (thermal zones), transducers (sensors and data points), and HVAC components (ducts, pipes, boilers, chillers, cooling towers, heat exchangers, and fans), along with their interrelationships. While modelling these entities, we identified limitations and refined **MetamEnTh** to address them:

- Added properties to the `Duct` class to explicitly relate ducts with fans, chillers, dampers, and heat exchangers installed within them.

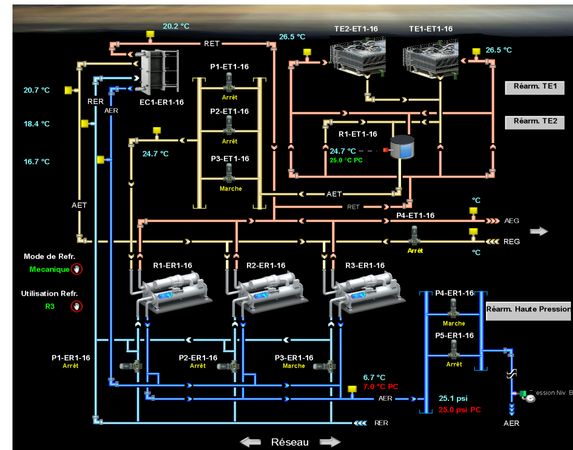
- Enabled transducers (sensors) to attach directly to spatial entities such as rooms and corridors.
- Extended the `DuctConnection` property to represent substance flow between HVAC components like chillers and cooling towers.
- Introduced the `ductSubType` property to further classify ducts (e.g., ReturnAir, FreshAir).
- Added a boolean property to `Sensor` to indicate whether recorded readings are determined by a change of value rather than sampling interval.
- Included a property in `SolarPV` to model the heat capacity of BIPV/T panels.
- Modified thermal zone relationships to allow rooms to belong to multiple overlapping zones.

In the MB/JMSB Building, we focused on the 15th, 16th floors, and the first sub-basement. The 16th floor houses the central ventilation system and BIPV/T installation, while the other floors help model ventilation connections across levels. This case study introduced entities and relationships not encountered in the EV building, including baseboard heaters, dampers and actuators, VAV boxes, and BIPV/T panels. We also modelled the complex interconnections of HVAC components such as boilers and chillers, whose pipe arrangements exceeded the complexity seen in the EV building (see Figure 3.12). To accommodate these findings, we made further updates:

- Added a class to represent baseboard heaters used for supplemental heating.
- Introduced a damper class to model dampers regulating airflow.
- Added a boolean property to the `AirVolumeBox` to indicate whether it provides heating.



(a) Boilers



(b) Chillers and cooling towers

Figure 3.12: Interconnected through a complex network of pipes, the boilers, chillers, and cooling towers form the principal ventilation systems of the MB/JMSB Building.

In the Varennes Library, Canada's first institutional net-zero building, we encountered heat pumps and radiant slabs not present in the prior cases. **MetamEnTh** already included a heat pump class, which we applied as the library's primary heat source. We modelled radiant slabs—floor installations that contribute thermal mass and maintain indoor conditions—as new control system entities, with floors divided into multiple thermal zones corresponding to slab configurations. Based on this case study, we made additional refinements:

- Added a control system class to represent radiant slabs for heating and cooling.
- Included a `heatSource` property to specify heat pump sources.
- Added a `Other` option to most enumerated property values within HVAC components, recognising the impracticality of exhaustively listing all possible configurations.

Figures A.1 and A.2 on Page 193 of Appendix A show UML instances of aspects of the EV Building and MB/JMSB Building as modelled with **MetamEnTh**. Collectively, these case studies demonstrated that **MetamEnTh** can comprehensively model a wide range of MEP systems and their complex relationships while remaining adaptable to unanticipated building configurations.

Through these case studies, we demonstrate how **MetamEnTh** addresses limitations L_1, L_3, and L_5 as explained below.

L_1: Possible Ambiguity in Defining an Entity. As illustrated in all three use cases, **MetamEnTh** uses predefined classes with explicitly defined relationships and cardinality constraints to model MEP subsystems. These structured class definitions eliminate much of the ambiguity inherent in tag-based modelling, where the meaning of an entity often emerges implicitly from the chosen tags. Although the predefined classes and relationships impose strict rules to enforce constraints and ensure accurate models—which can introduce rigidity—**MetamEnTh** is designed to support flexible yet structured configuration of complex subsystems, as illustrated in the model of the complex EV Building HVAC system.

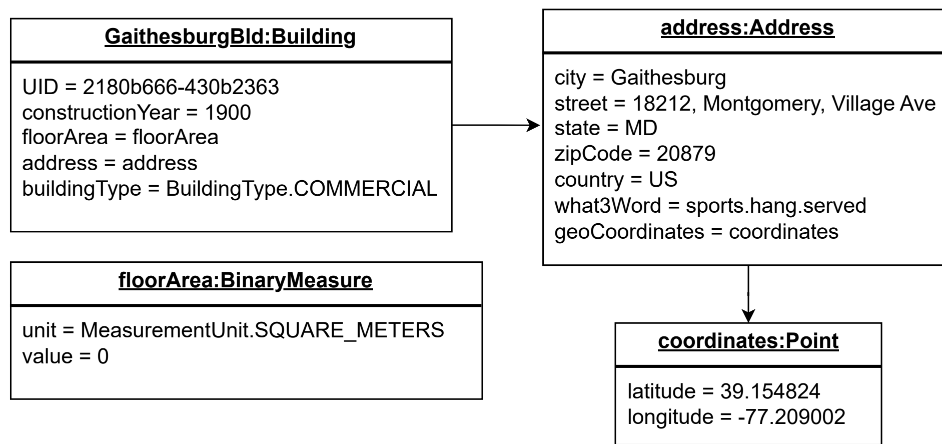


Figure 3.13: A **MetamEnTh** modelling of the building in Figure 2.1 on Page 16

For example, Figure 3.13 illustrates the **MetamEnTh** representation of a building, corresponding to the Project Haystack building shown in Figure 2.1 in Chapter 2 Section 2.1.1. In the **MetamEnTh** model, the `Building` class includes predefined properties such as `UID`, `constructionYear`, and `floorArea`, each associated with specific data types like `string`, `number`, and `BinaryMeasure`. This structure ensures that users provide valid, appropriately typed values for these properties, in accordance with the metamodel's constraints. Additionally, users must instantiate related objects such as `Address`, `Point`, and `BinaryMeasure` to fully specify the values of certain building properties, thereby ensuring a richer, more structured representation.

L_3: **Lack of Entities for Ducts, Pipes, and Detailed Spatial Context.** In Figure 2.2b (Chapter 2 Section 2.1.1), four `Duct` instances represent the vents, each linked to the same air handling unit (AHU)—with two ducts supplying air and two returning air to rooms 101 and 102, respectively. In this configuration, the user sets the `source` attribute of each `DuctConnection` object to the AHU's UID for all four ducts. In contrast, the `destination` attribute references the specific room each duct serves. The user can link entities such as `VAV` boxes and dampers to the corresponding ducts by assigning their properties to reference the hosting `Duct` instance. In cases where distinct `VAV` boxes independently regulate airflow and temperature in rooms 101 and 102, the user can unambiguously associate each device with the relevant supply air duct.

In the EV Building case study, we demonstrate how **MetamEnTh** can model more complex HVAC configurations. The 17th-floor principal ventilation duct serves as the AHU for the HVAC system, connecting various MEP components—including chillers, boilers, fans, and heat exchangers—for air conditioning. This central duct supplies conditioned fresh air to multiple zones and returns exhaust air via dedicated return air ducts.

L_5: **Inconsistent Data and Structure Semantics.** In **MetamEnTh**, the structured design of classes promotes inheritance-based design related entities, ensuring that common properties are meaningfully shared. By organising related classes under appropriate parent abstractions, the metamodel avoids inconsistencies between data semantics and structural semantics. For example, the `AbstractTransducer` class serves as the parent of `Sensor` and `Actuator`, as these entities share core attributes and behaviours that define them as transducers. Any class inheriting from `AbstractTransducer` acquires these properties and, by definition, qualifies as a transducer.

The MB Building case study demonstrates these capabilities of **MetamEnTh**: specifically, it illustrates how concrete `Sensor` and `Actuator` instances can possess distinct attribute values for common properties inherited from the `AbstractTransducer` base class. This disciplined approach to inheritance prevents the semantic ambiguities often encountered in less formally structured ontologies or metadata schemes.

3.3.2 Case Study: Practical and Empirical Evaluation

We adopted a two-pronged evaluation strategy to assess the implementation of **MetamEnTh**. First, we conducted an empirical evaluation in which developers implemented predefined scenarios and provided feedback via survey responses. Second, we conducted a practical evaluation using Concordia University's Webster Library as a real-world use case.

3.3.2.1 Implementation of MetamEnTh

We implemented **MetamEnTh** in Python to ensure ease of use and seamless integration by researchers at the Next Generation Cities Institute, Concordia University. The implementation includes thoroughly documented unit tests to verify correctness and illustrate practical application scenarios. **MetamEnTh** leverages established design patterns such as Visitor, Strategy, Factory, Observer, and Decorator to ensure modularity, enforce constraints, and support extensibility. Validation logic is encoded within setter methods and complemented by utility classes that perform complex checks. While this approach enhances correctness and prevents invalid configurations, it introduces a steeper learning curve for ontology users, particularly in non-standard cases where users may need to extend classes or adapt constraint logic.

MetamEnTh is publicly available on GitHub (Peter Yefi, 2025c), with detailed documentation hosted on GitHub Wiki (Peter Yefi, 2024b). The current release is published on PyPI (Peter Yefi, 2025b) to facilitate installation and adoption. The documentation provides information about the packages, classes, properties, behaviours, and relationships defined in **MetamEnTh**.

3.3.2.2 Empirical Evaluation of MetamEnTh

After implementing **MetamEnTh** in Python, we conducted an empirical experiment comparing **MetamEnTh** with the Brick ontology. This evaluation, which involved creating and comparing Brick and **MetamEnTh** models, helps us fully address **RQ1.2** and **RQ2**.

We contacted prospective respondents via email, resulting in 10 participants—8 males and 2 females. Figure 3.15 presents the demographic characteristics of the participants. As shown, four participants held master's degrees, four held bachelor's degrees, one held a doctoral degree, and

one had completed secondary education. Five participants were software engineers or developers, while the remaining five were software engineering students or researchers. Prior to the main study, we conducted a pilot experiment with one additional participant (excluded from the final sample) to refine the experimental design based on their feedback. To further mitigate potential bias, the study design was reviewed by researchers experienced in ontology engineering.

The experiment consisted of two parts. In part one, participants performed two tasks: (1) modelling the HVAC system shown in Figure 3.14 using both **MetamEnTh** and Brick, and (2) creating appropriate relationships between selected HVAC entities to represent their interactions within each modelling approach. Part two consisted of a post-task questionnaire in which participants reflected on their experience using both approaches.

Comparison Criteria: The questionnaire responses enabled a comparative evaluation of Brick and **MetamEnTh** across three criteria derived from participants' post-experiment feedback:

- **Correctness** in defining entities and their relationships
- **Level of detail** in capturing critical entities and their interactions
- **Fault tolerance**, i.e., the ability to prevent users from creating incorrect or inconsistent models through the use of constraints

These criteria are particularly relevant for modelling the operational aspects of built environments with energy-efficiency objectives, as identified in the needs assessment we conducted with researchers and practitioners. The importance of the level of detail in capturing critical entities and their interactions is also emphasised in the ontology literature Bhattacharya et al. (2015).

Results of Empirical Evaluation. Tables 3.1 and 3.2 present the summary statistics and results of statistical tests based on participants' responses after completing Task One. On a Likert scale of 1 to 5, where 1 indicates low familiarity and 5 indicates high familiarity with building systems and their operations, the average response was 2.1, with a median of 2. This outcome suggests that most participants had limited prior exposure to building systems. Although Likert data are ordinal, we report means and standard deviations to summarise central tendencies and variability, following

recommendations by Norman (2010); Sullivan and Artino Jr (2013) that Likert data can be treated as interval-level for descriptive and inferential analyses when scales have five or more points.

To compare participant ratings of Brick and **MetamEnTh** across the four questions, we conducted paired statistical tests. Specifically, we used a paired t-test when the differences between paired responses followed a normal distribution and the non-parametric Wilcoxon signed-rank test otherwise. These choices ensure the validity of the comparisons by accounting for the scale of the data and the small sample size ($N = 10$). The following summarises the findings:

- **Modelling Detail (General and HVAC-specific):** Brick and **MetamEnTh** received comparable ratings for general modelling detail (means: 3.9 vs. 3.8; medians: both 4.0), suggesting similar perceived completeness. For HVAC-specific detail, **MetamEnTh** was rated higher (mean = 4.3 vs. 3.6) and exhibited less variability (SD = 0.82 vs. 1.35), indicating a perceived advantage, though the difference was not statistically significant ($p = 0.100$, Wilcoxon test).

- **Modelling Difficulty:** Participants rated **MetamEnTh** as significantly more difficult (mean = 3.6 vs. 2.5; medians: 4.0 vs. 2.0; $p = 0.027$, Wilcoxon test). This outcome is expected, as **MetamEnTh** enforces modelling constraints and flags errors during the modelling process, reducing the chances of inaccurate models. In contrast, Brick, based on RDFS and OWL, does not provide such enforcement.

- **Model Accuracy:** **MetamEnTh** was rated slightly higher in producing accurate models (mean = 4.2 vs. 3.9), with more consistent responses (SD = 0.63 vs. 1.10). However, this difference was not statistically significant ($p = 0.541$, paired t-test).

Only the difference in perceived modelling difficulty reached statistical significance (Wilcoxon signed-rank test, $p = 0.027$). We observed no statistically significant differences for the other criteria ($p > 0.05$). Given the small sample size ($N = 10$), these results should be interpreted cautiously. However, the consistent trend favouring **MetamEnTh** across all criteria suggests a potential participant preference, warranting further investigation in future work.

The goal of Task Two was to evaluate whether participants could define four specific domain relationships using two different modelling approaches — **Brick** and **MetamEnTh**. In this task,

Table 3.1: Summary statistics for questionnaire responses comparing Brick and **MetamEnTh**

Question	Approach	Mean	Median	Min	Max	Std Dev
General Detail of Model	Brick	3.9	4.0	2	5	0.99
	MetamEnTh	3.8	4.0	2	5	1.03
Modelling Difficulty	Brick	2.5	2.0	1	5	1.08
	MetamEnTh	3.6	4.0	2	5	0.84
Model Accuracy	Brick	3.9	4.0	2	5	1.10
	MetamEnTh	4.2	4.0	3	5	0.63
Detail in Modelling HVAC Entities	Brick	3.6	4.0	1	5	1.35
	MetamEnTh	4.3	4.5	3	5	0.82

Table 3.2: Results of statistical tests comparing Brick and **MetamEnTh** responses

Question	Test Used	p-value	Significant ($p < 0.05$)?
General Modelling Detail	Paired t-test	0.859	No
Modelling Difficulty	Wilcoxon	0.027	**Yes**
Modelling Accuracy	Paired t-test	0.541	No
HVAC Modelling Detail	Wilcoxon	0.100	No

participants built upon the model created in Task One to define additional entities and their relationships. To mitigate any bias in favour of **MetamEnTh**, we engaged external researchers experienced in ontological engineering to independently review the task, ensuring conceptual neutrality between the two modelling approaches. The task involved creating the following entities and relationships:

- Creating a relationship between the actuator and the temperature sensor
- Adding the pressure sensor to the kitchen
- Adding any damper to Office 1
- Adding the actuator to the corridor

While Brick allowed participants to define these relationships freely, **MetamEnTh** enforced validation rules, guiding users to build semantically correct models.

A total of 10 participants were enrolled in the study. However, we removed four from the analysis of Task Two:

- Three participants did not attempt Task Two, as evidenced by the absence of relevant implementation in their code.

- One participant's code was missing, so their responses could not be verified.

Thus, six participants remained for the final analysis, where both their perceived success (as indicated by survey responses) and actual success (as determined by code inspection) were evaluated.

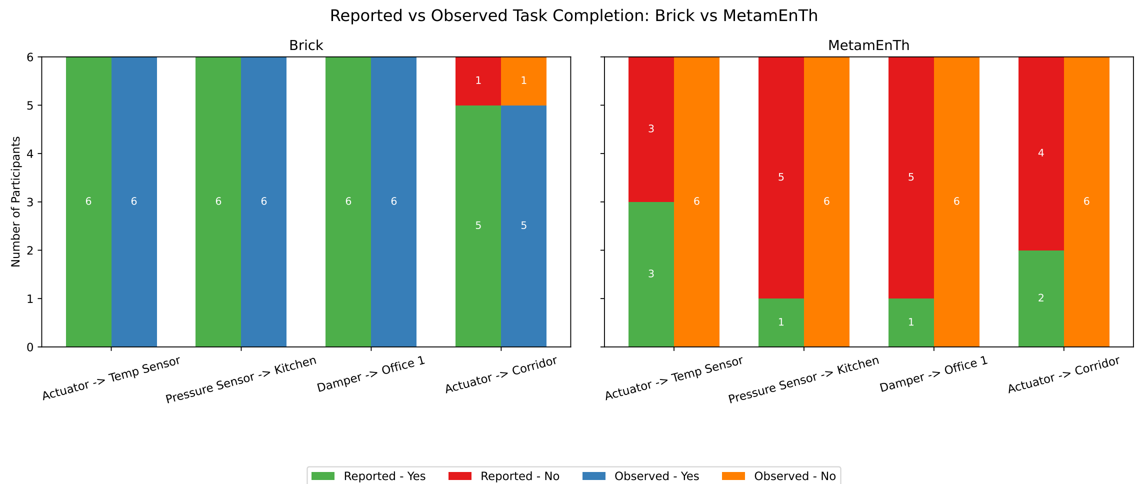


Figure 3.16: Observed versus reported responses of participants on their ability to create various relationships between entities in Task Two

Findings of Empirical Evaluation. All six participants successfully created the four relationships using Brick, except one who failed to connect the actuator to the corridor. None of the six participants correctly implemented all four relationships using **MetamEnTh**.

Figure 3.16 shows the responses as observed and reported by the participants. Although some participants reported success with **MetamEnTh** in the survey—e.g., three participants reported creating a relationship between the actuator and temperature sensor—a closer inspection of their code revealed discrepancies between perceived and actual outcomes. Examples include:

- One participant attempted to add a damper to Office 1, but the operation failed due to validation errors, and the damper was never added.
- Another participant retrieved an actuator object as `None` and attempted to add it to the corridor, but the operation failed with no reported error, but the relationship was never established.
- In multiple cases, participants created relationships between the controller and temperature sensor instead of between the actuator and temperature sensor. This could be because they

did not see methods or properties for adding sensors to actuators.

- One participant added the pressure sensor to a VAV box instead of the kitchen.

- Another added the actuator to the VAV box rather than the required corridor.

These examples illustrate how **MetamEnTh** helped prevent invalid or semantically inappropriate relationships, even if it didn't always flag the issues as errors. In contrast, Brick allowed all of these relationships—correct or not—to be added without restriction or feedback.

Because all six participants failed to establish the correct relationships using **MetamEnTh**, and due to the small sample size, McNemar's test (McNemar, 1947) could not be applied. However, the qualitative insights clearly support the conclusion that **MetamEnTh** enforces higher modelling accuracy and semantic fidelity through validation constraints, while Brick prioritises flexibility over correctness, demonstrating how **MetamEnTh** addresses L_2: Possibility of Composing Flawed Entity Relationships and L_4: Difficulty Validating Entities and Relationships. Because validation is an integral part of model instantiation, **MetamEnTh** enforces defined constraints to ensure their conformance to the metamodel.

3.3.2.3 Practical Evaluation of MetamEnTh

In this section, we describe the process of modelling Concordia University's Webster Library (LB Building) to evaluate the modelling capabilities of **MetamEnTh**. The LB Building, located on the Sir George Williams campus in Montreal, comprises the Mackay and Bishop towers, which span six and ten floors, respectively (excluding the ground floor). The building also includes two basement levels. For this use case, we manually collected detailed envelope data from architectural drawings for all four façades of the building, excluding the basement levels. Additionally, we extracted data on the various floors, rooms, and open spaces from Archidata (Archidata Inc., 2025). This information was then structured into a JSON file using a dedicated Python class. The subsequent sections detail how we extracted and structured data from different sources to support the creation of the building model.

Table 3.3: Layer composition of cover RT-B1 of the LB Building

Name	Length (mm)	Height (mm)	Thickness (mm)	Material	Layer Order	Description
RT-B1	3,200	2,800	16	ordinary gypsum board	1	Window Area
RT-B1	3,200	2,800	42	metal half-timbering	2	Window Area
RT-B1	3,200	2,800	92	semi-rigid insulation	4	Window Area
RT-B1	3,200	2,800	16	ordinary gypsum board	5	Window Area
RT-B1	3,200	2,800	38	semi-rigid insulation	6	Window Area
RT-B1	3,200	2,800	21	air gap	7	Window Area
RT-B1	3,200	2,800	94	varnished concrete block	8	Window Area
RT-B1	2,000	2000	6	glass	1	Window
RT-B1	2,000	2000	12	window gas	2	Window
RT-B1	2,000	2000	6	glass	3	Window
RT-B1	2,000	2000	12	window gas	4	Window
RT-B1	2,000	2000	6	glass	5	Window

Data Extraction and Structuring.

- **Architectural Drawings**: We reviewed the architectural drawings and structured the envelope data into different tabs of a spreadsheet. For the block covers, we collected data on each unique cover (e.g., *RT_C_P*, as illustrated in Figure 3.17), including their constituent layers, dimensions, and the materials used in each layer. We also documented the building components associated with each block cover (e.g., *RT_C_P*), such as walls, windows, doors, and roofs. A separate tab in the spreadsheet recorded the number and distribution of unique block covers across the building's facades and floors. This level of detail supports precise queries, such as retrieving all windows located on the north facade of the first floor. For example, one could query all *RT_C_P* block covers on the first-floor northern facade. Another tab in the spreadsheet detailed the roof configurations of both towers, including the intermediate roofs between floors (e.g., the ground of the second floor acts as the roof of the first floor). Table 3.3 lists the layers in the *RT-B1* block cover, which comprises both a window (the opening) and the surrounding wall area. The *Layer Order* column indicates the sequence of layers from the interior to the exterior side of the building.

- **Archidata**: Concordia University uses Archidata (Archidata Inc., 2025) to manage some of its buildings. We manually extracted the HTML table of floors, rooms, and open spaces for the LB Building using Chrome's Developer Tools, as the website is protected, and automatic

scraping was not feasible. We developed a Python class to parse and structure the data into a JSON file containing details about the building, its floors, rooms, and open spaces. The rooms and open spaces are nested within their respective floors to enable accurate spatial mapping during model creation. Each spatial entity includes information such as the type (e.g., office), name, and area. In addition, each floor record specifies the floor number and type (e.g., basement). By combining this spatial data with the envelope data, we modelled all spatial entities of the LB Building, including the complete envelope and the roof.

- **BMS**: The LB Building is managed by the Siemens Insight BMS (Siemens Building Technologies, 2018). For each sensor located on the third floor—where the library is located—we extracted the corresponding metadata and embedded it as JSON objects within the relevant room and open space entities in the JSON file generated from the Archidata data. Additionally, we retrieved the historical time-series data for these sensors and exported it into Excel files. These datasets were then linked to the corresponding sensor instances in the model, enriching their representation with detailed, real-world operational information.

Figure 3.17: A section of the architectural drawings of the LB Building showing labelled (e.g., RT_-D_P) block covers with their dimensions and further information (below the block covers) pointing to details of the layers.

Model Instantiation. To instantiate a **MetamEnTh** model of the LB Building, we developed a modular middleware composed of four primary classes. Each class abstracts specific parts of the

modelling workflow, enabling separation of concerns and reusability. We describe the individual classes that constitute this middleware and how we utilised them for model instantiation:

(1) **Parsing and Type Conversion with *BuildingStructure***

The *BuildingStructure* class acts as a low-level interface between the extracted JSON data and **MetamEnTh's** API. It includes methods that accept string and numeric inputs, convert them to appropriate data types, and call the corresponding **MetamEnTh** methods to create spatial components such as rooms, floors, open spaces, layers, and the building entity itself.

(2) **Coordinated Model Creation with *BuildingCreator***

The *BuildingCreator* class takes a *BuildingStructure* object as input and orchestrates the instantiation of the LB Building. It sequences the use of *BuildingStructure* methods to build a complete model hierarchy within **MetamEnTh**, ensuring that entities are created in the correct order and relationships are established properly.

(3) **Populating Sensor Histories with *LBDynamicDataReader***

The *LBDynamicDataReader* class iterates over the sensor objects defined in the JSON file, creating corresponding sensor instances within their respective spatial entities. For each sensor, it accesses the associated Excel file, reads the historical time-series data, and appends these records to the sensor's data property.

(4) **Envelope Integration with *BuildingEnvelopeProcessor***

To incorporate the building envelope details, we implemented a *BuildingEnvelopeProcessor* class. This component reads envelope data from the spreadsheet and maps it to **MetamEnTh** structures. It adds detailed block covers, assigns their constituent layers and materials, and associates each with a specific location in the building (e.g., the west façade of the third floor).

Listing A.1 in Appendix A shows the *add_envelope_method* in the *BuildingCreator* class, which internally invokes *BuildingEnvelopeProcessor* to add envelope data to the instantiated building model.

The `LBDynamicDataReader` demonstrates how **MetamEnTh** addresses L_6: Complex and Deep Inheritance Hierarchies. Contrary to the four levels of inheritance illustrated in Figure 2.4, **MetamEnTh** defines CO_2 concentration sensor for the library spaces on the third floor of the LB Building

using the `Sensor` class. Figure 3.18 is a **MetamEnTh** model of CO_2 concentration sensor with `Sensor` class, which only inherits from `AbstractTransducer`.

Sensor: C02Sensor
+ UID: ec2b53d7-7ac3-4c7c-96ff-3d731244f6f5
+ name: TEC1.345-00.LB:ROOM C02
+ changeOfValue = False
+ measure: bool = SensorMeasure.CARBON_DIOXIDE_CONCENTRATION
+ dataFrequency = 900
+ unit = MeasurementUnit.PART_PER_MILLION
+ measureType = MeasureType.NON_DISPERSIVE_INFRARED
+ currentValue = None

Figure 3.18: **MetamEnTh** representation of the CO_2 sensor as illustrated in the Brick class hierarchy in Figure 2.4

To demonstrate the practical utility of the LB Building model, we developed a Python client application named *EnvelopeAnalysis*. This utility provides methods to calculate the total area of block covers, estimate U-values for specific envelope sections, and perform integrity checks by identifying layers that use low-density or high-conductivity materials.

Listing 3.1 illustrates how the model is instantiated and how the envelope data is integrated. It also shows how to query all block covers on the ground-floor southern facade. Additionally, the example demonstrates how to extract block covers that form the roof of the Bishop Tower into the *bishop_roof_covers* variable.

To further highlight the level of detail and structured querying supported by the model, we include an example that retrieves all layers containing glass material from block covers on the southern orientation of the ground floor. Furthermore, the code demonstrates how we query spatial entities within the library space on the third floor of the building.

To use the client application, we created an instance of *EnvelopeAnalysis* using the extracted ground floor block covers. The application is then used to compute total area, U-values, and perform integrity checks on the selected section of the building envelope.

```python
from utils.building_creator import BuildingCreator
from webster_library.building_envelope_processor import
    BuildingEnvelopeProcessor
from metamenth.enumerations import BuildingOrientation
from envelope_analysis import EnvelopeAnalysis
from metamenth.enumerations import MaterialType
from webster_library.lb_dynamic_data_reader import LBDynamicDataReader

if __name__ == "__main__":
    LIBRARY_FLOOR = 3
    COVER_INDEX   = 0
    webster_library = BuildingCreator('webster_library.json',
    LBDynamicDataReader(LIBRARY_FLOOR))
    webster_library.create_building()
    webster_library.add_sensor_data()

    webster_library.add_building_envelope(BuildingEnvelopeProcessor('
    webster_library_layer_catalog.xlsx'), 'LB Envelope')

    library_space = webster_library.building.get_floor_by_number(LIBRARY_FLOOR).
    get_rooms({'room_type': 'Library'})

    envelope = webster_library.building.get_envelope_by_name('LB Envelope')
    ff_north_covers = envelope.get_covers({'floor_number': 0,
                        'building_orientation': BuildingOrientation.SOUTH.value})

    bishop_roof_covers = envelope.get_covers({'floor_number': 6,
                            'building_orientation': BuildingOrientation.TOP.value})

    glass_layers = ff_north_covers[COVER_INDEX].get_layers({'material.
    material_type': MaterialType.GLASS.value})

    print(bishop_roof_covers)
    print(glass_layers)

```

```
31      envelope_obj     = EnvelopeAnalysis(ff_north_covers)
32      cover_u_values   = envelope_obj.calculate_envelope_u_values()
33      integrity_checks = envelope_obj.check_integrity()
34      total_area       = envelope_obj.calculate_total_cover_area()
35
36      print(total_area)
37      print(integrity_checks)
38      print(cover_u_values)
```

Listing 3.1: Code snippets demonstrating the use of the LB Building model by a client application to compute U-values, total area and perform integrity checks.

The implementation of **MetamEnTh** leverages object-oriented principles, encapsulating methods that dynamically act upon and update the state of class objects. This approach directly addresses the limitations of static semantic ontologies. For example, methods such as `add_transducer` enforce run-time data validation, ensuring that only specific sensor types can be assigned to spatial entities (e.g., rooms), thereby maintaining model integrity. Furthermore, clients such as `EnvelopAnalysis` can operate directly on the dynamic, memory-resident **MetamEnTh** model, as illustrated in Listing 3.1, to perform computations such as U-value analysis of the building envelope. These capabilities collectively demonstrate the dynamic nature of **MetamEnTh** models, specifically addressing L_7 (Inability to Represent System Behaviours) and L_8 (Static Artefacts and Limited Interactivity). Chapter 5 will further detail how **MetamEnTh** models the dynamic behaviour of MEP entities for integration with control logic.

Expert Verification and Feedback We conducted three separate feedback sessions, each lasting 30–45 minutes, with three experts to evaluate the practical use cases. The first expert holds a PhD and has over 20 years of experience in civil engineering, architecture, and building science, with a focus on project management in the building sector and research in urban energy modelling. The second expert, also with more than 20 years of experience—16 of which are in energy consultancy, engineering, and research—is the current lead for Zero Carbon Communities at Concordia's Next Generation Cities Institute. The third expert is a final-year PhD candidate at Concordia's Loyola Sustainability Research Centre, working on retrofitting strategies for building façades.

All three professionals were already familiar with MetamEnTh and had contributed to its development. During these sessions, we demonstrated the practical use cases and collected their feedback, which is summarised below:

- **Utility Methods**: The experts recommended incorporating generic utility methods to support the development of energy efficiency solutions. These methods would provide commonly required functionalities, helping users streamline their workflows and improve usability across various energy modelling scenarios. We plan to collaborate with researchers and practitioners to identify and implement these utility methods in future iterations of MetamEnTh.

- **Conditional Objects in Control Integration**: Our initial implementation treated variables used in control strategies (e.g., occupancy) as setpoints. However, the experts clarified that the term **setpoint** should be reserved for parameters that can be directly adjusted to influence indoor environmental conditions—such as temperature or carbon dioxide concentration. Occupancy, in contrast, reflects a condition but is not itself a controllable variable in most BMSs. Similarly, while CO_2 levels can be indirectly influenced by ventilation control, they are not typically adjusted via a CO_2 setpoint. Based on this feedback, we revised our control logic interface to distinguish between controllable setpoints and conditional variables by introducing **conditional objects**.

- **Occupancy Modelling**: Although MetamEnTh currently supports general space occupancy modelling, the experts highlighted the need for more granular representation. Specifically, they recommended adding classes to support occupancy profiles and sub-space assignments (e.g., desks or workstations within a room). We plan to extend MetamEnTh to accommodate these detailed occupancy modelling requirements in future releases.

Following the review of existing ontologies, the needs assessment that informed the requirements for an alternative representation system, and the subsequent design, implementation, and both practical and empirical evaluation of **MetamEnTh**, we provide a comprehensive answer to our second research question, **RQ2**: How can an object-oriented metamodel represent mechanical, electrical, and plumbing (MEP) systems and their complex interactions?

An object-oriented metamodel defines concrete classes with predefined properties, behaviours, and relationships. It incorporates built-in validation rules and enforceable constraints to ensure the creation of accurate and easily verifiable models. By leveraging design patterns such as the Visitor, Strategy, Factory, Observer and Decorator patterns, the metamodel promotes extensibility and modularity. Additionally, it produces memory-resident objects that can be programmatically manipulated, enabling the integration of control logic and deployment of interactive applications, such as digital twins.

3.4 Discussions

In this section, we discuss the approach and evaluation of **MetamEnTh**, focusing on how it addresses the research question through its implementation and assessment.

3.4.1 Approach

To inform the design of **MetamEnTh**, we reviewed existing ontologies focused on the operational phase of buildings to identify gaps in current modelling practices. Recognising the importance of grounding our solution in practical challenges, we conducted a needs assessment involving both researchers and practitioners in the building systems domain. Before this, we toured HVAC systems at Concordia University and examined commonly used Building Management Systems (BMS), including Enteliweb and Desigo. These efforts deepened our understanding of the complexity and variability of HVAC operations and illuminated the limitations of current ontologies.

We employed both surveys and interviews to gather requirements, combining standardised questions with open-ended discussions to capture a broad range of perspectives. The questions explored how practitioners model, maintain, and interpret relationships in building systems, as well as the challenges they face. Although the insights were valuable, recruiting participants proved difficult, yielding only 10 survey responses and 2 interviews. This small sample limits generalisability and underscores the importance of future studies with broader participation.

We opted for facility managers for the interviews because they are the primary stakeholders responsible for the day-to-day operation, monitoring, and optimisation of building systems. Their

practical experience offers critical insights into the limitations of current building management tools, the complexity of MEP systems, and the types of data and models required to support informed operational decision-making. Likewise, our survey included researchers, students, and practitioners to capture diverse perspectives and skill levels. Researchers contributed insights into theoretical modelling needs and alignment with existing ontologies and standards, while practitioners offered feedback on real-world applicability and usability in operational contexts. Including students ensured that we considered the learning curve of the approach for less experienced users.

We designed UML models to represent **MetamEnTh's** classes and their relationships, iteratively refining them through feedback and validation using three institutional HVAC systems: Varennes Library, Concordia's EV Building, and the MB Building. To address practical modelling challenges, we extended the metamodel to include additional HVAC entities and to better represent associations, such as ductwork.

The Python implementation of **MetamEnTh** implements design patterns such as Visitor, Strategy, Observer, Factory, and Decorator to ensure modularity, enforce constraints, and support extensibility. Validation logic was encoded within setter methods and supported by a utility class for complex checks. While this approach enhances correctness and prevents invalid configurations, it introduces a steeper learning curve, particularly for non-standard cases where users may need to extend classes or adjust constraint logic.

The current implementation supports only a subset of building subsystem entities, with occupancy and certain interactions yet to be modelled, limiting completeness compared to mature ontologies such as Brick and Project Haystack.

3.4.2 Evaluation

To assess the effectiveness of **MetamEnTh**, we conducted a comparative experiment with Brick, a mature ontology for building systems. Participants modelled an HVAC system adapted from Pritoni et al. (2021) using both tools. Results from paired t-tests and Wilcoxon tests showed that participants created slightly more accurate and detailed models with **MetamEnTh** (mean = 4.2 vs. 3.9 and 3.9 vs. 3.8, respectively). However, they reported higher difficulty using **MetamEnTh** (mean = 3.6 vs. 2.5; $p = 0.027$). While validation mechanisms improved model accuracy, they also

steepened the learning curve.

The small sample size limits the statistical power of the findings. Furthermore, discrepancies between intended and actual models—especially in HVAC relationships—highlight a need for more guided tooling or educational resources to help users navigate the modelling process.

We collected data from multiple sources, including architectural drawings, BMS interfaces, and Concordia's Archidata portal. Architectural drawings were manually examined to extract details about block covers—an intensive process that requires expert interpretation and is prone to human error. We extracted metadata into JSON, while historical CSV data was mapped to individual sensors. Additionally, we utilised web scraping to retrieve tabular building data from the Archidata portal, which was then structured with a custom Python middleware.

Although this approach enabled the creation of models integrating diverse data sources, it also highlighted the significant manual overhead involved. Automating data extraction workflows and improving access to BMS APIs are critical steps toward reducing user effort and enhancing data reliability. At present, limited access to BMS programmatic interfaces remains a key bottleneck in deploying and scaling **MetamEnTh**-based solutions.

3.5 Threats to Validity

We outline the possible threats to the validity of our findings.

3.5.1 Internal Validity

Participants may have had varying levels of familiarity with object-oriented programming and building subsystems, which could have influenced their ability to effectively use **MetamEnTh**. Although we provided documentation and examples, differences in prior experience may have affected task outcomes and perceived difficulty.

3.5.2 External Validity

Our study involved 10 participants, primarily from research settings. As such, the findings may not generalise to practitioners in industry or those from different geographic backgrounds.

3.5.3 Construct Validity

We defined model quality based on semantic correctness, relationship accuracy, and completeness. These definitions were operationalised using predefined criteria and domain validation rules. However, alternative criteria or definitions could influence the interpretation of accuracy and detail. The perceived difficulty was self-reported and may also have been influenced by individual comfort with programmatic modelling or the clarity of the provided instructions.

3.5.4 Conclusion Validity

Due to the small sample size, most differences between **MetamEnTh** and Brick were not statistically significant. Only the difference in perceived modelling difficulty reached statistical significance. Thus, while trends are observable, stronger conclusions require a larger and more diverse participant pool in future work.

3.6 Conclusion

In this chapter, we presented **MetamEnTh**, an object-oriented metamodel developed to represent the operational aspects of Mechanical, Electrical, and Plumbing (MEP) systems in buildings. This work addressed **RQ1**, which investigated the limitations of existing ontology-based approaches, and **RQ2**, which explored how an object-oriented paradigm can overcome these limitations to deliver practical advantages for representing, validating, and operationalising MEP models.

Our approach comprehensively addresses the diverse challenges we identified in ontologies, including L_1 ambiguity in defining entities; L_2 the potential for composing flawed entity relationships; L_3 absence of entities for ducts, pipes, and detailed spatial contexts; L_4 difficulty in validating entities and relationships; L_5 inconsistent data and structural semantics; L_6 complex and deeply nested inheritance hierarchies; L_7 inability to represent system behaviours; and L_8 static artefacts with limited interactivity.

MetamEnTh defines concrete classes with explicit properties, relationships, and behaviours, embedding validation rules and enforceable constraints to ensure correctness and semantic clarity.

Unlike RDF/OWL ontologies, which assume an open world and require external validation models, **MetamEnTh** provides a structured and self-contained modelling environment. Its memory-resident objects can be manipulated programmatically to model dynamic behaviours and integrate seamlessly with building management systems for implementing control strategies.

Through the needs assessment, a systematic review of existing ontologies, iterative UML-based design, and the development of **MetamEnTh**, we identified the limitations of current approaches and introduced a metamodel that leverages their strengths while overcoming their shortcomings.

Our empirical evaluation demonstrated that **MetamEnTh** improves modelling accuracy and expressiveness compared to Brick, while supporting richer representations of HVAC components and their relationships. In practical use cases, we also illustrated its capacity for a detailed modelling of built environments.

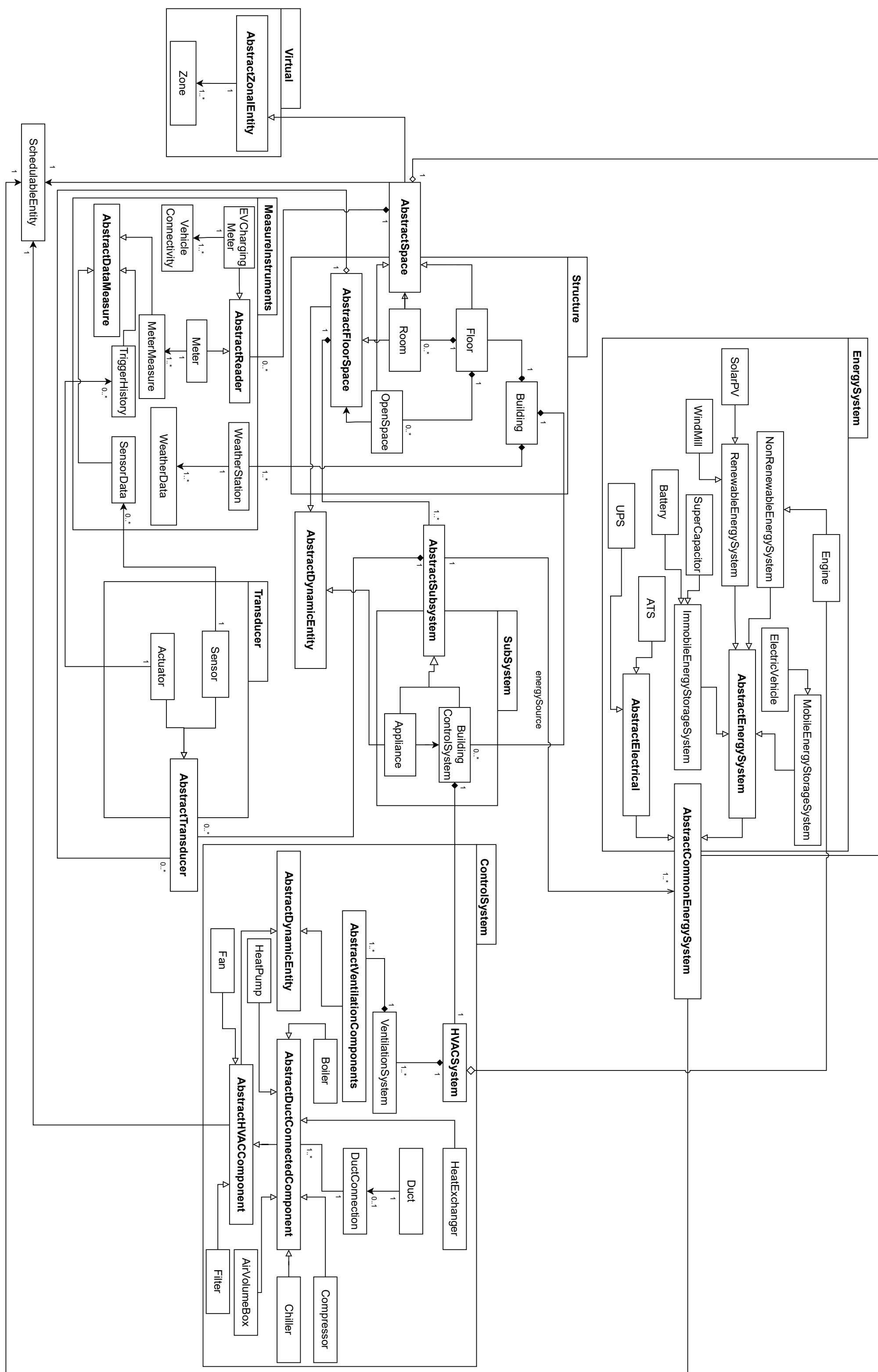


Figure 3.7: **MetamEnTh**, a complex low-level model instance showing domain-specific classes, their relationships and packages to which they belong

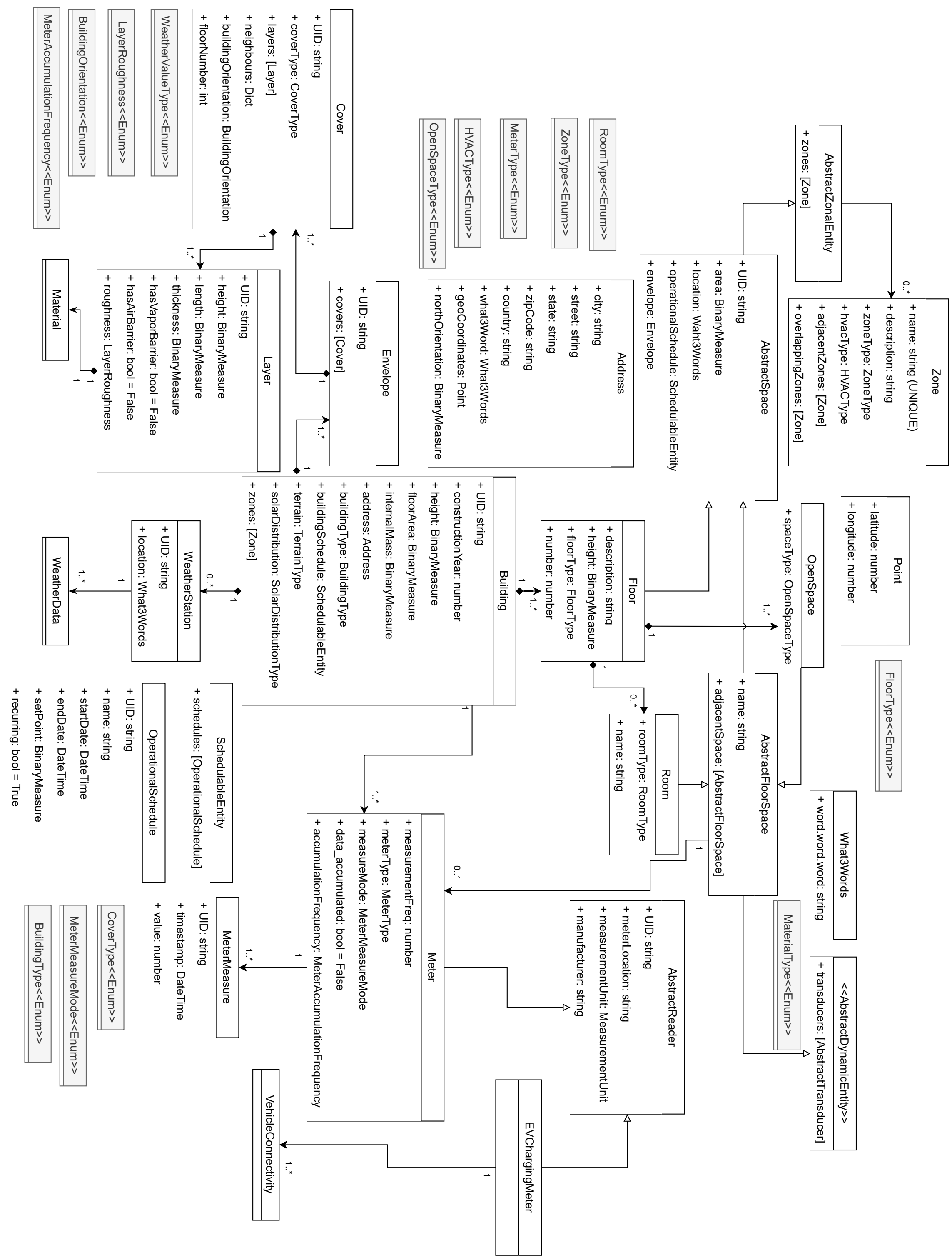


Figure 3.8: **MetamEnTh** Structure, showing the various classes, their properties, and relationships

Figure 3.10: **MetamEnTh** HVAC control system represented as a network of HVAC components connected through pipes and ducts. The design decision was taken in consultation with an expert to ensure that **MetamEnTh** can model the network of HVAC entities through ducts and pipes.

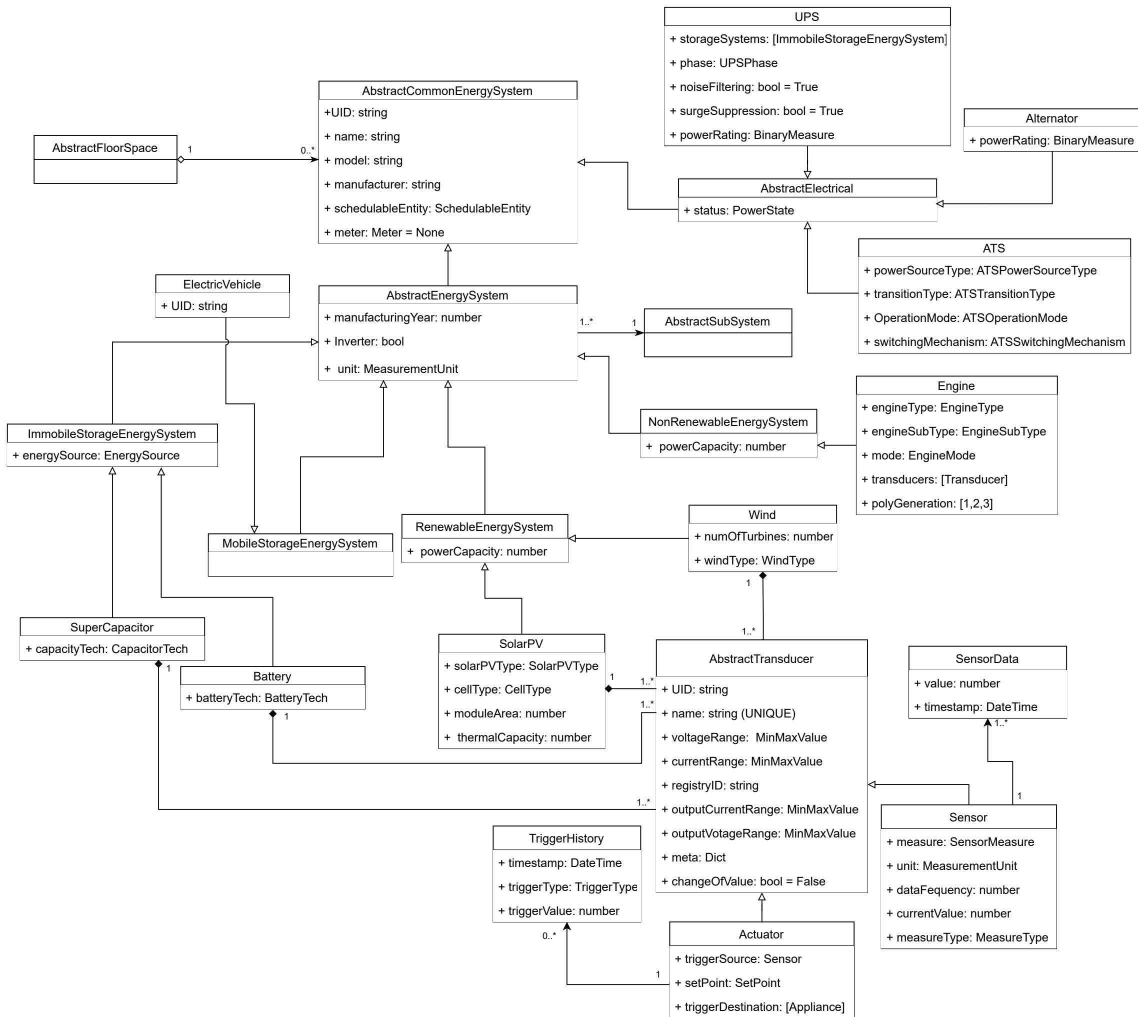


Figure 3.11: **MetamEnTh** Energy System showing the different energy systems, how they relate with themselves and with spatial entities (**AbstractFloorSpace**), **AbstractSubSystem** and **AbstractTransducer**

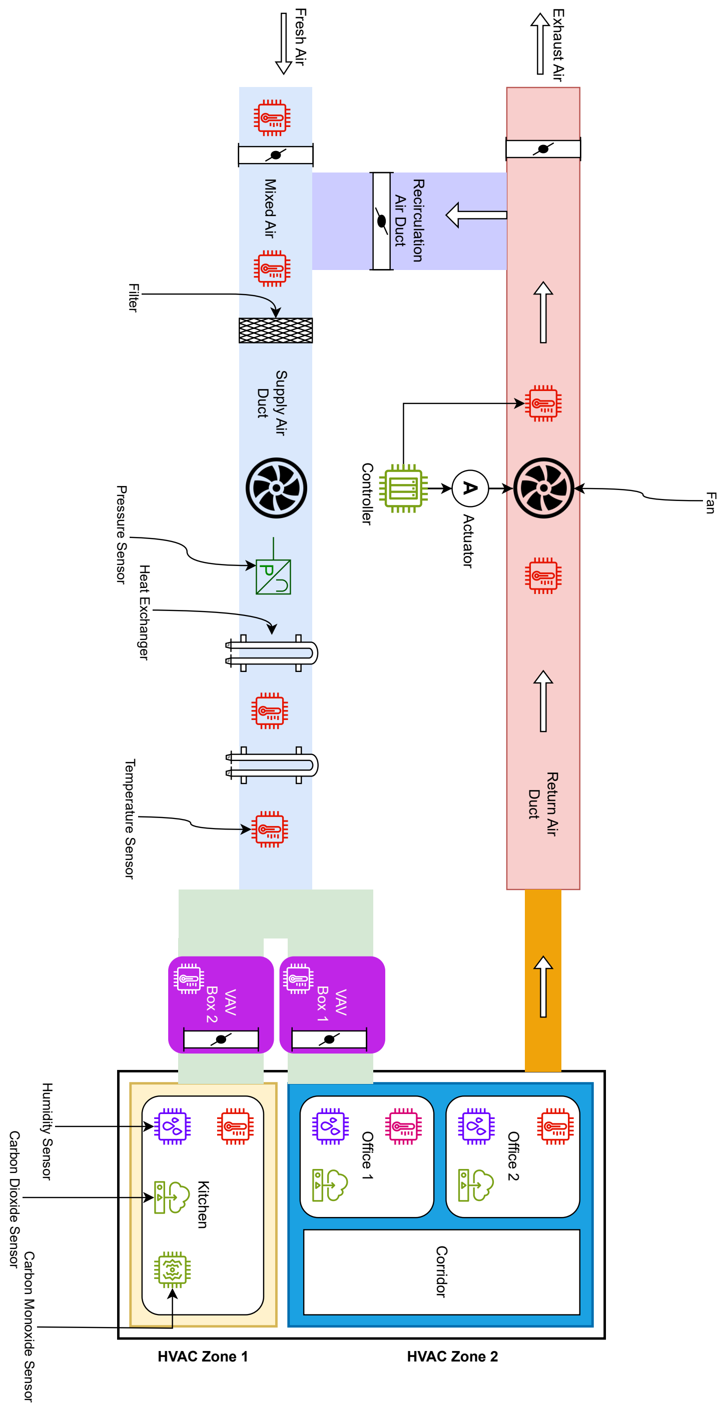


Figure 3.14: An HVAC system showing ventilation ducts, HVAC components, and airflow to rooms and open spaces, highlighting the distribution of conditioned air throughout the building for optimal climate control.

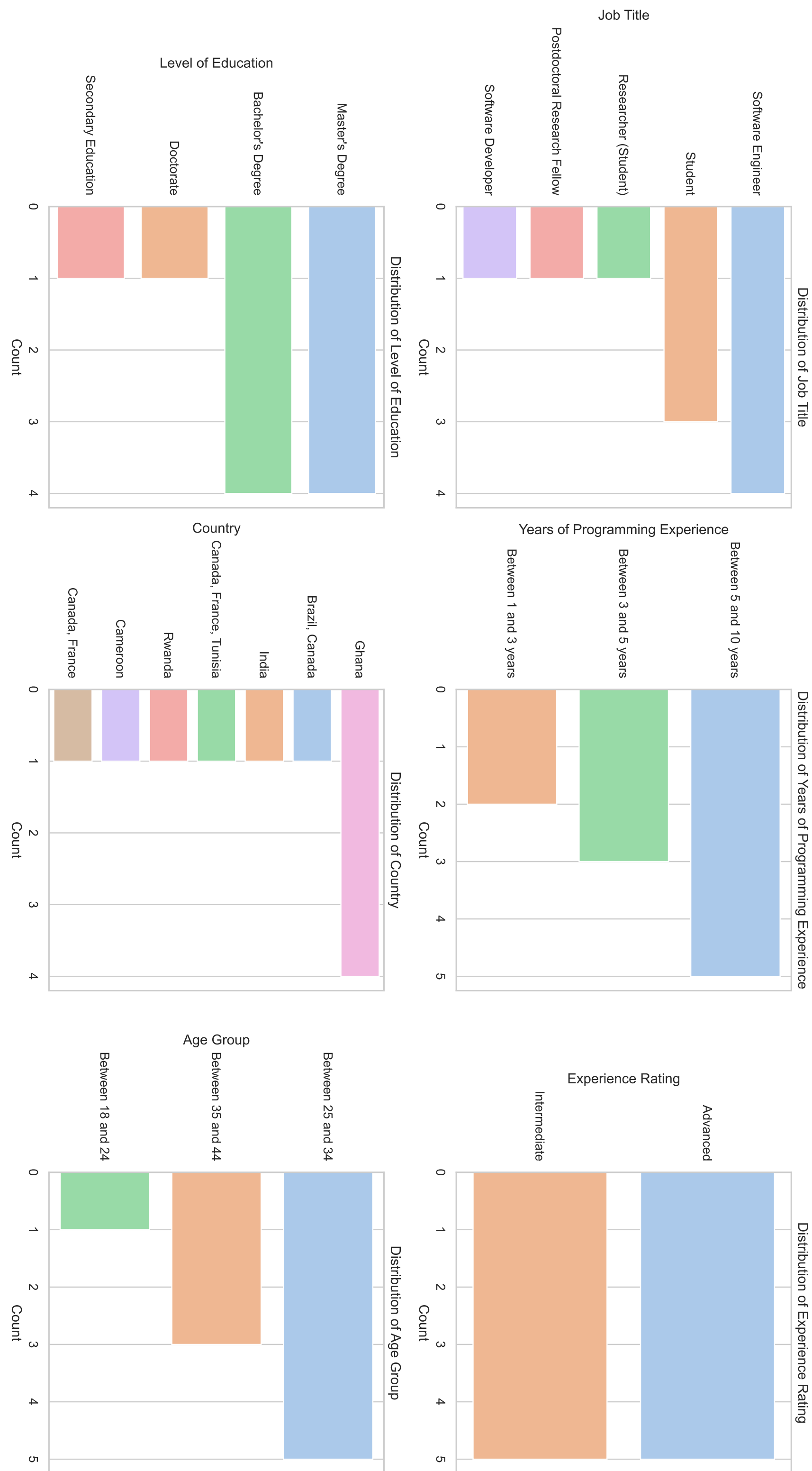


Figure 3.15: Demographic distribution of experiment participants by job title, years of programming, experience rating, level of education, country and age group

Chapter 4

Interoperable Data Access for MEP Systems

In this chapter, we address **RQ3** and its associated sub-research questions. Similar to Chapter 3, this chapter presents the research approach, results, discussion, limitations, and conclusion.

4.1 Introduction

The primary barrier to deploying intelligent and energy-efficient solutions across buildings is the lack of consistent, standardised access to interoperable building data. Different BMS vendors expose data through proprietary models and heterogeneous APIs. Even though BMS devices can communicate at the hardware level using protocols such as BACnet (BACnet, 2025), LONWorks (WAGO, 2025), and Modbus (Schneider Electric, USA, 2013), higher-level programmatic access for third-party applications remains fragmented and inconsistent (Balaji et al., 2016). This limits the ability of researchers, developers, and facility managers to extract, integrate, and act upon building data in a unified way.

Many research initiatives (Balaji et al., 2016; BuildingSMART International, 2022; gbXML.org, 2000; Haystack, 2011) have proposed semantic models and ontologies to harmonise building data representations. While these approaches provide standardised structures for describing building

assets and measurements, access to the underlying BMS data remains essential to automate translation and synchronisation with these models. Unfortunately, as Schneider Electric emphasises in its white paper *Three Essential Elements of Next Generation Building Management Systems*, despite widespread claims of openness and interoperability, vendors have done little to realise consistent programmatic access for external applications (Donovan, 2020).

To address these challenges, this chapter makes two contributions. First, it proposes a five-layered architecture for accessing interoperable, semantically structured building data, building on insights from prior metamodeling and ontology-based approaches. As our second contribution and to address **RQ3.1**, we evaluate the Web APIs provided by five major BMS vendors—Metasys, Enteliweb, EcoStruxure Building Operation, Desigo, and Enterprise Building Integrator—to assess the endpoints they expose for data exchange. Together, these contributions address **RQ3** and its sub-questions by clarifying the technical barriers to achieving interoperable building data access:

- **RQ3**: How can practitioners and researchers access interoperable data of mechanical, electrical, and plumbing (MEP) systems in buildings?
    - **RQ3.1**: What capabilities do BMS APIs provide to interact with building systems and the data they generate?
    - **RQ3.2**: How can practitioners and researchers interact with building systems and their data in a standardised way?

4.2 Approach

To overcome the fragmentation in how building systems expose and manage operational data, this research adopts a dual approach. We propose a layered architecture that clarifies how different stakeholders—from control engineers to application developers—can access and interact with building data in a consistent, interoperable manner. We evaluate the capabilities of BMS APIs from leading vendors at the third layer of the proposed architecture to assess their effectiveness in supporting data access and integration. This evaluation highlights the types of interactions enabled by these APIs and their limitations in real-world scenarios.

4.2.1 Needs Assessment

Most commercial and residential buildings are inherently complex, comprising numerous spatial and MEP entities. Creating digital models of such buildings requires significant manual effort from practitioners, who must gather data from fragmented, often incompatible sources. As shown in Chapter 3, Section 3.3, and Subsection 3.3.2.3, this data collection process is tedious, error-prone, and time-consuming.

Compounding the challenge, even within a single building, data sources are typically heterogeneous—ranging from BMS software exports to floor plans, spreadsheets, and building documentation. This heterogeneity makes it difficult for users to process and unify the data into coherent, homogeneous representations required for accurate and reusable building models. The problem becomes even more acute when modelling multiple buildings managed by different BMS platforms, each with its own proprietary structure and semantics.

These challenges point to the critical need for an architecture that facilitates programmatic access to building systems and their data in a consistent, translatable format—regardless of the BMS vendor. Such an architecture would empower researchers and practitioners to retrieve standardised information needed for model instantiation, greatly reducing the reliance on manual data extraction.

Furthermore, eliminating manual data collection processes necessitates access to BMS interfaces (e.g., APIs). Although current APIs may not provide all data types—such as spatial hierarchies—they serve as a foundational layer for automated, scalable model instantiation. Understanding the capabilities and limitations of these APIs is essential for developing abstraction layers that harmonise BMS data and support integration with control logic.

This research, therefore, seeks to address these needs by evaluating the interfaces of major BMS vendors and proposing architectural mechanisms to enable interoperable data access across heterogeneous systems.

4.2.2 Interoperable Data Access of Building Systems

Most existing research related to this work has focused on integrating BMS dynamic data with Building Information Modelling (BIM) to support objectives such as improving building operations

and management, increasing occupant comfort, monitoring and controlling building systems, optimising space utilisation, and enhancing energy efficiency. However, a gap remains in the literature regarding how client applications and stakeholders can access consistent, comparable data across diverse buildings and heterogeneous systems.

To address this gap, we propose an architecture for information exchange between client applications and building systems that enables standardised access to building system data. This research describes the five layers of the architecture in detail and examines the interfaces that BMS platforms provide for data access and interoperability.

We demonstrate the feasibility of the approach by instantiating building models for the Varennes Library and Concordia University's Webster Library (LB Building) in Quebec, Canada. We further demonstrate how client applications can utilise this unified model for various purposes, including monitoring, control, and analysis.

In Chapter 2, Section 2.2, we identified limitations in existing research: current approaches to BMS integration remain fragmented and predominantly conceptual, and lack empirical validation in real-world systems. Moreover, most existing architectures are tailored to specific use cases and do not provide a generic framework or demonstrate how practitioners and researchers can access standardised BMS data through APIs. Our approach introduces a novel contribution by focusing on the standardised acquisition of BMS data through vendor-provided APIs, thereby enabling consistent, interoperable access to building information across heterogeneous systems.

In the proposed architecture depicted in Figure 4.1, building subsystems form the fifth layer. These systems and equipment are primarily connected through devices deployed at the automation level of the BMS architecture. The second component, which enables transmutable and standardised access to building data, comprises middleware components and the MetamEnTh metamodel, from which clients can generate interoperable building models. The middleware interfaces with APIs from different BMS vendors to facilitate data exchange between vendor-specific systems and the unified representation layer, forming a five-layer architecture.

We also discuss how data structured differently across diverse BMSs can be uniformly modelled and exposed through a consistent interface, enabling standardised access for a variety of data consumers and applications to facilitate the deployment of cross-building solutions.

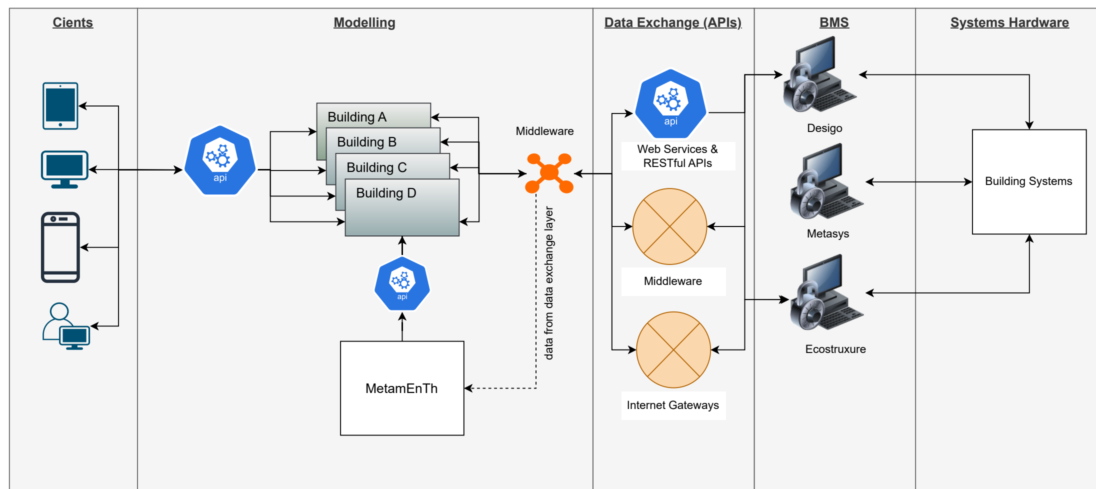


Figure 4.1: A five-layered architecture showing how different clients can access transmutable data from buildings and their systems

4.2.2.1 Clients

This layer represents third parties who must access data across multiple building systems for diverse purposes. The raw data collected from different building systems must be transformed and restructured into a model to be comparable and useful for applications. Clients could be researchers, software applications, and other interested stakeholders. These dissimilar clients may require building data for various purposes, including achieving energy efficiency, predicting building systems and occupant behaviour, remotely controlling building systems, informed decision-making and policy-making, and running simulations to optimise building management. A researcher working on occupancy profiling may need access to the presence data of different building occupants at various levels (rooms, floors, zones, etc.). Likewise, an application that monitors building energy usage may need access to the metering data of diverse building systems. An application that employs control strategies to manage the behaviour of building systems may require access to various setpoints, sensors, actuators, and controllers.

Bhattacharya et al. (2015) researched the data requirements of eight application categories with eighty-five (85) published applications. The ability to standardise the data accessed by client applications is essential for deploying research outputs and operational solutions across buildings with diverse, heterogeneous system representations.

4.2.2.2 Modelling

The **modelling** layer of the five-layered architecture is crucial in standardising data from the data exchange layer to clients. As indicated earlier, different BMS systems use different approaches to represent building systems and their data. The same BMS deployed across multiple buildings may have configurations that impact data representation, even when using the same vendor.

The modelling layer has three components, including an IoT middleware, application programming interfaces (APIs), and model(s) (of a building) created from a metamodel.

- **IoT middleware**: BMS vendors have various APIs to access the BMS in buildings. The IoT middleware abstracts the complexity and heterogeneity of BMS APIs, providing a standard, vendor-agnostic interface for integrating with and exchanging data with BMSs. The IoT middleware sends requests to the BMS through the data exchange layer and is responsible for creating building models from the metamodel and populating them with dynamic (real-time) and static data from the BMS.

- **APIs**: The APIs sit between building models instantiated from the metamodel and the IoT middleware. There are APIs between the IoT middleware and the metamodel to enable the middleware to create instances of the metamodel. In this arrangement, requests from clients are processed through the IoT middleware, which performs two functions: it either requests data (or executes an action) from a building model (if one exists) via the APIs or instantiates a model from the metamodel after querying the necessary data from the BMS through the data exchange layer. Additionally, the metamodel offers APIs for interacting with these instantiated models.

- **Metamodel and model(s)**: The metamodel is a prescriptive model with rules defining different building entities and their relationships (Yonglin et al., 2020). The metamodel helps create building models that abstract clients from diverse BMS representations of building systems, regardless of the underlying data structure, enabling clients to access vendor-agnostic, transmutable data from buildings. The proposed architecture employs **MetamEnTh**, introduced in Chapter 3, to model both the dynamic and static BMS data in this layer.

4.2.2.3 Data Exchange

The **data exchange layer** highlights the various APIs offered by BMS vendors to facilitate access to building systems and their associated data. Within the **modelling layer**, IoT middleware interacts with these APIs and abstracts vendor-specific differences by providing a uniform interface for data exchange. This abstraction enables seamless integration between building models and client applications. Facility managers often contact third-party service providers to enhance their BMS capabilities, such as enabling advanced data analytics for informed decision-making. These third-party tools (e.g., KMC Commander) may also expose supplementary APIs that interface with BMS data and systems, enabling additional functionalities and integrations not natively supported by the underlying BMS platform.

To understand current industry practices, we reviewed the APIs offered by five leading BMS vendors: Siemens, Schneider Electric, Honeywell, Johnson Controls, and Delta Controls. These vendors are consistently identified by reputable market research institutions as major industrial players with significant market shares (Allied Market Research, 2023; PR Newswire, 2024). For example, ABI Research (ABI Research, 2025) evaluated BMS solutions across ten key criteria.

Our analysis shows that these vendors typically offer three main types of APIs for integration and data exchange: Web services and RESTful APIs, Middleware platforms, and Internet gateways supporting lightweight protocols, such as MQTT.

Web Services and APIs. While both Web services and Web APIs at the BMS management level enable machine-to-machine communication, they differ in purpose and implementation. Web services typically use protocols such as SOAP, XML-RPC (XML-RPC Working Group, 2025) or JSON-RPC (JSON-RPC Working Group, 2010) to enable software applications to exchange data, and often require a client application or browser interface. In contrast, Web APIs expose application data and functionality over the Internet using protocols such as REST, GraphQL, or gRPC (Google & CNCF, 2025) and can be accessed directly via code without relying on user interfaces (Priyanka, 2025; Thomas Bush, 2023).

Schneider Electric offers Web services that support both SOAP and REST protocols, along with the EcoStruxure Energy API, which utilises GraphQL. In contrast, Johnson Controls offers RESTful

APIs for Metasys. While most BMS vendors make APIs available to access system data, not all offer public documentation. Some APIs are accessible online, while others require connecting through BMS software configured on a control network or subnet. In all cases, valid authorisation credentials are needed. Additionally, not all APIs grant full access to building systems: some expose only information about sensors, actuators, and their locations, without supporting control functions. Certain APIs also require users to purchase licenses from the vendor before using them.

Middleware. According to Campbell, Coulson, and Kounavis (1999), the primary function of middleware is to manage the complexity and heterogeneity of distributed systems, creating a simpler programming environment for developers. In BMS architectures, middleware typically sits between the automation and management layers, providing interfaces to exchange data with building systems. Because vendor APIs and services may not grant full access to all systems and data, third parties often build middleware that connects to core building systems via APIs (Facility Executive, 2014). This integration of third-party APIs enables data exposure via Web services or RESTful APIs built on top of the middleware. Integrating middleware can require additional hardware investments and vendor collaboration, particularly for legacy systems.

IoT Internet Gateway. Some BMS vendors provide gateways with MQTT support to transfer building system data to the cloud. These automation-level gateways are primarily hardware devices, such as specialised BACnet routers, that act as middleware within private networks. They support multiple protocols, including BACnet and TCP/IP, enabling seamless communication among devices from different vendors. Through IoT services from BMS vendors, clients can develop APIs to access building data and information. Gateways allow reading and writing data via Web services and APIs. Vendors use cloud platforms such as Microsoft Azure IoT Hub (Microsoft Azure, 2025), Amazon IoT Core (Amazon Web Services (AWS), 2025), IBM Watson IoT (IBM Cloud, 2025), and Mosquitto (Eclipse Foundation, 2025), where clients subscribe to building data sent to the cloud infrastructure.

Table 4.1 lists five BMS vendors and their integration APIs. Johnson Controls Metasys BMS offers a RESTful API and a .NET SDK. Metasys version 12.0 also supports MQTT for IoT data

Table 4.1: BMS and their integration APIs

APIs / BMS	**Web Services**	**Middleware**	**Internet Gateway**
Metasys (Johnson Controls)	REST API, .NET SDK	N/A	MQTT Broker
Enterprise Building Integrator (Honeywell)	EBI Web Services	N/A	MQTT Broker with Niagara IoT Framework (Tridium, 2025)
Enteliweb (Delta Controls)	Web Services	PHP Programming Module	MQTT Broker
EcoStruxure Building Operation & Enterprise Central (Schneider Electric)	EcoStruxure Web Services	Smart Connector SDK (Schneider Electric, 2025c)	MQTT Broker
Desigo CC & Optics (Siemens)	Building X Openness API	N/A	MQTT Broker

exchange (Johnson Controls, 2022), but we were unable to locate any documentation regarding middleware support for Metasys. Delta Controls' Enteliweb provides RESTful APIs and an MQTT broker. It includes PHP libraries for creating custom reports within Enteliweb, but does not permit data access or programming outside the platform. Siemens Desigo CC (Siemens, 2025d) and Desigo Optic (Siemens, 2025e) offer RESTful APIs and Internet gateways with MQTT for data exchange.

4.2.2.4 BMS Software

This layer contains the BMS software, which resides at the management level of a BMS architecture (BMS System, 2025). The BMS software integrates with the control networks of MEP subsystems (NIST, 2025). It serves as an interface for facility managers to monitor and control building systems, enabling them to achieve specific goals. While some vendors use a single software environment to control multiple building systems, others use multiple environments targeted to different building systems. BMS software is typically installed on computers that form part of the control network for various building systems. This arrangement provides the software with direct access to the building systems, aiding in their monitoring and control. Remote monitoring and control of building systems is only possible through the APIs described in the *Data Exchange* layer.

The BMS software provides an environment that enables users, such as facility managers, to create control programs. These programs can access and update control variables such as temperature set points, access controllers, sensors, and actuators to interact with the building's systems. With this functionality, facility managers could create programs that retrieve temperature values from room sensors and compare them against predefined minimum and maximum thresholds for the room. Subsequently, the program could control airflow to the room by adjusting a damper connected to a variable-air-volume (VAV) box installed to regulate airflow.

4.2.2.5 Systems Hardware

This layer comprises various MEP entities, including chillers, boilers, heat pumps, variable frequency drives (VFDs), fans, dampers, air handling units (AHUs), and filters, which work together to achieve multiple purposes, such as enhancing energy efficiency, promoting occupant comfort, and ensuring safety. These systems and equipment are mostly connected to automation devices, such as gateways and controllers, at the appropriate automation levels within a BMS.

Different vendor devices and equipment which use the same protocol(s) can communicate in this layer. It is thus not uncommon to see various devices (which use the same protocol) from disparate vendors configured together in a subnet. All vendors use standard protocols for their disparate devices and equipment in this layer, so interoperability between different vendor devices is usually not a problem, unlike the modelling layer. A retrofitting exercise can easily replace legacy devices with state-of-the-art ones to enhance the capabilities of building systems.

4.2.3 BMS APIs Capabilities Enabling Interoperable Access to BMS

In this section, we examine the role of BMS APIs within the *Data Exchange* layer of the proposed five-layered architecture. This layer facilitates communication between building systems and external applications, making it central to enabling smart building functionalities. BMS software provides facility managers (FMs) with powerful tools to monitor and control building operations. Below are some core capabilities typically offered by BMS software:

- Controlling and managing electrical and mechanical equipment, including lighting, HVAC

systems, water supply, fire safety, security systems, boilers, chillers, and elevators.

- Setting up alerts and alarms for early fault detection and timely resolution.
- Monitoring occupancy of building spaces in real time.
- Scheduling system operations, such as shutting down boilers during unoccupied hours.
- Generating reports from data collected by sensors, with analytics modules aiding data-driven decision-making.

Through web-based APIs, BMS vendors facilitate remote access to real-time system data and control functionalities, enabling facility managers and researchers to integrate advanced technologies such as predictive maintenance, optimisation algorithms, energy analytics, and digital twins for enhanced operational intelligence and decision support.

4.2.3.1 APIs Categorisations

We reviewed the online documentation of five BMS and compiled all API endpoints. Most documentation groups endpoints by managed resources; for example, Desigo's Building X Openness Structure API (Siemens, 2025b) includes endpoints for managing built environment locations. Following this, we categorised endpoints into 12 API functions: alarms and events, audits, equipment, network devices, objects, space, time-series data, energy, carbon dioxide emissions, assets, energy LEED scoring, and building energy modelling. We excluded user account management APIs and Desigo's Life Cycle Twin API (Siemens, 2025g), which manages Siemens' Lifecycle Twin (Siemens, 2025h) but is outside core BMS functions. We also excluded Enteliweb endpoints related to user interface management.

Metasys. Metasys offers a documented, publicly available RESTful API (Johnson Controls, 2025b) for reading and writing data and controlling Metasys objects (Johnson Controls, 2019). To access the API with a custom application, a Metasys user account—configured with API access, whether local or through Active Directory—is essential (Johnson Controls, 2025a).

With this API, users can subscribe to system events, manage alarms, retrieve audits and associated users, and discard audits. They can also access equipment served by other equipment and network devices, query equipment associated with specific spaces, and retrieve all equipment points. Additionally, the API enables the management of BACnet objects, including creation, attribute retrieval, and command execution. However, commanding Metasys objects via API calls requires a monitoring and commanding API license.

Users can retrieve various space configurations, including nested spaces and those served by network devices. Furthermore, the API supports reading time series data from all points within the BMS. We did not find API support for operations related to carbon dioxide emissions, asset management, LEED energy scoring, building energy modelling and management.

Enteliweb. Delta Controls' Enteliweb offers three features: energy analytics, engineering tools, and centralised facility management. The management module enables users to connect and control multiple buildings from a single interface (Delta Controls, 2025).

The Enteliweb API requires an account with privileges; it is documented but not publicly accessible. Through it, users can categorise, create, update, list, acknowledge, and delete alarms and events. They can also read and write property values to manage equipment and network devices, retrieve various node types, and control BACnet objects, including commanding HVAC systems.

The Enteliweb *Delta_Idotmulti API* lets users perform a single request affecting multiple objects via a *multi* operation. For example, users send a POST request specifying multiple resources for an action, and receive a unique ID for later GET requests to fetch data resulting from the initiated action on those resources.

Users can retrieve registered sites (buildings) but cannot access individual floors or rooms. The API provides trend logs and time-series data for BACnet objects, but does not support energy LEED scoring, carbon dioxide emissions retrieval, or building energy modelling.

EcoStruxure Building Operation. EcoStruxure Building Operation (Schneider Electric, 2025b) is a scalable, modular platform that manages multiple buildings through a single interface. Schneider Electric's enterprise server connects its systems with compliant third-party systems via Web

services. Unlike other vendors, Schneider Electric offers distinct Web APIs for different building management functions.

The Wiser Energy API provides data on energy consumption. The EcoStruxure Energy GraphQL API supports asset and alarm management, letting users retrieve asset details, service levels, maintenance records, and health indexes. Users can subscribe to health data, rank alarms, and access aggregated time-series data. The API also manages equipment and devices, including hierarchical parent-child data. Like other vendors, Schneider Electric's APIs manage BACnet objects and supply data on energy production, consumption, usage costs, and performance scores, as well as active energy delivered and received, active power, and gas utility data.

Building Operation includes an API for aggregated carbon dioxide emissions, depending on configured emission factors (Environment and Climate Change Canada, 2025). Another API creates building energy models to predict energy consumption. We did not find documentation on APIs for managing spaces within buildings.

Desigo. The Siemens Desigo system includes two main BMS platforms, Desigo CC and Desigo Optic, for managing high-performance buildings and automation systems. Siemens provides the Building X Openness API, which includes the Lifecycle Twin API (formerly the Ecodomus API).

The Building X Openness API provides endpoint groups, including Operations, Structure, Accounts, Security, Geometry, and Fire APIs. The API enables users to list alarms and events, manage devices and points, list network devices, and command systems by updating point values. They can also retrieve energy consumption, costs, and carbon dioxide emissions by space or meter. It also supports listing buildings, detailing rooms and floors, creating location hierarchies, managing addresses, and retrieving time-series data. The Geometry API lets users create and interact with 2D building floor plans (Siemens, 2025f).

Enterprise Building Integrator. Honeywell offers building management software for energy management, video surveillance, access control, and safety management (Honeywell, 2025). Honeywell Building Manager controls HVAC and lighting systems to maintain compliance, while Energy Manager analyses data to reduce energy use and emissions. EBI integrates Honeywell's software to manage multiple facilities through one interface. We found no publicly available documentation for Honeywell's BMS APIs. Honeywell indicates that EBI supports web services for third-party data exchange, but API details may only be accessible after installation or through support channels. Despite our efforts, we were unable to obtain further information; therefore, this paper relies solely on publicly available sources.

Each BMS API has strengths and limitations. Metasys and Enteliweb excel in alarms and device management but lack energy modelling. EcoStruxure leads in energy and emissions tracking, Desigo's Building X Openness API excels in building and device control, and Honeywell's EBI integrates multiple systems but lacks accessible API documentation.

Table 4.2 summarises the capabilities of the BMS APIs by the five vendors based on the 12 categories. From Table 4.2, none of the BMS vendors provides endpoints covering the 12 categories. Schneider Electric's EcoStruxure offers extensive APIs for 10 of the 12 categories, followed by Siemens Desigo, which covers 9 of the 12. Metasys and Enteliweb cover six categories, while EBI covers two categories.

Table 4.2: APIs of five (5) BMS vendors and their capabilities grouped into 12 categories

Criteria/BMS	**Metasys**	**Enteliweb**	**EcoStruxure**	**Desigo**	**EBI**
Alarms and Events	List events and alarms. Subscribe and unsubscribe to alarms. Perform batch operations on events. Update alarms (to discard or acknowledge them)	Get alarms and notifications. Create alarm details and categories. Get event notifications. Acknowledge event notifications. Get alarm details for event notifications	Get all alarms. Get (ranked and binned) alarm occurrences. Acknowledge alarms. Create and update alarm instances and occurrences. Manage events (CRUD). Get alarm transitions	Retrieve all events and their transition histories. Create, get and update alarm configurations. Delete alarm configuration.	N/A
Audits	List all audits. Subscribe and unsubscribe to audits. List audit users. Add audit annotations. Edit to discard audits	No endpoints for audits, however, there are endpoints to get all event and trend logs	N/A	N/A	N/A

APIs of five (5) BMS vendors and their capabilities grouped into 12 categories - continued from the previous page

Criteria/BMS	**Metasys**	**Enteliweb**	**EcoStruxure**	**Desigo**	**EBI**
Equipment	Retrieve equipment hosted by network devices. Retrieve equipment serving other equipment or spaces. Retrieve equipment points	Get all systems. Retrieve system properties. Write property values of a system	Get asset hierarchies with parent-child relationships. Get equipment associated with network devices. Get equipment of a particular type.	List all equipment. Create and update equipment. Delete equipment.	N/A
Network Devices	List network devices serving a space. List children of network devices. Delete offline child network devices	Get a list of node types. Get a list of protocol nodes. Get a list of network nodes. Get a list of device nodes	Get communication device hierarchies	List all devices. Create and update devices. List devices behind gateways.	N/A

APIs of five (5) BMS vendors and their capabilities grouped into 12 categories - continued from the previous page

Criteria/BMS	Metasys	Enteliweb	EcoStruxure	Desigo	EBI
Objects	Manage (CRUD) objects. List object attributes. List commands an object supports. Send commands to an object. Perform batch operations on objects. List object points	List object nodes. List objects of a device. Get object lists and properties. Get object property values. Write object properties (commands). Create and delete objects. Get the value of multiple object properties using multi	Retrieve a list of all objects. Get object properties. Update object properties (commands).	Get a list of all objects. Update object values (commands). Add object values. Create, get and update groups of points.	Get objects and properties

APIs of five (5) BMS vendors and their capabilities grouped into 12 categories - continued from the previous page

Criteria/BMS	**Metasys**	**Enteliweb**	**EcoStruxure**	**Desigo**	**EBI**
Space	Get all spaces. Get spaces served by network devices. Get spaces within other spaces	Get a list of all sites (buildings)	N/A	List buildings in a partition (e.g., campus). List building parts (e.g., rooms). Create location hierarchy (e.g., room on a floor). Get, update, and delete a location. Create, get, update and delete addresses. Get a 2D geometry of floors.	N/A
Time-series Data	Get data for objects. Get network device attributes that have time-series data. Get data for network device attributes	Get log records from trend log objects. Get a list of historical or trend log objects	Get device time-series data. Get aggregated time-series data. Get binned energy usage data (value and cost)	Get time-series data for points. Add point values.	Retrieve time-series data for points

APIs of five (5) BMS vendors and their capabilities grouped into 12 categories - continued from the previous page

Criteria/BMS	Metasys	Enteliweb	EcoStruxure	Desigo	EBI
Energy	N/A	N/A	Get energy production and consumption data. Get energy intensity data. Get energy usage cost data. Get meters and their measurements. Get active energy data. Get utility (gas, water flow rate) data	List all medium (space and equipment) consumption. Read consumption per location or meter. Read consumption cost per location or meter	N/A
Carbon Dioxide Emissions	N/A	N/A	Get aggregated carbon dioxide data (requires emission factor configuration)	Read emission data per location	N/A

APIs of five (5) BMS vendors and their capabilities grouped into 12 categories - continued from the previous page

Criteria/BMS	Metasys	Enteliweb	EcoStruxure	Desigo	EBI
Assets	N/A	N/A	Get asset details, including service level, maintenance, and health indexes. Get asset tickets. Create, get, update and delete asset ticket subscriptions. Create, get, update and delete asset health subscriptions. Create, get, update and delete site risk-level subscriptions	Manage (get and update) asset locations and their relationships	N/A
Energy LEED Scoring	N/A	N/A	Get a building performance score. Create performance scoring requests. Request performance scoring status	N/A	N/A
Energy Modelling	N/A	N/A	Create an energy model. Apply the existing model to predict building energy consumption. Assess model quality savings	N/A	N/A

4.2.3.2 Practical Usage and Assessment of Non-functional Aspects of BMS APIs

Because we lacked access to the BMS APIs of the five vendors, we mocked selected API endpoints (Peter Yefi, 2025a) from Enteliweb, Desigo, and Metasys to demonstrate their practical use in Postman. We then evaluated the (mocked) BMS APIs on four criteria beyond their functional capabilities—-standardisation, error handling, security, and interoperability. We selected these criteria for the following reasons:

- **Standardisation** (Bogner, Kotstein, & Pfaff, 2023): This criterion focuses on adherence to best practices in using standard HTTP status codes and verbs in APIs. Adhering to these standards is crucial for maintaining interoperability with other systems and ensuring consistent communication (Massé, 2011).
- **Error Handling** (Aué, Aniche, Lobbezoo, & van Deursen, 2018): We evaluate how the BMS APIs communicate errors to users. While this overlaps with standardisation, we selected it for a more detailed analysis due to its significant impact on the robustness and reliability of applications utilising BMS APIs (Brian Mulloy, 2011).
- **Security** (Guo, Xiao, Liu, & Zhuge, 2023): We assess the authorisation and authentication aspects of the APIs and their conformance to industry best practices. This evaluation is essential for protecting data and preventing unauthorised access.
- **Interoperability** (Borgogno & Colangelo, 2019): We examine the differences in data representation across various APIs and their potential impact on integration with third-party systems. Effective interoperability is key to seamless integration and efficient system operation.

4.2.3.3 Standardisation

We selected some endpoints that cover the unique HTTP verbs and status codes used in our mock-up implementation and across the BMS APIs. The selected endpoints perform CRUD operations for each vendor API.

HTTP Verb Usage: Our analysis reveals that all three BMS APIs utilise standard HTTP verbs for the majority of operations. The *POST* verb is consistently employed to create resources, *GET* for read operations, and *DELETE* for resource deletion. However, discrepancies arise in resource updates. For example, the Enteliweb API uses the *PUT* verb for partial updates and full resource replacements. The *write value to object property* endpoint, which updates BACnet standard properties, employs *PUT* rather than the more appropriate *PATCH* verb. Furthermore, the Enteliweb 4.24 documentation does not reference the use of *PATCH*. Similarly, Metasys uses *PUT* for its *send command* endpoint, even when it updates only specific properties of a resource.

In contrast, Building X Openness adheres to HTTP standards by differentiating between *PUT* and *PATCH*. For example, its endpoint for updating set points uses *PATCH* for partial updates.

HTTP Status Code Usage and Associated Responses: In contrast to HTTP verb usage, there are significant differences in HTTP status code usage and their associated responses among the APIs. While all three APIs return the status code 200 for successful read operations, variations exist in their handling of other operations:

- **Desigo Building X Openness:**
  - Uses status codes 200, 202, and 204 for *PATCH* operations, even for synchronous requests, creating inconsistencies.
  - Consistently returns status code 201 for *POST* requests, including a link to the created resource. However, the response does not include the *Location* header.
  - Employs status code 204 for *DELETE* operations with no response body, in conformance with best practice.
- **Enteliweb:**
  - Consistently returns status code 200 for both *PUT* and *POST* operations, including resource creation. However, the response sometimes includes an object for the created or updated resource.
  - Often returns a generic message object containing *error* and *errorText* fields, set to *1* and *OK*, respectively, for successful operations.

- Uses the non-standard status code 203 for *DELETE* operations, accompanied by a message object that conveys operation details.

- **Metasys:**
  - Returns status code 200 (instead of 201) for *POST* requests that create resources, including a *Location* header but omitting a representation of the created resource.
  - Employs status code 204 for *PATCH* operations, aligning with HTTP standards.
  - Uses both status codes 200 and 204 for *DELETE* operations, with no response body in either case, violating best practices for status code 200.

Table 4.3 shows the selected endpoints of the three BMS APIs, their HTTP verbs, responses, and status codes.

Error Handling: Enteliweb uses status codes such as 400, 401, 403, 404, and 501 to indicate unsuccessful requests. While most conform to HTTP standards, the API returns 403 for requests with invalid attributes, which is non-standard. It sometimes returns a 200 status code for failed operations, forcing users to inspect the response content to detect errors, violating the *NoRC200Error* rule (Bogner et al., 2023). The API uses the same object structure for errors and successes: for failures, the *error* attribute holds an undocumented number and *errorText* contains the message; for successes, *error* is "-1" and *errorText* is "OK." For all failed *DELETE* operations, the API mostly returns 403, regardless of whether the issue is an invalid attribute, a missing resource, or another problem better indicated by 400 or 404. Non-standard codes complicate failure diagnosis.

The Metasys API only documents responses for some unsuccessful *POST*, *PATCH*, and *DELETE* operations with status codes 400, 401, 403, 405 and 409, leaving failures for most of these requests undocumented. Additionally, the API does not document responses for unsuccessful *GET* and *PUT* requests, making it difficult and unpredictable for users to handle failures for these operations. Furthermore, the API uses the 405 status code for *PATCH* operations that attempt to discard an audit that has already been discarded, which contradicts the intended use of 405 (indicating that the method is not allowed for the requested resource).

Like Metasys and Enteliweb, the Desigo Building X Openness API does not document responses for all failed requests. It describes some 4xx and 5xx responses but does not clearly define the status codes or their usage, making it hard to verify compliance with HTTP standards. The API returns a standard error object with messages and codes for all failures. When we tested endpoints without an authorisation token, we received a 401 status code and could not proceed further. We submitted a request for sandbox credentials through the Siemens portal (Siemens, 2025i); however, we had not received a response at the time of this study.

4.2.3.4 Security

We assess the authorisation and authentication mechanisms of the BMS APIs. *Section W.3.2 OAuth* of the ASHRAE Addendum 135-2012am (ASHRAE, 2016) specifies using OAuth 2.0 bearer tokens and defines other security requirements for BMS servers. This assessment focuses solely on the security measures within the APIs and does not cover the servers' security setup.

Enteliweb: Users must provide their Enteliweb portal username and password for authentication to the API. Enteliweb's authentication module supports two methods for applications to present credentials for authentication, *HTTP Basic authentication* and *login and session cookie*.

- **HTTP Basic Authentication**: Applications and users include a base64-encoded username and password in the *Authorization* header of each request. This approach starts a new session for every request without using session cookies, timeouts, or Cross-Site Request Forgery (CSRF) tokens (OWASP, 2025).

- **Login and Session Cookie**: Users authenticate by logging into Enteliweb. The server returns a session cookie and a CSRF token, which the application must include in request headers. Enteliweb enforces a session timeout based on the user account settings and resets the timer with each request. If the timeout expires, the server ends the session and logs the application out. The server authorises requests based on the user's account group permissions.

Due to the lack of an OAuth implementation, the two authentication mechanisms in the Enteliweb API may not conform to the ASHRAE Addendum 135-2012am specification.

Metasys: Like Enteliweb, Metasys API provides two authentication methods: Bearer and API-Key authentications.

- **Bearer Authentication**: Users first obtain an access token (JSON Web Token, JWT) via the `/login` endpoint and include it in the *Authorization* header of subsequent requests (e.g., `Authorization: Bearer accessToken`). The token's expiration depends on the requesting account's profile.
- **API-Key Authentication**: Metasys uses API-Key authentication for streaming operations, since web browsers block custom headers when establishing Server-Sent Event (SSE) connections (Mozilla Contributors, 2025).

Metasys's bearer authentication may conform to Addendum 135-2012am; however, this depends on the specification's implementation, which must include the scope (what actions are permitted by the token) and claim (role, access level, and resource) definitions. The API Key implementation may not conform to the addendum.

Desigo Building X Openness: Unlike Metasys and Enteliweb, Siemens requires users with the *company administrator* role to create a machine user in the API Manager and use its client ID and secret to generate a *Bearer* token. To obtain a token, users call the authentication endpoint (Siemens, 2025a) with the *client ID* and *secret* in the request body. The request returns a JSON Web Token valid for 12 hours. Building X Openness bearer authentication may comply with the addendum, depending on how it defines the token's scope and claims.

4.2.3.5 Interoperability

The semantic representation of BMS objects follows the system's communication protocol. All three BMS evaluated here primarily use the BACnet protocol, which defines standard data formats for RESTful APIs in the ASHRAE 135-2012am Addendum. The addendum specifies XML and JSON data standards, including data types and required elements, and defines URI paths for locating resources (e.g., `<base-url>/.data/objects` to access objects). APIs that adhere to this standard enable interoperability across buildings with different BMS systems.

Table 4.3: Selected endpoints of the mocked BMS APIs, their HTTP verbs, responses, and HTTP status codes. The responses and status codes are limited to successful HTTP requests.

BMS	**API Category**	**Description**	**Endpoint**	**Response**	**Code**	**Verb**
Metasys	Objects	List all objects	/objects	Object list	200	POST
		Create object	/objects	Location header	200	POST
		Get object	/objects/{objectId}	Object	200	GET
		Send command	/objects/{objectId}/commands/{commandName}	Success	200	PUT
		Edit object	/objects/{objectId}	–	200	PATCH
		Delete object	/objects/{objectId}	–	204	DELETE
EnteliWEB	Delta Idot Bacnet	Get object list	/.bacnet/{siteName}/{deviceNumber}	Object list	200	GET
		Write object value	/.bacnet/{siteName}/{deviceNumber}/{objectType},{instance}/{propertyName}	Message object	200	PUT
		Delete object	/.bacnet/{siteName}/{deviceNumber}/{objectType},{instance}	Message object	203	DELETE
	Delta Idot Multi	Create multi	/.multi	Multi object	201	POST
		Write object value	/.multi	Multi object	200	POST
	Delta Idot System	Write system props	/systems/{systemName}	Property object	200	PUT
Desigo Building X Openness	Building Operations	Get all devices	/devices	Device list	200	GET
		Create device	/devices	Device object	201	POST
		Update device	/devices/{deviceId}	–	202	PATCH
		Update point	/points/{pointId}	–	204	PATCH
		Update point tags	/points/{pointId}/tags	–	204	PUT
	Building Structure	Delete location	/locations/{locationId}	–	204	DELETE

4.2.3.6 Built Environment Applications and BMS API Requirements

After evaluating the functional and non-functional capabilities of BMS APIs, we analyse how these APIs support a broad spectrum of building applications. Specifically, we assess which categories of BMS API functionality are essential for enabling representative application scenarios.

This analysis builds upon the work of Bhattacharya et al. (2015), who categorised 85 building applications into eight distinct groups. We refine these into seven application groups to better reflect their relevance to modern building operations. Specifically, we remove the *Web Display* category, as its functionality can be inherently integrated into the other seven groups (e.g., through dashboards for energy monitoring or occupancy visualisation). This refinement enables us to assess more clearly the alignment between BMS API capabilities and the application requirements most relevant to today's buildings.

In the following paragraphs, we define and describe each of the seven application categories, mapping them to the specific types of BMS API capabilities required to support them effectively. This mapping allows us to evaluate the readiness of existing BMS APIs to support a diverse and practical range of building applications.

Occupancy Modelling. It refers to the prediction and representation of human presence in indoor spaces, which is vital for optimising building energy systems. Approaches vary from correlating arrival and departure times across different days to recognising patterns in user mobility traces to forecast occupancy based on current location or habitual behaviours. These models are essential for enabling responsive HVAC and lighting control systems (Kleiminger, Santini, & Mattern, 2014).

Energy Apportionment. Energy apportionment focuses on associating energy consumption data with specific users or usage contexts within a shared environment. By learning individuals' unique appliance usage patterns and monitoring energy consumption via smart meters or distributed smart plugs, these systems identify which user is operating which appliance. This technique enables detailed energy profiling and supports personalised energy feedback and cost attribution (Garnier-Moiroux, Silveira, & Sheth, 2013).

Model Predictive Control (MPC). It is an advanced control strategy that utilises predictive models of building dynamics, along with forecasts of external disturbances such as weather or occupancy, to determine an optimal control trajectory. The goal is to balance competing objectives such as maintaining occupant comfort and minimising energy consumption or cost (DQ, Rawlings JB Mayne, 2009; Sturzenegger, Gyalistras, Morari, & Smith, 2012).

Participatory Feedback. Participatory feedback systems incorporate real-time occupant input to enhance comfort and system responsiveness. Unlike traditional HVAC systems that operate on fixed rules or setpoints, participatory systems collect feedback—either explicitly or passively—from users regarding their thermal preferences. By integrating this feedback into control algorithms and connecting with BMS infrastructure, such systems dynamically adjust temperature setpoints to better align with occupant comfort levels (Hang-yat & Wang, 2013).

Fault Detection and Diagnosis (FDD). FDD techniques identify inefficiencies or malfunctions in building systems by analysing energy consumption patterns. These systems often leverage the hierarchy of submeters within buildings to attribute unusual energy use to specific equipment or operational contexts. Modern approaches utilise statistical models to assess the extent of abnormal behaviour and distinguish between equipment faults and context-driven variations, such as weather or occupancy (Ploennigs, Chen, Schumann, & Brady, 2013).

Non-Intrusive Load Monitoring (NLIM). NLIM refers to techniques that infer the operation of individual electrical appliances from aggregate power consumption data, without the need for dedicated sensors on each device. Using signal processing and machine learning, NILM algorithms identify appliance-specific energy signatures, enabling detailed profiling of occupant behaviour, preferences, and routines. These profiles can be used for energy management, occupancy inference, or even behavioural insights, although with some implications for privacy (Liu et al., 2014; McLaughlin, McDaniel, & Aiello, 2011).

Demand Response (DR). DR enables buildings or devices to adjust their energy consumption patterns in response to signals from utilities or grid operators, typically to reduce demand during

peak periods or in response to dynamic pricing. This capability can be achieved through direct load control or by incentivising users to shift usage (Agarwal, Balaji, Dutta, Gupta, & Weng, 2011).

To support the functionality of the identified application categories, these systems must access both *metadata* (describing entities and their context) and *record data* (capturing real-time or historical values) associated with building components. Critical to this access is the representation of various **entity types and their relationships (ERs)** that define how physical and functional entities are interconnected within a building.

Below, we enumerate the key types of building entities and their relationships (ER) required across these applications, along with illustrative examples for each:

- **ER1 — Sensor-to-Spatial Location:** Captures where a sensor is physically located (e.g., a temperature sensor **in** Room 305).
- **ER2 — Sensor-to-HVAC Component or Appliance:** Associates a sensor with the specific HVAC component it monitors (e.g., a pressure sensor **in** a VAV box).
- **ER3 — Person-to-Spatial Location:** Maps occupants to spaces (e.g., a person **in** Room 41).
- **ER4 — Spatial Hierarchy:** Defines containment and hierarchy among spatial zones (e.g., Room 305 **on** the third floor).
- **ER5 — HVAC Component-to-Spatial Location:** Indicates where HVAC components or appliances are situated (e.g., a boiler **in** the mechanical room).
- **ER6 — HVAC Component-to-HVAC Component:** Describes connections between components (e.g., a VAV box **connected to** a duct).
- **ER7 — Meter-to-Sensor:** Links sensors to the meters that aggregate or relay their data (e.g., a temperature sensor **within** a thermostat).
- **ER8 — Meter-to-HVAC Component:** Relates a meter to the appliance or HVAC component it monitors (e.g., an energy meter **of** a boiler).
- **ER9 — Meter-to-Spatial Location:** Locates meters within the building structure (e.g., a power meter **in** Room 205).

- **ER10 — Appliance-to-Person:** Connects appliances to specific users (e.g., a desktop computer **assigned to** a staff member).

Table 4.4 presents the seven common building applications, along with their required entity relationships and the BMS API categories necessary to support them. Among these applications, **occupancy modelling** demands the most comprehensive data, requiring all entity relationships except ER7 and ER9. The API categories needed for occupancy modelling include *Equipment*, *Objects*, *Space*, and *Time-series*.

All five evaluated BMS vendors provide APIs for accessing *Objects* and *Time-series* data. However, **Honeywell's EBI** and **Schneider Electric's EcoStruxure** do not expose endpoints for *Space*, while **EBI** also lacks support for *Equipment*.

In contrast, all five BMS platforms offer the API categories necessary for **non-intrusive load monitoring** applications. For more complex applications such as **demand response**, **fault detection and diagnosis, participatory feedback**, and **model predictive control**, only **Metasys** and **Desigo** provide the full set of required APIs.

These findings contribute to answering **RQ3.1**: What capabilities do BMS APIs provide to interact with building systems and the data they generate?

4.2.3.7 Assessing Ontology-based Representation of Data Acquired Through BMS APIs

Given the heterogeneity of BMS API data, researchers and practitioners can benefit from standardised representations enabled by ontologies and metamodels. While **MetamEnTh** was used in the five-layered architecture to provide a standardised view of BMS data, ontologies are widely adopted for semantic data modelling (see Section 4.2.2). To explore their suitability, we assess how two operationally focused ontologies—Project Haystack and Brick—represent BMS API data across the 12 categories listed in Table 4.2. We also evaluate the extent to which their APIs enable access to standardised building data, and whether this access is comparable to that provided by native BMS APIs. This evaluation contributes to answering **RQ3.2**.

Brick and Haystack Representation of BMS API Data The BMS APIs offer capabilities that extend beyond the semantic representation of ontologies. Conversely, ontologies include semantic

Table 4.4: Common Building Applications, Building Entities and their Relationships, and API Requirements

Common Application	**Entity Relationship**	**API Category**
Occupancy Modelling	ER1, ER2, ER3, ER4, ER5, ER6, ER8, ER10	Equipment, Objects, Space, Time-series (occupancy)
Energy Apportionment	ER1, ER2, ER4, ER7, ER9	Objects, Space, Equipment, Time-series (metering data), Energy
Model Predictive Control	ER1, ER2, ER5, ER6	Objects, Space, Time-series, Equipment
Participatory Feedback	ER1, ER2, ER5, ER6	Objects, Space, Time-series, Equipment
Fault Detection and Diagnosis	ER1, ER2, ER5, ER6	Objects, Space, Time-series, Equipment
Non-Intrusive Load Monitoring	ER7, ER9	Time-series (energy), Objects
Demand Response	ER1, ER2, ER5, ER6	Objects, Space, Time-series, Equipment

entities for which the BMS APIs do not provide data. We focus on ontologies' ability to represent data from BMS APIs.

- **Audits**: Both Project Haystack and Brick provide classes (and tags for Project Haystack) to represent audit data directly linked to built environment systems. However, they do not have classes for process audits involving building systems. For example, the Metasys Audit API provides detailed information about the originating application, audit period, audit family, action types, users, equipment, spaces, audit annotations, and subscriptions. The `Agent` class in Brick, derived from RealEstateCore (Hammar et al., 2019), can partially capture the human-related aspects of audit data exposed by the Metasys API.

- **LEED energy scoring**: While both ontologies can represent energy data, they lack classes for LEED energy scoring, including the score, date of the score, type of score, and the score category as defined in the EcoStruxure Energy LEED Scoring API (Schneider Electric, 2025d).

- **Energy modelling**: Project Haystack and Brick can represent real-time energy data, but

they do not support representing data such as target cost, building signature, degree days method, day groups, mean absolute error, etc., resulting from energy models as specified by the EcoStruxure Building Energy Modelling API (Schneider Electric, 2025a).

Table 4.5 compares the classes and tags offered by Brick and Project Haystack for representing BMS API data. While Brick lacks support for historical time series data, Project Haystack includes tags for it. Both ontologies focus on semantically representing building systems and their interactions, not interactions with users or user-driven processes (Pritoni et al., 2021). Though extensible with custom classes and tags to model process audit data, frequent customisations undermine the goal of standardised data representation.

4.2.3.8 Project Haystack and BrickSchema APIs

Project Haystack's HTTP API facilitates data exchange over HTTP through defined operations (Project Haystack, 2025b). These operations specify URIs for POST and GET requests, returning responses in a grid format with rows and columns. While Zinc and JSON are the primary formats, other supported formats include Trio (Project Haystack, 2025d), CSV, Turtle (David Beckett and Tim Berners-Lee, 2014), JSON-LD (JSON-LD Community Group, 2025), and JSON V3 (Project Haystack, 2025c). Clients send grids in requests, and servers respond in kind, using the *err* grid marker to indicate errors. Servers may truncate data due to throttling or response size limits. The API provides endpoints to retrieve all data represented by the Haystack Tagging Ontology. The implication is that as long as data fetched via BMS APIs can be represented using the Haystack Tagging Ontology, the Haystack HTTP server provides endpoints to retrieve it. Unlike BMS API write requests that modify system objects, Haystack API write requests typically update tagging and metadata in the Haystack server's ontology without directly altering the underlying BMS.

Brick offers APIs for interacting with building models, focusing on reading and writing data through the BrickSchema Python package (Gabe Fierro and Bharathan Balaji, 2025) and the Brick Example Server (BrickSchema, 2025). The Python package, which includes a web server, enables users to manage, query, and convert Haystack-based models to Brick while adding Virtual Building Information System (VBIS) tags (VBIS, 2025). It supports creating RDF-based models from

scratch, validating them with SHACL, and querying them using SQL via an experimental object-relational mapping (ORM) module. Users can persist, version, and manage changes to Brick graphs in SQL databases, leveraging pre- and post-commit hooks to enhance workflows. The Brick Example Server provides RESTful APIs for managing Brick graphs, including querying and deleting time-series data, creating entities, and retrieving data using SQL or SPARQL queries.

Project Haystack and Brick cover nine out of 12 BMS API categories but lack support for audits, LEED energy scoring, and energy modelling. Both offer APIs to access standardised BMS data through their ontologies, enabling practitioners and researchers to interact with this data. Researchers and practitioners can model BMS data acquired through BMS APIs using ontologies and access the standardised ontology representation of the data through the ontology APIs.

Table 4.5: Comparison of Brick and Project Haystack Entities for Representing BMS Data Acquired via APIs Across 12 Functional Categories

API Category	**Brick**	**Project Haystack**
Alarms and events	Alarm class (which extends the Point class). Multiple Alarm types, e.g., Air_Alarm, C02_Alarm, Failure_Alarm, Cycle_Alarm, Humidity_Alarm, etc. Event class (from RealEstateCore)	Alarm tag for conditional notification. There isn't a standard event tag, however, a custom tag to represent events can be included
Audits	Does not possess classes for BMS audits (e.g., compliance audits) but may have classes to model audit data generated by building systems, e.g., energy usage data. Agent class could model building interactions with users, which is important in compliance audits	Does not possess classes for BMS audits (e.g., compliance audits) but may have classes to model audit data generated by building systems, e.g., energy usage data.

Comparison of Brick and Project Haystack Entities for Representing BMS Data Acquired via APIs Across 12 Functional Categories – continued from previous page

API Category	Brick	Project Haystack
Equipment	Equipment class with multiple equipment types, e.g., ICT_Equipment, Elevator, Fire_Safety_Equipment, Meter, PV_Panel, Lighting_Equipment, etc	Equip tag with multiple subtags, e.g., actuator, airHandlingEquip, airTerminalUnit, ates, battery, boiler, cable, chiller, circuit, coolingTower, etc.
Network Devices	ICT_Equipment class with multiple types, e.g., AudioVisualEquipment, Controller, Gateway, ICTHardware, ITRack, and SensorEquipment	Device, network, networking-device, networking-router, and networking-switch tags
Objects	Does not have a direct class for BACnet objects but has classes for the entities modelled as BACnet objects. For example, the Point class can model point objects like sensors and setpoints. The Command class, which extends the Point class, can model command objects	It does not have direct tags for BACnet objects, but it does have tags for the entities modelled as BACnet objects. For example, the point class can model data points such as sensors or actuators and cmd tags for commands.
Space	Space class with multiple types, e.g., Architecture, Region, and Wing. The Architecture class has subclasses such as Building, Room, Level, Site, OutdoorSpace, and Zone	The space tag has multiple subtags, such as dataCenter, floor, room, and zone-space. The Floor has subtags such as the ground-floor, roof-floor and subterranean-floor

Comparison of Brick and Project Haystack Entities for Representing BMS Data Acquired via APIs Across 12 Functional Categories – continued from previous page

API Category	Brick	Project Haystack
Time-series Data	Sensor classes (e.g. C02_Sensor) have attributes for lastKnownValue and an aggregate of data. It is not meant to store multiple time-series data	curVal tag for the current values of points. The curStatus tag captures the status of a point's value. currErr indicates an error condition. The his tag is used for historical data, and hisMode indicates the way historical data is collected
Energy	The Meter Class measures usage and consumption of electricity with properties like measuredPowerInput, measurePowerOutput, ratedCurrentOutput, ratedCurrentInput, ratedVoltageInput, ratedVoltageOutput, ratedPowerInput, ratedPowerOutput, etc.,	The meter, electric-meter, ac-electric-meter, dc-electric-meter, and equip tags model a kind of meter. They can be used in conjunction with point and unit tags to measure energy-related phenomena.
Carbon Dioxide Emissions	The Outside_Air_C02_Sensor class has properties like lastKnownValue and aggregate (for aggregated emissions)	CO2 and point tags with quantity tags such as concentration, flow, and level to measure emissions
Assets	Asset class (RealEstateCore) with multiple types, e.g., Equipment, Furniture, ArchitecturalAsset	equip tag to measure equipment assets. Does not have a dedicated tag for assets like Brick
LEED Energy Scoring	N/A	N/A
Energy Modelling	N/A	N/A

4.3 Evaluation

The evaluation of this work is twofold. First, we create building models using manually extracted actual BMS data from various sources, as described in Chapter 3, Section 3.3.2.3. Second, we demonstrate practical, real-time data access via Siemens' Desigo API, deployed at Concordia University's Webster Library. The following sections detail these evaluation steps.

4.3.1 Use Case: Varennes and Webster Libraries

In this use case, we demonstrate how built-environment data from the Varennes Library and the LB Building are represented in a **MetamEnTh** model to enable interoperable access. We show how different client applications can consume the standardised data without requiring changes to the underlying code. Model instantiation for both buildings follows the process described in Chapter 3, Section 3.3.2.3 on Page 60.

In the third layer of the five-layer architecture—responsible for data exchange through BMS APIs—we did not have access to the APIs of the two buildings, which operate on Siemens Insight and Delta Controls Enteliweb systems. Consequently, because direct access for static metadata via the BMS APIs was unavailable, we manually extracted the necessary data, following a procedure consistent with that outlined in Chapter 3, Section 3.3.2.3.

The core objective of deploying solutions based on abstractions of building systems, rather than vendor-specific BMS semantic representations, is to enable the interoperability and transferability of energy-efficient solutions across multiple buildings. This approach reduces the time and cost of customising solutions to the unique data representations of specific BMS platforms and buildings.

Beyond modelling the envelope, floors, rooms, and open spaces (described in Chapter 3), the LB Building model includes 80 sensors located on the third-floor library. We populated historical time-series data for two sensor types—temperature and occupancy—recorded at 5-minute intervals. The temperature data spans January to March 2024, and the occupancy data spans May to July 2024.

The Varennes Library model comprises 62 sensors, categorised into three primary types—temperature, carbon dioxide concentration, and humidity—along with corresponding historical data. This model also features a weather station with accumulated data for global horizontal irradiance,

diffuse horizontal irradiance, direct normal irradiance, and outside temperature. Furthermore, it includes an electricity meter with recorded consumption data.

4.3.1.1 Temperature Analysis Client Application

To demonstrate how a single client application can operate across both models, we implemented a simple `TemperatureAnalysis` class. It supports various analyses, including ARIMA forecasting (Rob J Hyndman and George Athanasopoulos, 2018), peak-load analysis, correlation of room temperatures, time-of-day variability, and pairwise temperature comparisons.

Listing 4.1 illustrates how the client performs time-of-day variability analysis using both models. Figure 4.2 presents the resulting output. In the code snippet, the `TemperatureAnalysis` client accepts the two building models as input without altering its internal logic, illustrating how a single application can operate across multiple building models.

```
1 # Varennes Library
2 from utils.building_creator import BuildingCreator
3 from varennes.vl_dynamic_data_reader import VLDynamicDataReader
4 from temperature_analysis import TemperatureAnalysis
5
6 varennes_library = BuildingCreator('varennes-library.json',
      VLDynamicDataReader())
7 varennes_library.create_building()
8 varennes_library.add_sensor_data()
9 data = varennes_library.building.get_floor_by_number(1).get_rooms()
10 tempAnalysis = TemperatureAnalysis(data)
11 tempAnalysis.time_of_day_variability()
12
13 # LB Building
14 from webster_library.lb_dynamic_data_reader import LBDynamicDataReader
15
16 LIBRARY_FLOOR = 3
17 webster_library = BuildingCreator('webster-library.json',
      LBDynamicDataReader(LIBRARY_FLOOR))
18 webster_library.create_building()
19 webster_library.add_sensor_data()
```

```
20 data = webster_library.building.get_floor_by_number(LIBRARY_FLOOR).get_rooms
       ({'room_type': 'Library'})
21 tempAnalysis = TemperatureAnalysis(data)
22 tempAnalysis.plot_room_correlation()
```

Listing 4.1: Snippets of Python code showing how two different building models (for buildings utilising different BMS) use the same client application

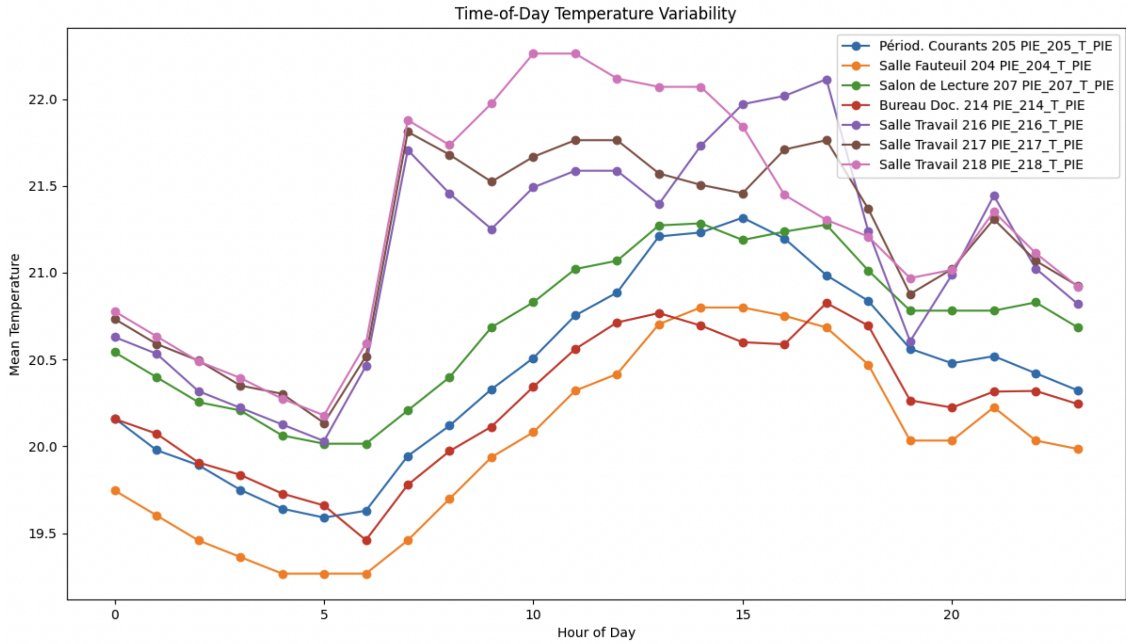
These use cases, although relying on manual data collection due to the lack of API access to building management systems, demonstrate how clients in the first layer of the proposed architecture can access standardised data from building systems (fifth layer) via an instantiated **MetamEnTh** model (second layer). Together, the proposed architecture and these practical use cases allow us to answer **RQ3.2**: How can practitioners and researchers interact with building systems and their data in a standardised way? *The proposed architecture enables practitioners and researchers to access standardised building system data by abstracting vendor-specific BMS representations through a metamodel and middleware integrated with BMS APIs*

4.3.2 Use Case: BMS API Real-time Data Access

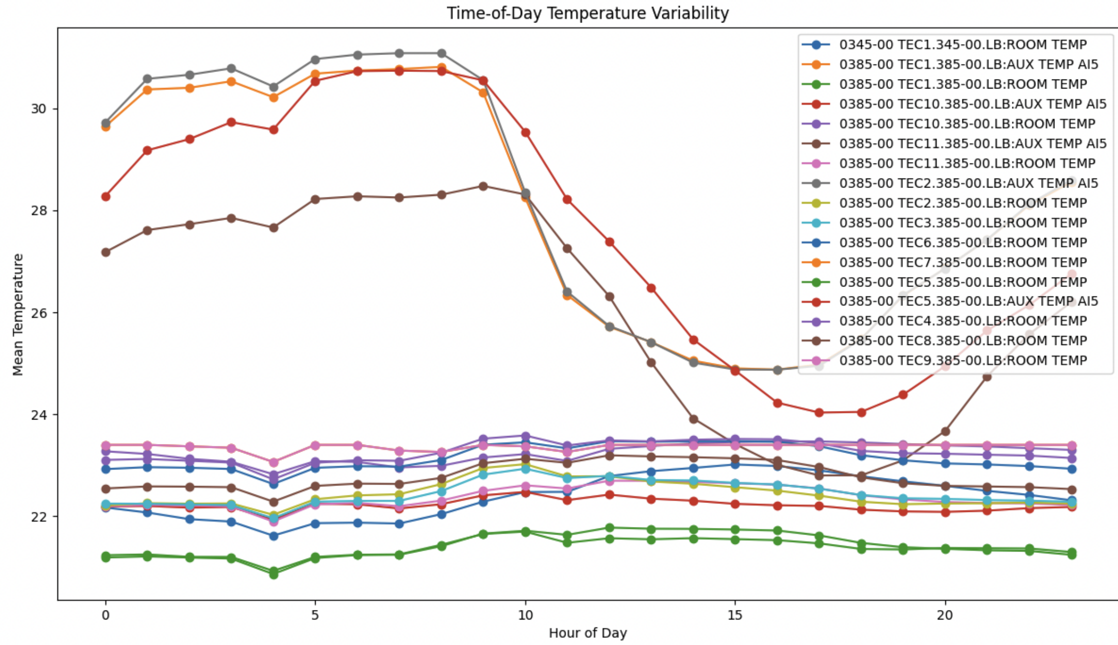
The third floor of the LB Building at Concordia University—home to the Webster Library on the Sir George Williams Campus—was part of a living lab project that deployed multiple environmental sensors. These include occupancy sensors, CO_2 concentration sensors, and temperature sensors. As part of the project, Siemens' Desigo system was configured with an API that allows authenticated users to read real-time sensor data.

Although the available API is limited to sensor data, the pattern for accessing different resources (e.g., setpoints or equipment status) follows a similar structure. Therefore, this section demonstrates the practical usage of the Desigo API to access real-time sensor data. This same API is also used in Chapter 5 to showcase how control logic can be integrated into building models.

The following subsections provide a step-by-step guide for accessing real-time sensor data through the API, including code examples where applicable.



(a) Varennes Library temperature variability



(b) LB Building temperature variability

Figure 4.2: Time of day temperature variability analysis based on models for the Varennes Library (Enteliweb BMS) and LB Building (Desigo BMS)

Token Generation. To access any endpoint on the Desigo API, users must authenticate using credentials provided by an administrator. These credentials include a username, password, and a grant type set to `password` for API access.

Once the credentials are supplied, an API call must be made to the authentication endpoint `/api/token` to obtain a bearer token, which is required for all subsequent API calls. For security reasons, the full URL of the API endpoint is omitted, as access is restricted to authorised users.

Listing 4.2 shows two methods we implemented to authenticate and retrieve the token using Python's `requests` package with HTTP *POST* requests. The API expects the `Content-Type` header to be set to `application/x-www-form-urlencoded`.

```
1 def _request_token(self, payload):
2     headers = {
3         "Content-Type": "application/x-www-form-urlencoded"
4     }
5     response = requests.post(f'{self._api_url}token', data=payload, headers=
       headers, verify='HVAC.CONCORDIA.CA.pem')
6     if response.status_code == 200:
7         token_info = response.json()
8         self._access_token = token_info.get("access_token")
9         expires_in = token_info.get("expires_in", 900)  # Default: 30 minutes
10        self._token_expiry_time = time.time() + expires_in - 60
11
12 def get_token(self):
13    """Fetches a new access token."""
14    payload = {
15        "username": self._username,
16        "password": self._password,
17        "grant_type": self._grant_type
18    }
19    self._request_token(payload)
```

Listing 4.2: Snippets of Python code show how we authenticated with Desigo API

Retrieving the Living Lab View. The Desigo API installed for the living lab on the third floor of Concordia's LB Building is structured to mirror how users interact with the graphical user interface. To access the sensor data, we performed two sequential `GET` requests. First, we retrieved a list of available system browsers and identified the one corresponding to the living lab. Then, using details extracted from that response—specifically the `SystemId`, `ViewId`, and `Designation`—we made a second `GET` request to retrieve the view associated with the living lab.

Listings 4.3 and 4.4 show the two `GET` requests and their sample responses. The method `make_authenticated_request`, implemented in the `BMSAPI` class, takes a single argument: the URL for the `GET` request.

```
1 # Get system browsers
2 system_browsers    = self._bms_api.make_authenticated_request('systembrowser')
3 living_lab_browser = system_browsers[1] #living lab system
4
5 # Get living lab view
6 system_id   = living_lab_browser.get('SystemId')
7 view_id     = living_lab_browser.get('ViewId')
8 designation = living_lab_browser.get('Designation')
9 base_uri    = f'systembrowser/{system_id}/{view_id}/'
10 lab_view    = self._bms_api.make_authenticated_request(f'{base_uri}{
       designation}')
```

Listing 4.3: Snippets of Python code show how we authenticated with Desigo API

```
1 # Response for getting system browsers
2 [
3     {
4         "SystemId": 1,
5         "ViewId": 16,
6         "Descriptor": "Application View",
7         "Designation": "System1.LivingLAB_View",
8         "Name": "LivingLAB_View",
9         "SystemName": "System1",
10        "ViewType": 0,
11        "_links": [
```

```
12          {
13              "Rel": "systembrowser",
14              "Href": "api/systembrowser/1/16/System1.LivingLAB_View",
15              "IsTemplated": false
16          }
17          ]
18      }
19  ]
20
21  # Truncated Response for getting the living lab view details
22  [
23      {
24          "HasChild": true,
25          "SystemId": 1,
26          "ViewId": 16,
27          "Name": "LivingLAB_View",
28          "Descriptor": "Project",
29          "Designation": "System1.LivingLAB_View:LivingLAB_View",
30          "ObjectId": "System1:GMS_Aggregator_5",
31          "Attributes": {
32              "Alias": "Project Alias"
33          },
34          "Location": "System1.LivingLAB_View:Project",
35          "_links": [
36          {
37              "Rel": "systembrowser",
38              "Href": "api/systembrowser/1/16/System1.LivingLAB_View%
        LivingLAB_View",
39              "IsTemplated": false
40          }
41          ]
42      }
43  ]
```

Listing 4.4: Sample responses for retrieving the Living Lab View

Extracting Indoor Air Quality Point. To access the indoor air quality (IAQ) sensor data, we issued two consecutive *GET* requests. First, we retrieved the root child node from the Living Lab view, which contains the point node associated with IAQ sensors. Using information from this initial response, we then made a second *GET* request to extract the specific point node for IAQ data. Listing 4.5 shows the code used for this process.

```
1 # Get root living lab view root child node
2 node = living_lab_view[0]['Designation']
3 living_lab_root_child = self._bms_api.make_authenticated_request(f'{base_uri}{
      node}')
4
5 # Get IAQ point node
6 point_node = living_lab_root_child[15]['Designation']
7 living_lab_point_node = self._bms_api.make_authenticated_request(f'{base_uri}{
      point_node}')
```

Listing 4.5: Snippets of Python code show how we authenticated with Desigo API

Retrieving Sensor Values. Each IAQ point contains multiple objects, each representing an individual sensor. To extract the current readings—for example, temperature and CO_2 concentration—we iterated over these objects and made individual *GET* requests using the corresponding `objectId` in the request URL. Listing 4.6 shows how we retrieved sensor values using a helper method, `_get_point_value`.

```
1 process_values = {}
2 point_mapping = {
3     'Temperature': 'temperature',
4     'CO2': 'carbon_dioxide'
5 }
6
7 for point in living_lab_point_node:
8     object_id  = point.get('ObjectId')
9     point_name = point.get('Name')
10    if not object_id or not point_name:
```

```
11          continue  # Skip if the object_id or point_name is missing
12      # Check if the point name matches one of the keys in point_mapping
13      for key, process_key in point_mapping.items():
14          if key in point_name:
15              value = self._get_point_value(object_id)
16              if value is not None:
17                  process_values[process_key] = value
18
19  def _get_point_value(self, object_id):
20      """
21      Helper method to fetch and extract the value of a point.
22      :param object_id: the unique ID of the point
23      """
24      POINT_INDEX=0
25      try:
26          value_arr = self._bms_api.make_authenticated_request(f'values/{
     object_id}')
27          print(f"Acquired sensor data: {value_arr[POINT_INDEX].get('Value')}")
28          return float(value_arr[POINT_INDEX].get('Value', {}).get('Value', 0))
29      except Exception as e:
30          # Log or handle error appropriately
31          print(f"Error fetching data for {object_id}: {e}")
32          return None
```

Listing 4.6: Snippets of Python code show how we authenticated with Desigo API

Interoperable Representation of Sensor Data. Building on the LB Building model we previously instantiated, we created two sensor instances—one for temperature and another for CO_2 concentration—to represent the real-time values retrieved through the Desigo BMS API. By embedding these values as `SensorData` within the **MetamEnTh** model, we demonstrate how heterogeneous BMS sensor data can be accessed and represented in a standardised, interoperable format. This step shows how researchers and practitioners can decouple sensor data from vendor-specific representations, enabling consistent use across diverse modelling and analytics tasks. Listing 4.7 shows how the sensors were instantiated and populated with their respective measurements.

```python
# Create sensors
co2_sensor = Sensor('TEC1.345-00.LB:ROOM CO2', SensorMeasure.CARBON_DIOXIDE,
    MeasurementUnit.PARTS_PER_MILLION,
                    SensorMeasureType.THERMO_COUPLE_TYPE_A, 900)
temp_sensor = Sensor('TEC1.345-00.LB:ROOM TEMP', SensorMeasure.TEMPERATURE,
    MeasurementUnit.DEGREE_CELSIUS,
                    SensorMeasureType.THERMO_COUPLE_TYPE_A, 900)
#Add sensors to the third floor
webster_library.building.get_floor_by_number(3).get_room_by_name('0345-00').
    add_transducer(co2_sensor)
webster_library.building.get_floor_by_number(3).get_room_by_name('0345-00').
    add_transducer(temp_sensor)
# Add temperature and carbon dioxide concentration data to building model
co2_sensor.add_data(SensorData(process_value['CO2']))
temp_sensor.add_data(SensorData(process_value['Temperature']))
```

Listing 4.7: Snippets of Python code show how we authenticated with Desigo API

4.4 Discussions

We now discuss the approach and its evaluation, and how it addresses the research questions.

4.4.1 Approach

Instantiating models for large, complex buildings typically requires users to retrieve data on building systems and their operations from multiple, disparate sources. This process is often tedious, labour-intensive, and error-prone. Moreover, the semantic heterogeneity and varying configurations of these data sources further complicate achieving interoperable access to building systems.

Motivated by the research gaps identified in the literature review (Chapter 2, Section 2.2) and the theoretical needs assessment (Section 4.2.1), this research investigates how practitioners and researchers can access building system data in a semantically consistent and interoperable manner.

To this end, we proposed a five-layer architecture that structures the pathway to achieving interoperable access to MEP subsystem data. Central to this architecture is the *Data Exchange* layer

(Layer 3), which decouples the top two layers—concerned with user access and modelling—from the bottom two layers comprising building systems (e.g., HVAC) and the BMS that operate them.

The *BMS Software* layer comprises various vendor systems, each with its own semantic conventions, even though they frequently communicate via standard protocols such as BACnet. Sitting atop this layer, the *Data Exchange* layer aggregates and exposes these heterogeneous data models through vendor-specific APIs.

The *Modelling* layer plays a crucial role in homogenising data retrieved from the *Data Exchange* layer into a memory-resident model that clients can access for various purposes, such as control strategy deployment, energy analysis, and indoor air quality monitoring.

Understanding the capabilities of APIs in the *Data Exchange* layer is crucial for researchers and practitioners to assess what types of data access and system control can be supported. We evaluated the Web APIs of five leading BMS vendors—Johnson Controls, Honeywell, Delta Controls, Siemens, and Schneider Electric—and categorised their capabilities into 12 functional groups.

Despite having user interface access to some of these systems, enabling API access proved difficult. BMS installations often come with APIs disabled, and facility managers—who primarily use GUI dashboards—rarely see the need to enable them, especially given the additional cost. This limited access not only impedes research but also hinders the scalable deployment of energy-efficient solutions across buildings. After navigating lengthy bureaucratic processes without success, we relied on vendor documentation and manually extracted data for our evaluation. To simulate functionality, we mocked endpoints for Desigo Building X Openness, EnteliWEB, and Metasys, and documented them using Postman collections and a Python server.

Our assessment focused on demonstrating API usage for key tasks rather than replicating all five vendor APIs. Thus, we highlight critical gaps related to error handling, standardisation, security, and interoperability. Most of the evaluated APIs do not fully comply with HTTP or ASHRAE Addendum standards. This lack of conformance—in URI structure, data formats, or security practices—raises interoperability challenges and inhibits transferability of solutions between buildings with different BMS. As a remedy, researchers and practitioners may adopt semantic models or ontologies to standardise API-extracted data before application-level integration.

To understand the practical utility of the APIs, we assessed them against seven common application domains (e.g., occupancy modelling, energy apportionment), adapted from Bhattacharya et al. (2015). We mapped each application's data needs to 12 functional API groups, helping clarify what each BMS can support in practice.

We also examined how ontologies—particularly Brick and Project Haystack—can represent data retrieved through APIs. While these semantic models offer structured representations, they may lack coverage for data related to energy audits, simulation models, and building certifications such as LEED (Abdullah, Cross, & Aksamija, 2014). Additionally, ontology APIs typically only support read/write operations within the model itself. Without integration with BMS control APIs (e.g., POST, PATCH, PUT), ontology updates—such as changing a setpoint—do not reflect in the actual BMS, risking divergence between the semantic model and system state.

Overall, our evaluation underscores the need for improved API openness and standardisation. It also demonstrates how combining object-oriented models with ontologies can provide a foundation for scalable and interoperable building applications.

4.4.2 Evaluation

The limited access to BMS APIs—apart from the Living Lab on the third floor of the LB Building—restricted our ability to demonstrate comprehensive interoperable data access across diverse building systems. Nonetheless, our practical use case involving real-time sensor data retrieval through the Desigo BMS API installed at Concordia's LB Building effectively illustrates how researchers and practitioners can access built environment data in a standardised and interoperable manner, abstracted from the heterogeneous semantic representations of individual BMS platforms.

This use case also highlights the inherent complexity of data retrieval, even for a single sensor. Acquiring the sensor values required a series of dependent API requests, beginning with navigating the system browser and view hierarchy, and culminating in endpoint queries for specific sensor objects. Such a multi-step process becomes even more cumbersome when dealing with APIs from multiple BMS vendors, each with its own unique structure and semantics. This complexity reinforces the need for a middleware—captured in the second layer of our proposed five-layered

architecture—that can harmonise these interactions and offer a unified interface to the user or application developer.

While we did not implement this middleware due to its inherent complexity—which warrants a dedicated research effort—it is important to highlight that such a system must integrate multiple built environment data sources beyond BMS APIs, including architectural and engineering drawings, BIM and IFC files, CAD models, and floor plans.

The models we instantiated for the LB Building and the Varennes Library further demonstrate the importance of interoperable data representation. The client application built on top of these models functioned seamlessly with both instances, without requiring any modifications to its codebase, reinforcing the value of standardised model representations in enabling reusable applications.

Together, these use cases—spanning both model instantiation and real-time data integration via the Desigo API—demonstrate the practicality and effectiveness of our proposed approach.

4.5 Threats to Validity

4.5.1 Construct Validity

One potential threat to construct validity is the reliance on API documentation rather than full programmatic access to all BMS systems. While documentation provides insights into available endpoints and their structure, it may not reflect the actual behaviour of deployed systems or support newer or deprecated features. Furthermore, because some APIs were mocked based on documentation, our assessments may not capture real-world constraints such as rate limiting, latency, or authentication challenges.

4.5.2 Internal Validity

The lack of direct access to multiple BMS installations limits our ability to empirically verify claims about API capabilities across vendors. Although the evaluation of Siemens' Desigo API in the Living Lab at Concordia University provides a concrete use case, conclusions drawn about other systems may be incomplete. Additionally, manual data extraction and mock endpoint implementations may introduce unintentional errors or overlook nuances present in live environments.

4.5.3 External Validity

Our study is primarily based on APIs from five industrial BMS vendors and specific to selected use cases. These systems may not be representative of all building types, regions, or software versions. Therefore, while the proposed architecture and evaluation offer general guidelines, their applicability across different types of buildings and BMS configurations may vary.

4.5.4 Conclusion Validity

The mappings between application requirements, API categories, and entity relationships are informed by literature and expert judgment. However, alternative categorisations or interpretations could lead to different conclusions.

4.6 Conclusion

We addressed **RQ3**—how practitioners and researchers can access interoperable data of MEP systems—by proposing a practical five-layered architecture and by empirically evaluating BMS vendor APIs. The five layers define the pathway from client applications to physical systems and clarify responsibilities at each stage:

(1) **Client layer (applications)**: researchers, tools, and dashboards that consume modelled data.

(2) **Modelling layer**: the metamodel (**MetamEnTh**) and model-managed representations that provide homogeneous, validated building models.

(3) **Data exchange layer**: the APIs and gateways vendors expose (the layer we assessed).

(4) **BMS software layer**: vendor management servers that implement proprietary data models and interfaces.

(5) **Systems hardware layer**: MEP subsystems.

We evaluated the APIs of five major BMS vendors against 12 feature groups and mapped those

capabilities to representative application needs. This analysis (answering **RQ3.1**) shows that vendors expose endpoints that enable the most common applications. However, no single vendor provides the full set of capabilities required by more advanced applications (e.g., demand response, predictive control, and LEED and audit scoring), and many useful endpoints remain gated behind additional licensing or configuration. We also assessed whether semantic ontologies (Brick, Project Haystack) can provide a standardised representation of the data acquired through BMS APIs (**RQ3.2**). While both offer rich schemas that help standardise representation, they lack entities for certain operational domains (e.g., audits, energy-score data). They cannot directly execute or synchronise control actions without additional integration layers.

To demonstrate a practical pathway from heterogeneous APIs to interoperable models, we showed how the modelling layer (**MetamEnTh**) instantiates API data and exposes a consistent interface for client applications and control strategies. This instantiation demonstrates the architecture's transmutability concept: clients operate against the same validated model regardless of vendor-specific details, facilitating portability and reuse. These results demonstrate the practicality of our approach, as we combined systematic API analysis, ontology assessment, and a working **MetamEnTh** instantiation to show how vendor-neutral workflows can function in practice.

Overall, the five-layered architecture clarifies where interoperability problems arise and where solutions must be applied: vendors must improve API coverage and conformance; model layers must provide validation and behavioural interfaces; and middleware must bridge semantic gaps. Our work provides researchers and practitioners with a structured methodology for evaluating BMS ecosystems and a practical metamodel, **MetamEnTh**, for making building models interoperable and portable across vendor systems.

In Chapter 5, we present the interfaces provided by **MetamEnTh** to enable seamless integration of BMS and IoT APIs—aligned with the proposed five—layer architecture—for control logic implementation using real-time sensor data in two buildings, one of which features a bidirectional data flow between the model and the physical system, forming a digital twin.

Chapter 5

Integrating Control Logic in Object-Oriented MEP Models

In this chapter, we address **RQ4** by presenting research focused on how our proposed object-oriented modelling approach integrates with control logic. Building on the design and implementation of **MetamEnTh** presented in Chapter 3, and the introduction of an architecture for accessing standardised data from MEP subsystems alongside the evaluation of BMS APIs to assess their capabilities for common building applications (Chapter 4), this chapter shifts focus to **MetamEnTh** interfaces that enable the integration of control logic with models, as delineated by the proposed five-layered architecture.

5.1 Introduction

Despite the presence of Building Management Systems (BMS) designed to optimise resource usage and maintain conducive indoor environmental conditions (IECs) for occupants, many commercial buildings continue to exhibit suboptimal energy performance. BMSs are designed to enable efficient operation of building systems, particularly HVAC, lighting, and other utilities, through configurable setpoints, schedules, and automated control logic. However, in practice, maintaining occupant comfort is often prioritised over energy efficiency.

For example, facility managers frequently configure HVAC systems with the primary goal of

ensuring thermal comfort, thereby overlooking energy-saving opportunities. As a result, energy performance becomes a secondary concern. This HVAC configuration pattern was particularly evident during the COVID-19 pandemic, when facility managers adjusted HVAC systems to increase ventilation rates to reduce the airborne transmission of the virus. These adjustments, i.e., turning off demand-controlled ventilation or increasing outside-air intake, resulted in significantly higher energy consumption. In many buildings, these settings remained unchanged even after the immediate public health threat had subsided, continuing to drive unnecessary energy use (Moghadam et al., 2023).

Facility managers typically interact with BMS via graphical user interfaces (GUIs) to configure system setpoints and monitor operations. Many also leverage built-in programming modules in BMS to develop simple control logic that automates responses to varying conditions. These scripts often compare process variables (DataPARC, 2025) with predefined setpoints to trigger actuators, such as dampers or boilers, thereby maintaining IECs.

Researchers have made significant efforts to develop and demonstrate advanced control strategies for deployment in buildings to improve energy efficiency and occupant comfort. Techniques such as Model Predictive Control (MPC) and Reinforcement Learning (RL) have shown substantial potential in optimising HVAC operations and reducing energy consumption (Afram & Janabi-Sharifi, 2014; Kim et al., 2022; Wei, Wang, & Zhu, 2017). However, despite these promising developments, the adoption of such strategies in industry remains limited. Commercial BMS generally lack the openness and interoperability needed to support the integration and deployment of these advanced control methods (Khabbazi et al., 2025). Researchers deploying advanced control strategies—often prototyped in programming languages such as Python—face considerable challenges, including acquiring the necessary data, developing accurate models, and creating APIs to actuate building systems. As a result, they often rely on ad hoc methods that are difficult to replicate across different BMS implementations. These steps often require custom integrations and demand expertise in both building subsystems and software engineering.

Because of this complexity and the high deployment overhead, most BMS continue to rely on basic on/off controllers or classical PID control loops (Geng & Geary, 1993), which require expert tuning and are often suboptimal in performance (Fütterer, 2017). Henze, Kircher, and Braun (2024)

identifies several barriers to adoption, including integration complexity, unreliable data acquisition, a lack of standardised frameworks, and the need for highly skilled personnel, which further hinder the translation of advanced control research into real-world building applications.

To address these challenges, this chapter makes two contributions. First, it proposes a method for integrating control logic into building models instantiated with **MetamEnTh** by leveraging object-oriented programming principles, including abstraction, encapsulation, and the dependency inversion principle. We provide a detailed explanation of how **MetamEnTh** enables the incorporation of on/off controls and PID loops through well-defined interfaces. Second, we demonstrate the practical application of this approach through two use cases: the integration of on/off controls in Concordia University's Webster Library and in a mixed-use (commercial and residential) building in Ludwigsburg, Germany. Each use case is implemented using distinct APIs, as proposed in the third layer of the five-layer architecture (Chapter 4, Page 81). Together, these contributions address **RQ4** by identifying the technical pathways and challenges associated with control logic integration in building systems:

- **RQ4**: How does an object-oriented modelling approach support the integration of control logic and promote broader adoption in practical energy-efficient applications?

5.2 Approach

We adopt a practical approach to propose object-oriented principles that facilitate the integration of control logic within building models. First, we define a set of abstractions that users must implement to embed control logic into **MetamEnTh** models, enabling real-time interaction with building systems via BMS and IoT APIs. This approach supports the creation of digital twins (Shahzad, Shafiq, Douglas, & Kassem, 2022) by replicating relevant aspects of the built environment. The interfaces specify specific variables within **MetamEnTh** that users must provide to enable seamless integration of control strategies.

Second, we demonstrate the feasibility and effectiveness of this approach by implementing an on/off control strategy using Siemens' BMS API deployed at Concordia University's Webster Library. We apply the same approach to implement an on/off control strategy for an IoT-based setup

in a mixed-use building in Ludwigsburg, Germany.

5.2.1 Needs Assessment

Advanced control strategies for building systems have demonstrated significant potential to improve energy efficiency and occupant comfort. However, their practical deployment remains constrained by several key challenges. Commercial BMs typically provide limited flexibility and interoperability, making it difficult to integrate sophisticated control logic developed in research environments (Henze et al., 2024; Khabbazi et al., 2025). This gap forces researchers and practitioners to invest substantial time and effort in bridging these technological divides.

Firstly, data acquisition poses a significant obstacle. Effective control strategies, such as MPC or RL, require high-quality, real-time data streams from diverse building subsystems (Afram & Janabi-Sharifi, 2014; Wei et al., 2017). In practice, BMSs utilise heterogeneous data formats and proprietary interfaces, which complicates consistent data retrieval and limits real-time model updating. Without standardised access, integrating control logic into operational systems is cumbersome.

Secondly, the architectural complexity of modern buildings demands flexible, extensible metamodels, such as **MetamEnTh**, that accurately represent mechanical, electrical, and plumbing (MEP) subsystems and their control mechanisms. Traditional BMS programming modules typically allow only basic control loops (e.g., on/off or PID) and are often vendor-specific, tightly coupled with their hardware. This limitation restricts the implementation of advanced algorithms and the adaptation of models across different buildings or platforms (Fütterer, 2017).

Thirdly, while ontologies are effective for semantically representing and abstracting heterogeneous building data, they are not inherently suited for integrating control logic (see Chapter 2, Section 2.1 for a discussion of their limitations). This limitation necessitates additional layers to bridge semantic models and control implementations, increasing complexity and hindering seamless integration of control logic.

To overcome these barriers, a unifying approach is needed—one that enables seamless integration of advanced control strategies into building models while facilitating bidirectional real-time interaction with operational systems. In the following sections, we describe our object-oriented approach for integrating control logic into building models and validate its effectiveness through

real-world use cases that demonstrate practical applicability.

5.2.2 Interfaces for Control Logic Integration

To enable seamless integration of user-defined control algorithms, this thesis specifies a set of well-defined interfaces (interfaces) that users must implement. These APIs establish the method signatures and required attributes through which control logic can interact with object-oriented models instantiated from **MetamEnTh**. By adhering to these APIs, users ensure that their algorithms depend on well-defined abstractions rather than specific model implementations, allowing high-level control logic to be decoupled from low-level model details and enabling the models to invoke and execute embedded logic at runtime. This section details each interface, its attributes, and the contractual obligations users must fulfil to integrate their control logic into the execution workflow.

5.2.2.1 AbstractControl

The `AbstractControl` interface in the `control` package serves as the root of the control logic inheritance hierarchy. It declares two essential abstract methods that users must implement, using the attributes provided by its constructor. These attributes define the contextual information and configuration parameters required for the control logic to operate effectively within instantiated models.

Attributes.

- `process_value_sensors: List[Sensor]`: Represents the sensor(s) providing the process variable (PV) for the control loop. In control theory, the PV—also known as the process value or process parameter—refers to the real-time measured quantity of a specific aspect of a process being monitored or regulated (DataPARC, 2025). Within the model, the first sensor in this list acts as the primary PV sensor, and its readings correspond to the condition targeted by the control logic. Any additional sensors in the list serve as *conditional sensors*, supplying supplementary data for the control algorithm's decision-making. Encapsulating PVs and related conditional inputs in the constructor (Wikipedia Contributors, 2025) ensures consistent execution of control logic.

- `actuator: Actuator`: The `actuator` is an instance of the `Actuator` class in **MetamEnTh**, responsible for executing control decisions, such as opening or closing a damper, or turning a boiler on or off. The `Actuator` constructor accepts a controllable `HVACComponent` (e.g., boiler, damper, fan) or an `Appliance` (e.g., smart plug), which it directly manipulates to implement the control action.

- `control_thresholds: List[ContinuousMeasure]`: The `control_thresholds` parameter is a list of `ContinuousMeasure` instances in **MetamEnTh** that specify the target values—such as setpoints—for the control strategy. These thresholds define the desired condition that the process variable should reach or remain within, typically as a range bounded by minimum and maximum values. For example, a user might specify a temperature range between 18 °C and 21 °C. The control logic then actuates systems to maintain the process value within this range. Within the list, the first element defines the primary setpoint associated with the `process_value_sensor`, while subsequent elements (if any) represent conditional thresholds that can influence control decisions.

- `run_duration: float`: This optional attribute allows users to specify the total duration, in hours, for which the control logic should remain active. When provided, the control strategy executes for the defined period before automatically terminating. If omitted (i.e., set to `None`), the control logic continues to run indefinitely until explicitly stopped by the modeller or an external process. This attribute provides flexibility for implementing time-bound strategies—for example, controlling ventilation for a fixed post-occupancy period—and persistent control behaviours intended to operate continuously throughout the model's lifecycle.

Abstract Methods.

- `acquire_process_value_data(self, *args, **kwargs) -> Any`: This abstract method, which accepts positional and keyword arguments (Peter D. Kazarinoff, 2025), requires users to implement the specific logic that retrieves real-time values from the primary process value sensor and any associated conditional sensors. The method actively collects all relevant sensor data needed to monitor the system's current state for the control strategy.

By defining this method, users specify how their control logic obtains up-to-date process and conditional values, enabling timely decision-making within the control algorithm.

- `execute_control(self, *args, **kwargs)`: This abstract method requires users to implement the specific logic that translates control decisions into actionable commands through BMS or IoT APIs for building systems. Using the acquired process value data and the predefined control thresholds, the method determines the appropriate control actions, such as adjusting thermostat settings, opening or closing dampers, or switching equipment on or off. By defining this method, users actively control the behaviour of actuators and other building components to maintain desired environmental conditions based on real-time sensor inputs and control strategy rules.

5.2.2.2 AbstractBinaryControl

The `AbstractBinaryControl` interface extends the `AbstractControl` base class for implementing binary (on/off) control strategies. This abstract subclass does not introduce new attributes; instead, it inherits all properties and constructor parameters from `AbstractControl`, ensuring consistency while specialising the control logic for discrete, two-state actuation scenarios.

Abstract Methods.

- `acquire_process_value_data(self) -> Dict`: In the `AbstractBinaryControl` class, this method executes at a frequency determined by the associated process value sensor. The `data_frequency` attribute of the primary process value sensor determines how frequently this method executes. It retrieves the current readings from the primary process value sensor, along with any conditional sensors specified in the `AbstractControl` constructor. The method returns a Python `Dict` mapping sensor identifiers to their measured values, thereby providing the real-time inputs required for subsequent binary (on/off) control decisions. Encapsulating data acquisition logic within this method ensures a clear separation of concerns between sensing and actuation, consistent with object-oriented design principles.

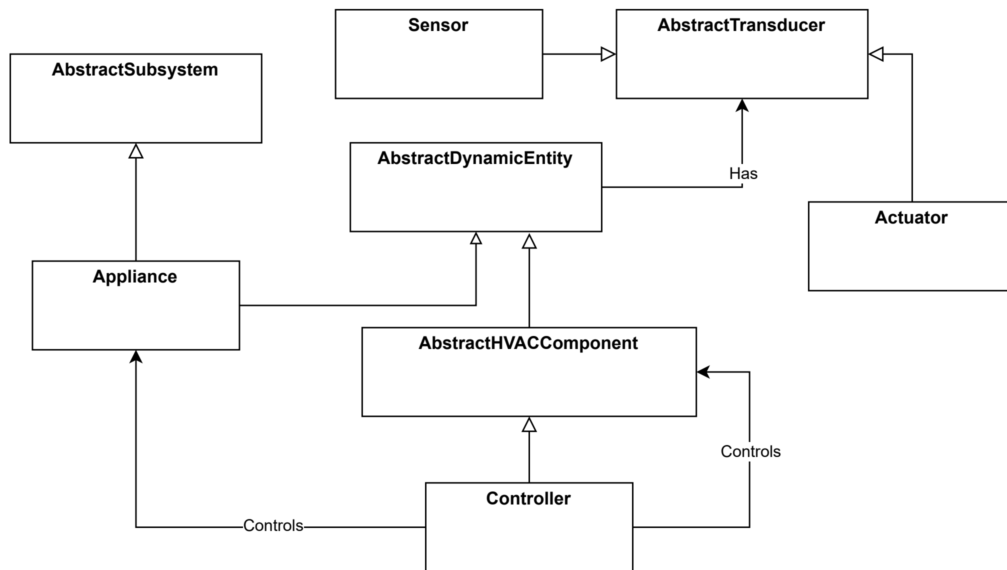


Figure 5.1: **MetamEnTh** classes that enable the integration of user-implemented control logic

- `execute_control(self, process_value: Dict)`: In the `AbstractBinaryControl` class, this method implements the binary control decision-making process. It accepts as input a Python `Dict` of sensor readings produced by the `acquire_process_value_data` method, where the primary process value is compared against the minimum and maximum thresholds defined in the `control_thresholds` attribute inherited from `AbstractControl`. Based on this comparison, the method determines whether to activate or deactivate the associated actuator, thereby altering the system's operational state. This design enforces the separation of data acquisition from decision logic, while supporting extensibility for diverse on/off control scenarios through polymorphic implementations.

5.2.3 MetamEnTh Classes and Attributes Enabling Control Logic Integration

Various **MetamEnTh** classes, as illustrated in Figure 5.1, participate in the integration and execution of control logic within instantiated building models. This section introduces these classes and examines, in detail, how their design and interactions–grounded in object-oriented principles enable seamless incorporation of user-defined control strategies. By exploring their responsibilities and collaborations, we illustrate how **MetamEnTh** provides the structural foundation for bridging high-level control logic with real-time actuation of the building subsystem.

Controller. The `Controller` class inherits from `AbstractHVACComponent` and models a physical controller in an HVAC system. In addition to attributes inherited from its superclass, it defines two key attributes:

- `set_points`: A `Dict` with a `str` key and an `AbstractMeasure` value that stores setpoints associated with specific sensor–actuator pairs. The key is a concatenation of the unique sensor and actuator names (e.g., `"sensor_name:actuator_name"`). Each value is a `ContinuousMeasure` instance (a subclass of `AbstractMeasure`) that specifies a minimum value, a maximum value, and a `MeasurementUnit`. This design ensures that the setpoint is always validated against the measurement unit of the associated sensor.
- `controller_entities`: A list of all `Appliance` objects (e.g., smart plug) and other `AbstractHVACComponent` instances (e.g., boilers, dampers, or fans) under the direct control of the `Controller`.

The `Controller` class provides methods to add, retrieve, and remove setpoints for a given sensor–actuator pair, as well as to manage the list of controlled entities. All modifications enforce strict validation to ensure the existence and compatibility of sensors, actuators, and measurement units before association.

The `control()` method in this class acts as the execution bridge between the physical HVAC components represented by the `Controller` and the abstract control strategies implemented by the user. This method accepts an `AbstractControl` instance—such as an on/off control or PID control strategy—that encapsulates the logic for acquiring process values and executing system adjustments. Before execution, `control()` performs a series of validation checks:

- Ensuring that process value sensors and actuators are configured in the `Controller`.
- Validates sensor-to-control threshold count.
- Verifying that the primary process value sensor specifies a data acquisition frequency.

Once validated, `control()` enters an execution loop that runs for the specified `run_duration` (in hours) or indefinitely if `run_duration` is `None`. Within this loop, the method:

(1) Calls `acquire_process_value_data()` on the supplied control object to retrieve real-time process and conditional values from the configured sensors.

(2) Passes the acquired data to `execute_control()` to enact changes via the configured actuator(s) based on the implemented control logic and threshold conditions.

(3) Waits for the next acquisition cycle based on the data frequency of the process value sensor.

Through this design, the `Controller` class serves as the orchestrator, binding real-world HVAC components to high-level, user-defined control strategies, ensuring seamless, modular integration between the simulation model and executable control logic.

Actuator. The `Actuator` class inherits from `AbstractTransducer` and models a physical actuator in a building system, such as a damper motor, valve actuator, or relay switch. Its primary role is to receive control signals—typically from a `Controller`—and alter the state of a connected HVAC component or appliance in response to those signals.

In addition to attributes inherited from `AbstractTransducer`, the `Actuator` class defines these attributes:

- `actuated_component`: An instance of `AbstractHVACComponent` or `Appliance` that represents the physical device being actuated.
- `controller`: An optional reference to the `Controller` responsible for issuing commands to this actuator.
- `actuation_interval`: An optional `float` value indicating the time interval (in seconds) between successive actuation events. Users can use `actuation_interval` to prevent overly frequent actuation, thereby reducing wear on mechanical components.

Within the broader control framework, the `Actuator` serves as the execution endpoint of control strategies. When a control strategy—executed by a `Controller`—determines that a system change is required, it issues a signal to the relevant `Actuator`, which in turn adjusts the state of its `actuated_component`. This separation of concerns ensures that the decision-making logic resides in the controller, while the actuator executes those decisions at the physical device level.

Sensor. The `Sensor` class inherits from `AbstractTransducer` and represents a data-acquisition device in a building system. Its primary role is to measure physical phenomena (e.g., temperature, humidity, pressure) and provide these readings at specified intervals to other components—most notably, the `Controller`—for use in control strategies.

In addition to attributes inherited from `AbstractTransducer`, the `Sensor` class defines:

- `measure`: A `SensorMeasure` value specifying the physical phenomenon measured, such as temperature or humidity.
- `unit`: A `MeasurementUnit` instance indicating the unit of measurement (e.g., °C, Pa). The `measure` and `unit` are validated together to ensure compatibility (e.g., a temperature sensor cannot have units of *litres*).
- `measure_type`: A `SensorMeasureType` defining the nature of the measurement data (e.g., continuous or discrete).
- `data_frequency`: A `float` specifying the time interval (in seconds) between data acquisition events by the process value sensor.
- `current_value`: An optional `float` representing the most recent measured value, allowing for quick access without querying historical data.
- `measure_range`: An optional `AbstractRangeMeasure` specifying the permissible range for the measured phenomenon, enabling validation and fault detection.
- `sensor_log_type`: A `SensorLogType` value that determines the logging behaviour of the sensor, such as polling-based logging or event-triggered logging.

In the control framework, the `Sensor` acts as the origin point of the feedback loop. `Controller` instances consume their readings to compare against setpoints and make actuation decisions.

AbstractHVACComponent. The `AbstractHVACComponent` class inherits from `Abstract-DynamicEntity` and serves as the parent class for all HVAC entities within the **MetamEnTh** framework. Examples of subclasses include `Boiler`, `Chiller`, `HeatExchanger`, and `Damper`.

This abstraction defines the fundamental attributes and behaviours common to all HVAC components while allowing subclasses to specialise their operational logic and data management. The class has helper methods for entity management:

- Adding, removing, and searching spaces served by the component.
- Adding, removing, and searching operational status measurements.
- Managing and extending `operating_conditions` in a controlled manner.
- Registering observers and notifying them when users add new `StatusMeasure` objects.

A key feature of the class is its handling of `StatusMeasure` instances, which capture the operational state of an HVAC component at a given time (e.g., when a boiler is switched on or off). To extend this capability, the class implements the *observer pattern*, allowing external functions to register as observers of these state changes. Whenever users add a new `StatusMeasure`, all registered observers are notified, enabling the implementation of user-defined behaviours such as logging, triggering additional control actions, or updating energy consumption models.

In the context of control logic integration, `AbstractHVACComponent` instances are often associated with `Actuator` objects, enabling automated changes to their operational state in response to control decisions. By modelling physical HVAC assets in an abstract form and supporting event-driven responses through observers, **MetamEnTh** enables users to implement control strategies in a hardware-agnostic, extensible, and responsive manner.

Appliance. The `Appliance` class models smart or controllable devices in the built environment, such as thermostats, smart switches, printers, or other network-connected appliances. It inherits from both `AbstractSubsystem` and `AbstractDynamicEntity`, reflecting its dual nature as a functional subsystem within a building and as an entity capable of interacting with associated `Sensor` and `Actuator` instances.

This class encapsulates attributes for appliance management and control:

- `name`: The unique identifier for the appliance within the model.

- appliance_category: A list of ApplianceCategory values that classify the appliance (e.g., home appliance, office appliance).
- appliance_type: An ApplianceType enumeration specifying the appliance's specific kind (e.g., smart plug, photocopier, lighting controller).
- manufacturer: An optional string identifying the device manufacturer.
- consumption_capacity: An AbstractMeasure instance representing the maximum power or resource consumption capacity of the appliance.
- rated_device_measure: A RatedDeviceMeasure specifying nominal performance characteristics of appliances.
- operating_conditions: A list of ContinuousMeasure objects representing measurable operating conditions (e.g., internal temperature, load percentage).

From a control logic perspective, Appliance instances serve as integration points between building automation logic and real-world device behaviour. Because they inherit from AbstractDynamicEntity, they can be directly associated with Sensor objects (to monitor process variables) and Actuator objects (to execute control commands).

In **MetamEnTh**, the integration of control logic is enabled through a set of abstract interfaces and concrete classes that model the behaviour of building entities, their sensors, and actuators. The acquire_process_value_data and execute_control interfaces define standardised mechanisms for retrieving process values and applying decision-making logic, ensuring that control strategies remain consistent and reusable across different entity types. These are supported by key classes such as Actuator, which links control logic to physical outputs, and AbstractHVACComponent and Appliance, which represent controllable systems ranging from HVAC equipment to smart appliances. The control framework also accommodates different control paradigms through interfaces such as AbstractBinaryControl for on/off actuation and AbstractPIDControl for continuous proportional–integral–derivative regulation. While their structural design is the same, the AbstractPIDControl implementation computes the proportional, integral, and derivative terms, enabling finer-grained control over system behaviour.

This object-oriented approach directly addresses **RQ4** by encapsulating both binary and continuous control capabilities within modular, interoperable components, enabling the same structural framework to be applied to different control strategies without redesign. It also models system behaviours—such as operating conditions, schedules, and actuation triggers—in ways that facilitate the integration and deployment of energy-efficient control strategies. This design bridges the gap between high-level building models and runtime control execution, reducing implementation complexity and supporting interoperability across HVAC, appliances, and IoT-based systems.

5.3 Evaluation

5.3.1 Use Case: Webster Library

The subsequent section details the implementation of **MetamEnTh** interfaces for a binary control strategy deployed for the LB Building.

5.3.1.1 AbstractBinaryControl Implementation

In the LB Building use case, we implemented an on/off control strategy for Room 345 (Living Lab) on the third floor using the `AbstractBinaryControl` interface. The control strategy retrieves temperature and CO_2 concentration values from sensors (Systems Hardware layer in Chapter 4 on Page 81) in the room via Desigo's BMS API (Data Exchange layer) and determines whether to activate or deactivate a modelled boiler based on predefined conditions. Because we lack write access to the deployed BMS system, we modelled the boiler as an HVAC component in **MetamEnTh** and simulated actuation decisions by printing the intended action to the console.

The boiler is turned *off* when the room temperature exceeds 24°C or the CO_2 concentration drops to 400 ppm or below—conditions that suggest the space is sufficiently warm or unoccupied. Conversely, the boiler is turned *on* when the temperature falls to 16°C or below and CO_2 levels reach 420 ppm or above, suggesting cooler conditions and potential occupancy. This experiment demonstrates how **MetamEnTh's** abstractions support the integration of control logic in a hardware-agnostic way. The `AbstractBinaryControl` interface encapsulates the decision rules, while the boiler (`AbstractHVACComponent`) provides a target for simulated actuation.

Listings 5.2 illustrates the implementation of the `execute_control` interface defined in `AbstractBinaryControl`. The `acquire_process_value_data` method builds on the API access patterns introduced in Chapter 4, Section 4.3.2, but encapsulates them in a dedicated `BmsAPI` class. This class exposes a `get_valid_token` method for authentication and a `make_authenticated_request` method for data retrieval (illustrated in Listings 5.1). Integrating these into `acquire_process_value_data` provides a reusable, modular interface for connecting external sensor streams to internal control entities.

```
1 def make_authenticated_request(self, api_url, method=None, data=None):
2     """Makes an authenticated API GET request, with retry if token has expired.
      """
3     token = self.get_valid_token()
4     headers = {"Authorization": f"Bearer {token}"}
5     response = requests.get(f'{self._api_url}{api_url}', headers=headers,
6                  verify='HVAC.CONCORDIA.CA.pem')
7     # If unauthorised, re-authenticate and retry once
8     if response.status_code == 401:
9         self.get_token()
10        token = self._access_token
11        headers["Authorization"] = f"Bearer {token}"
12        response = requests.get(f'{self._api_url}{api_url}', headers=headers,
13                     verify='HVAC.CONCORDIA.CA.pem')
14    if response.status_code == 200:
15        return response.json()
16    else:
17        print("API request failed:", response.status_code, response.text)
18        return None
```

Listing 5.1: Snippets of Python code show how we authenticated with Desigo API

```
1  def execute_control(self, process_value: Dict):
2       SETPOINT_INDEX=0
3       CONDITION_INDEX=1
4       if process_value['temperature']
5          > self.control_thresholds[SET_POINT_INDEX].maximum
6          or process_value['carbon_dioxide']
```

```
7             < self.control_thresholds[CONDITION_INDEX].minimum:
8              print(f'process value of {process_value['temperature']} and '
9                    f'{process_value['carbon_dioxide']} are greater or less '
10                   f'than desirable thresholds of '
11                   f'{self.control_thresholds[SET_POINT_INDEX].minimum} and '
12                   f'{self.control_thresholds[CONDITION_INDEX].minimum} '
13                   f'respectively')
14             print(f'Triggering process actuator to turn off boiler.')
15             self.actuator.actuated_component
16                 .add_status_measure(StatusMeasure(PowerState.OFF.value))
17         elif process_value['temperature']
18              < self.control_thresholds[SET_POINT_INDEX]
19              .minimum and process_value['carbon_dioxide']
20              >= self.control_thresholds[CONDITION_INDEX].maximum:
21             # Temperature is lower than the acceptable threshold,
22             # and there is someone in the room. Turn on the boiler
23             self.actuator.actuated_component
24                 .add_status_measure(StatusMeasure(PowerState.ON.value))
25             print(
26                 f'process value of {process_value['temperature']} and '
27                 f' {process_value['carbon_dioxide']} is lesser and '
28                 f'greater than desirable thresholds of '
29                 f'{self.control_thresholds[SET_POINT_INDEX].minimum} and '
30                 f'{self.control_thresholds[CONDITION_INDEX].minimum} '
31                 f'respectively')
32             print(f'Triggering process actuator to turn on boiler.')
33             self.actuator.actuated_component
34                 .add_status_measure(StatusMeasure(PowerState.ON.value))
```

Listing 5.2: Snippets of Python code for our execute_control implementation

5.3.1.2 Control Execution Workflow

Listing 5.3 illustrates the application of the implemented `AbstractBinaryControl` interface in the LB Building use case. The example demonstrates the end-to-end workflow of integrating building data, modelling both physical and virtual devices, and linking them to control logic within **MetamEnTh**. We instantiated a building model of the Webster Library using the `BuildingCreator` helper class (explained in detail in Chapter 3, Section 3.3.2.3), which constructs the hierarchical structure of floors, rooms, and transducers from the `webster.json` configuration file. We then define control targets and conditions using `MeasureFactory`, which encapsulates setpoints for both temperature (16–24°C) and a conditional variable for CO_2 concentration (400–420 ppm). The illustrated code shows a modelled boiler as the target HVAC component that the control algorithm will actuate. The `Controller` instance manages the association between sensors, actuators, and setpoints, thereby coordinating the execution of the binary control strategy. Finally, we created an instance of `LBBinaryControl`—the case-specific implementation of `AbstractBinaryControl`—that binds the process value sensors, actuator, and setpoints. When `controller.control` is executed, the binary logic continually evaluates the incoming sensor readings against the setpoints and determines whether to switch the boiler on or off. This workflow exemplifies how **MetamEnTh's** abstractions support the practical implementation of control logic in a modular and reusable manner.

```
1  # Create a model for the Webster library
2  LIBRARY_FLOOR = 3
3  MECHANICAL_ROOM = 11
4  webster_library = BuildingCreator('webster_library.json',
5                                    LBDynamicDataReader(LIBRARY_FLOOR))
6  webster_library.create_building()
7  # add sensor and sensor data
8  webster_library.add_sensor_data()
9
10 # Retrieve the process value sensor for the binary control strategy
11 temp_sensor = (webster_library.building.get_floor_by_number(LIBRARY_FLOOR)
12                .get_room_by_name('0345-00')
13                .get_transducer_by_name('TEC1.345-00.LB:ROOM TEMP'))
```

```
14
15 # Create and add co2 sensor to model
16 co2_conditional_sensor = Sensor('TEC1.345-00.LB:ROOM C02', SensorMeasure
17                          .CARBON_DIOXIDE, MeasurementUnit.PARTS_PER_MILLION,
18                          SensorMeasureType.THERMO_COUPLE_TYPE_A, 900)
19
20 webster_library.building.get_floor_by_number(LIBRARY_FLOOR).get_room_by_name(
       '0345-00').add_transducer(co2_sensor_conditional_sensor)
21
22 # Temperature setpoint for the process value. Ideally, this should be
23 # retrieved with an API for the process value. However, there is currently
24 # no endpoint to retrieve the setpoints for the living lab project
25 temperature_set_point    = MeasureFactory.create_measure(
26                            RecordingType.CONTINUOUS.value,
27                            Measure(MeasurementUnit.DEGREE_CELSIUS, 16, 24))
28 co2_conditional_variable = MeasureFactory.create_measure(
29                            RecordingType.CONTINUOUS.value,
30                            Measure(MeasurementUnit.PARTS_PER_MILLION,
31                            400, 420))
32
33 # Simulate HVAC component to actuate
34 boiler = Boiler('CTRL.BL', BoilerCategory.NATURAL_GAS, PowerState.ON)
35
36 # Create controller
37 controller = Controller('CTR')
38
39 # Actuator to actuate the boiler
40 actuator = Actuator("Boiler.ACT", boiler)
41
42 # Indicate the actuator and sensor (process value source) for the
43 controller.add_transducer(temp_sensor)
44 controller.add_transducer(co2_sensor_conditional_sensor)
45 controller.add_transducer(actuator)
46
47 # Boiler, controller to the mechanical room on the 11th floor
```

```
48 mechanical_room = webster_library.building.get_floor_by_number(
49                     MECHANICAL_ROOM).get_room_by_name('1120-00')
50 mechanical_room.add_hvac_component(controller)
51 mechanical_room.add_hvac_component(boiler)
52
53 # Add set point for this controller
54 controller.add_set_point(temperature_set_point, (temp_sensor.name, actuator.
       name))
55 controller.add_set_point(co2_conditional_variable, (co2_conditional_sensor.
       name, actuator.name))
56
57 # Instantiate control class
58 binary_control = LBBinaryControl([temp_sensor, co2_conditional_sensor],
59                    actuator, [temperature_set_point, carbon_dioxide_set_point])
60
61 if __name__ == '__main__':
62     controller.control(binary_control)
```

Listing 5.3: Snippets of Python code for our execute_control implementation

5.3.2 Use Case: Ludwigsburg Residential and Commercial Building

The Ludwigsburg residential and commercial building use case features an IoT setup that monitors and controls devices in an office labelled Unit 3. The setup uses a server running Node-RED (OpenJS Foundation, 2025) (BMS Software layer in Figure 4.1 on Page 81) to program two IoT entities: a power meter and a thermostat (Systems Hardware layer). The thermostat embeds a temperature sensor, maintains a setpoint, and actuates a valve, while both devices communicate over the LoRaWAN (The Things Network, 2025) protocol. We model these IoT entities using the FIWARE Data Model (FIWARE Foundation, 2025), which specifies their attributes and relationships. A FIWARE REST API (FIWARE Foundation, 2019) (Data Exchanger layer) exposes endpoints to retrieve device data—including real-time temperature, setpoint, and valve position values—via the Hypertext Transfer Protocol (HTTP). Figure 5.2 illustrates this setup, highlighting the entities and their communication flow.

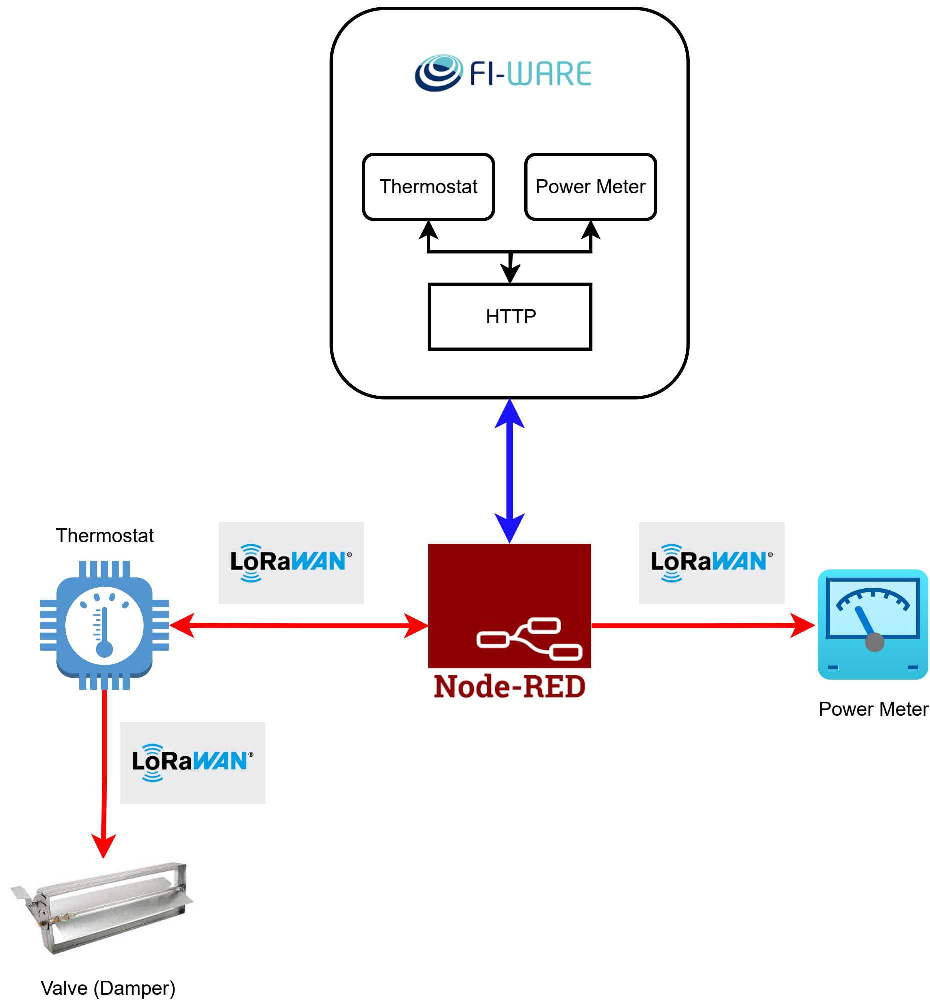


Figure 5.2: The IoT setup in office Unit 3 of the residential and commercial building in Ludwigsburg shows how various devices use LoRaWAN for communication and expose their modelled entities through FIWARE REST APIs

5.3.2.1 AbstractBinaryControl Implementation

Like in the LB Building use case, we implemented an `IotAPI` class with helper methods for sending `GET` and `PATCH` requests. The `AbstractBinaryControl` implementation for this use case relies on this class to communicate with the IoT system.

In contrast to the LB Building implementation of `acquire_process_value_data`, which makes multiple API requests to retrieve both temperature and carbon dioxide values, we issue a

single request to obtain the thermostat metadata modelled with the FIWARE Data Model. This response includes the current temperature, setpoint, and valve position, since the thermostat integrates a temperature sensor and directly actuates the valve.

The current configuration links setpoint updates directly to the valve position. As a result, the reported setpoint may not always reflect the optimum indoor temperature. We correct the setpoint using conditional logic, ensuring that the value used in the control loop always lies between 20°C and 26°C. This adjustment prevents unintended overheating or overcooling when the actual setpoint drives valve actuation.

Listing 5.4 shows the implementation of `acquire_process_value_data`.

```
1 def acquire_process_value_data(self) -> Dict:
2     """
3     Acquire the present value and setpoint of the temperature sensor in
4     Unit 3
5     """
6     SETPOINT_INDEX = 0
7     uri = f'entities/{self.actuator.actuated_component.name}'
8     response = self._iot_api.make_authenticated_request(uri)
9 
10    process_values = {
11        "temperature": response.get(f'{self.TEMPERATURE_KEY}').get("value")
12    }
13    valve_position = response.get('valveOpenness').get("value")
14 
15    # Ensure user-provided set point is in sync with thermostat values
16    setpoint = self.control_thresholds[SETPOINT_INDEX]
17    self._original_set_point = response.get('setPointTemperature').get('value'
      )
18 
19    # For demo purposes, we set the minimum temperature to the actual set
      point value minus 10
20    # and the maximum temperature value to the actual set point minus 6
21    setpoint.minimum =  self._original_set_point - 10
22    setpoint.maximum =  self._original_set_point - 6
```

```
23
24    # ensure minimum temperature is not below 20 degrees Celsius and not more
       than 22 degrees Celsius
25    MIN_THRESHOLD = 22
26    MAX_THRESHOLD = 26
27    setpoint.minimum = max(20, min(setpoint.minimum, MIN_THRESHOLD))
28
29    # ensure maximum temperature is not less than 22 and not more than 26
30   setpoint.maximum = max(22, min(setpoint.maximum, MAX_THRESHOLD))
31
32    self.control_thresholds[SETPOINT_INDEX] = setpoint
33    print(f'Retrieve temperature value of {process_values["temperature"]} '
34          f'degree Celsius, set point value of {self._original_set_point} '
35          f'degree Celsius, and valve position of {valve_position}')
36    return process_values
```

Listing 5.4: Snippets of Python code for our acquire_process_value_data implementation

Unlike in the LB Building use case, the implementation of `execute_control` issues a *PATCH* request to adjust the valve position by modifying the temperature setpoint. We employ the observer pattern to monitor updates to the damper position.

Listing 5.5 illustrates portions of the `AbstractBinaryControl` implementation. On *line 8*, we attach an observer to the damper (`actuated_component`) to track changes in its position. The observer implementation, `valve_update_observer`, begins on *line 25* and initiates the *PATCH* request (based on *data_frequency* of the *process_value_sensor*) whenever the `execute_control` method updates the damper to satisfy predefined conditions.

Currently, updates to the damper or valve position do not affect Unit 3's indoor temperature, as the damper is not physically connected to the building's HVAC system. Nevertheless, this setup demonstrates how **MetamEnTh** integrates control logic with a semantic representation of the built environment in a consistent and extensible way.

This implementation addresses **RQ4:** How does an object-oriented modelling approach support the integration of control? **MetamEnTh** applies object-oriented programming principles by defining interfaces as contracts that users must implement when integrating their control logic. Through

the observer pattern, **MetamEnTh** ensures that changes in building state—whether triggered by environmental sensors or control decisions—cascade into subsequent actions in monitored environments. Achieving this form of integration is not feasible with ontologies alone, since they rely on static semantic files and require additional integration layers to interact with live system behaviour.

```
1  class LudwigsburgBinaryControl(AbstractBinaryControl):
2   TEMPERATURE_KEY = 'https://smartdatamodels.org/dataModel.Device/
    temperature'
3   SETPOINT_INDEX=0
4
5   def __init__(self, process_value_sensors: [Sensor], actuator: Actuator,
    control_thresholds: [ContinuousMeasure]):
6       super().__init__(process_value_sensors, actuator, control_thresholds)
7       self._iot_api = IotAPI()
8       # register observer for damper/valve here
9       self.actuator.actuated_component
10           .add_update_observer(self.valve_update_observer)
11      self._original_set_point = None
12
13  def execute_control(self, process_value: Dict):
14      if process_value['temperature'] > self.control_thresholds[self.
    SETPOINT_INDEX].maximum:
15          (self.actuator.actuated_component
16           .add_damper_position(
17               DamperPosition( self._original_set_point - self.
    VALVE_UPDATE_VAL)))
18          print(f'Temperature is above maximum value of'
19                f' {self.control_thresholds[self.SETPOINT_INDEX].maximum}.
    Adjusting '
20                f'set point from {self._original_set_point} to '
21                f'{self._original_set_point - self.VALVE_UPDATE_VAL}')
22      elif process_value['temperature']
23      < self.control_thresholds[self.SETPOINT_INDEX].minimum:
24          # Increase the temperature by closing the valve.
25          (self.actuator.actuated_component
```

```
26            .add_damper_position(
27            DamperPosition( self._original_set_point + self.VALVE_UPDATE_VAL)
    ))
28           print(f'Temperature is below minimum value of'
29                 f' {self.control_thresholds[self.SETPOINT_INDEX].minimum}.
    Adjusting '
30                 f'set point from { self._original_set_point} to '
31                 f'{self._original_set_point + self.VALVE_UPDATE_VAL}')
32
33   def valve_update_observer(self, action, valve_position):
34       """
35       Observer for updating damper position in Unit 3
36       :param action: the action (mostly adding new damper/valve position)
37       :param valve_position: the damper position being added
38       :return:
39       """
40       uri = f'entities/{self.actuator.actuated_component.name}/attrs'
41       json_data = {
42           "setPointTemperature": {
43               "type": "Property",
44               "value": valve_position.value
45           },
46           "@context": [
47               "https://smartdatamodels.org/context.jsonld"
48           ]
49       }
50       self._iot_api.make_authenticated_request(uri, 'PATCH', json_data)
```

Listing 5.5: Snippets of Python code for our acquire_process_value_data implementation

5.4 Discussions

In this section, we discuss the approach and its evaluation, focusing on how it addresses **RQ4**.

5.4.1 Approach

Semantic modelling of built environments using standardised models, rather than vendor-specific representations of building systems, abstracts away the heterogeneity inherent in how different BMS vendors structure their data. Ontologies and metamodels provide a common vocabulary for describing entities and their relationships, thereby enabling interoperability and facilitating the development of portable applications across multiple buildings. However, semantic models are not an end in themselves. Practitioners and researchers utilise them to represent heterogeneous building data in a homogeneous manner, thereby achieving specific objectives such as energy efficiency, advanced monitoring, or predictive maintenance.

In **MetamEnTh**, we adopted an object-oriented approach to modelling building systems and their complex interactions, given the advantages this paradigm offers over traditional ontology-based representations. Beyond ensuring accurate modelling of real-world entities through constraint enforcement, object-oriented programming introduces dynamic characteristics that better support integration and system behaviour. Interfaces allow us to define contracts (e.g., for control strategies) that different implementations can fulfil, ensuring consistency while supporting extensibility. Encapsulation enables the separation of internal system states from external interactions, reducing complexity and avoiding unintended side effects. Design patterns, such as the observer pattern used in our control logic integration, ensure a systematic approach to handling dynamic interactions among sensors, actuators, and control entities.

Another advantage of this approach is the use of **dynamic, memory-resident objects**, which allow semantic models to function as active, computational artefacts rather than static knowledge graphs. This dynamism facilitates seamless integration with external data sources (e.g., BMS APIs, IoT platforms) and enables real-time decision-making without the need for additional integration layers. As a result, multiple applications—ranging from energy-efficient control strategies to digital twin implementations and real-time monitoring dashboards—can operate directly on the same object-oriented model. This approach reduces system complexity, increases portability across different building contexts, and lowers the barrier for adoption by practitioners who are already familiar with object-oriented programming concepts.

By embedding object-oriented programming principles within the semantic representation of built environments, **MetamEnTh** bridges the gap between static domain models and dynamic control execution. This design choice supports **RQ4** by showing that object-oriented modelling not only enhances the integration of control logic but also promotes broader adoption in practical applications aimed at improving building energy efficiency.

5.4.2 Evaluation

To evaluate our approach and demonstrate its generic applicability in deploying control logic within built environments, we implemented two use cases in two different buildings with distinct technical setups. In the LB Building, we utilised Siemens' Desigo API, which provides endpoints for retrieving sensors. In contrast, the mixed-use building in Ludwigsburg, Germany, employed an IoT-based architecture. Devices in this setup were modelled using the Fiware Data Model and exposed through the Fiware REST API, with communication facilitated via LoRaWAN.

These two cases demonstrate how practitioners and researchers can incorporate their own control logic and integrate standardised models with heterogeneous building systems by utilising the interfaces provided by **MetamEnTh**. Importantly, these integrations do not require modifications to the core model or the introduction of additional middleware layers. Instead, the defined interfaces and patterns (e.g., the Observer pattern) provide a mechanism for seamlessly coupling semantic models of built environments with actionable control strategies. Through this approach, a change in the state of model attributes—such as a valve position or setpoint—automatically propagates into corresponding control actions, demonstrating a direct mapping between the semantic representation and physical behaviour.

The use cases show that the object-oriented interfaces defined in **MetamEnTh** generalise well across different system architectures—ranging from traditional BMS to IoT-based setups—while avoiding the complexity of additional integration layers. The use cases also expose socio-technical barriers, such as licensing and risk perception, that limit researchers' and practitioners' ability to experiment with live systems.

5.5 Threats to Validity

5.5.1 Construct validity

We do not claim expertise in designing or implementing control strategies, especially regarding the underlying logic of `execute_control`. Our focus is on demonstrating how **MetamEnTh** enables their integration, rather than proposing new or optimised strategies.

5.5.2 Internal validity

The current implementation does not yet include interfaces for more advanced control strategies. Extending the system in this way may require modifications to the classes in **MetamEnTh** that handle control-strategy integration. This limitation introduces uncertainty regarding whether our current design can fully support all advanced use cases without structural changes.

5.5.3 External validity

Our evaluation centres on how **MetamEnTh** addresses integration challenges that traditional ontologies cannot solve. Because we did not implement or test a wide variety of advanced control strategies, the generalizability of our results to all control scenarios may be limited.

5.5.4 Conclusion validity

Because our work primarily focuses on integration mechanisms rather than the performance or correctness of the control logic, readers should interpret the conclusions accordingly. Any inference beyond the integration capabilities of **MetamEnTh** may not hold.

5.6 Conclusion

This chapter investigated how an object-oriented modelling approach can support the integration of control logic and facilitate broader adoption in practical energy-efficient building applications, directly addressing **RQ4**: How does an object-oriented modelling approach support the integration of control logic and promote broader adoption in practical energy-efficient applications?

Through the design, implementation, and evaluation of **MetamEnTh**, we demonstrated that the object-oriented approach enables the seamless integration of control strategies with semantic representations of building systems. Defined interfaces, such as `AbstractBinaryControl` and `AbstractPIDControl`, combined with the Observer pattern, enable control actions to be triggered dynamically in response to changes in sensor readings or equipment states, without modifying the underlying models. This approach overcomes limitations inherent in static ontologies, which cannot natively execute control logic or propagate changes through system models.

The chapter further highlights the versatility of **MetamEnTh** through two distinct use cases: the LB Building with Siemens Desigo BMS and the Ludwigsburg residential and commercial building, featuring IoT devices modelled via the Fiware Data Model. These examples demonstrate that researchers and practitioners can effectively integrate control logic, link sensors and actuators, and deploy energy-efficiency strategies in heterogeneous building environments without requiring additional integration layers. While the current implementations primarily focus on binary control strategies, the adopted object-oriented principles enable future extension to more advanced control schemes, such as multi-variable control, with minimal impact on existing components.

In conclusion, **MetamEnTh** demonstrates that an object-oriented modelling approach actively supports the integration and execution of control logic. **MetamEnTh** provides a reusable, modular, and dynamic foundation for developing energy-efficient applications, bridging the gap between semantic models and actionable control strategies, and thereby enabling broader practical adoption in diverse built environments.

Chapter 6

Conclusion

Buildings comprise multiple complex systems that work together to maintain comfortable indoor conditions for occupants while ensuring efficient resource use. Building Management Systems (BMS) coordinate and control these systems to function as a coherent unit, supporting both occupant safety and energy efficiency. Despite these efforts, buildings still consume roughly 40% more energy than necessary, highlighting significant inefficiencies in building management systems.

To deploy energy-efficient solutions, researchers and practitioners need access to building systems and their data in a standardised and interoperable form. However, BMS vendors rely on proprietary representations of both systems and data, which creates interoperability challenges and prevents the deployment of unified solutions across multiple buildings. To address this challenge, researchers have explored ontology-based approaches that abstract vendor-specific heterogeneity into standardised representations. Although ontologies offer flexibility and can effectively capture the complexity of mechanical, electrical, and plumbing (MEP) subsystems, we identified eight key limitations: L_1 ambiguity in defining entities; L_2 potential for composing flawed entity relationships; L_3 absence of entities for ducts, pipes, and detailed spatial contexts; L_4 difficulty in validating entities and relationships; L_5 inconsistent data and structural semantics; L_6 complex and deeply nested inheritance hierarchies; L_7 inability to represent system behaviours; and L_8 static artefacts with limited interactivity. These limitations restrict their ability to support diverse energy-efficiency applications, particularly those requiring dynamic interaction, real-time monitoring, and portable control strategies.

In this thesis, we go beyond merely identifying the limitations of ontology-based approaches and implement an alternative metamodel. We evaluate BMS APIs to assess their interoperability and investigate how object-oriented metamodels overcome the limitations of ontologies. We further demonstrate how features such as predefined classes, validation mechanisms, and the integration of control logic make metamodels a more practical alternative for modelling built environments and deploying energy-efficient solutions.

The thesis statement below guided this research:

> **Thesis Statement**
>
> The current complexity of representing mechanical, electrical, and plumbing (MEP) systems hinders data interoperability and dynamic control deployment. Our thesis posits that an object-oriented modelling approach resolves these issues by providing consistent, standardised access to MEP data and enabling the integration of control strategies via structured class definitions and interfaces and validation mechanisms compatible with Building Management Systems (BMS).

This thesis demonstrates that modelling the operational aspects of buildings—particularly mechanical, electrical, and plumbing (MEP) systems—requires representations that extend beyond static semantic abstractions. By introducing an object-oriented metamodel grounded in practitioner and researcher requirements, evaluating real-world BMS APIs, and validating the approach through empirical use cases, this work shows that executable, validated, and interoperable models can effectively bridge the gap between building system representation and operational control. The results indicate that **MetamEnTh** provides a practical foundation for deploying portable, energy-efficient control strategies across heterogeneous BMS platforms, thereby supporting both research innovation and real-world building operations.

While this thesis provides an empirical comparison between Brick and **MetamEnTh** for modelling the operational aspects of buildings, it does not attempt a comprehensive qualitative and quantitative benchmarking of ontology-based and metamodel-based approaches. A systematic evaluation—covering dimensions such as modelling effort, expressiveness, validation capability, scalability, fault

tolerance, level of modelling detail, ease of use, and integration with BMS—remains an open research challenge. Future work should pursue controlled qualitative studies with domain experts alongside quantitative experiments to rigorously assess the trade-offs between ontology-based and executable metamodel-based approaches. Such evaluations would strengthen the evidence base for selecting appropriate modelling paradigms and clarify the contexts in which ontologies or object-oriented metamodels are most effective for supporting energy-efficient building operations.

This work assumes that BMS and IoT platforms expose data that is functionally correct and representative of the true state of building subsystems. This assumption reflects current practice in both research and industry, where control and optimisation strategies typically rely on the accuracy of sensor measurements and system feedback from BMS platforms. However, in real-world deployments, data quality issues such as sensor faults, calibration drift, communication delays, or misconfigured points may violate this assumption and affect the reliability of control execution.

Addressing such data reliability concerns lies outside the scope of this thesis, which focuses on the modelling, validation, and integration of operational building systems rather than on fault detection or data assurance. Nevertheless, future research should investigate mechanisms for validating, filtering, and reasoning over uncertain or inconsistent BMS data within the proposed metamodel. This may include integrating data quality indicators, confidence measures, anomaly-detection techniques, or fault-tolerant control strategies to ensure safe operation when underlying data sources cannot be assumed fully reliable.

6.1 Summary of Contributions

We summarise the contributions of this thesis and organise them around four research questions.

6.1.1 RQ1 & RQ2

After identifying the research gap through an extensive literature review in Chapter 2, we identified **eight** limitations of existing ontology-based representation systems and grounded our approach in expert input and detailed analysis of HVAC systems in Chapter 3.

To capture the practical needs of researchers and practitioners, we conducted a needs assessment

involving surveys and interviews with facility managers. These engagements revealed recurring challenges with interoperability, a lack of validation, and difficulties extending existing ontologies for advanced applications, aligning with the limitations identified in our literature review.

Guided by these findings and the central research questions—**RQ1**: In what ways does an object-oriented metamodel enable the creation of accurate, inherently validated models compared to existing ontology-based approaches? and **RQ2**: How can an object-oriented metamodel represent mechanical, electrical, and plumbing (MEP) systems and their complex interactions?—We designed and implemented an object-oriented metamodel, **MetamEnTh**, that enforces constraints, validates model consistency, and captures the complexity of MEP subsystem interactions.

Our empirical evaluation revealed that **MetamEnTh** reduces semantic modelling errors and enables more detailed representations of HVAC components and their relationships compared to Brick. In a structured experimental setup, developers implemented typical modelling scenarios in both **MetamEnTh** and Brick, and their feedback confirmed **MetamEnTh's** fault tolerance and model accuracy despite a steeper learning curve. Overall, **MetamEnTh** provides reusable abstractions, encapsulated behaviours, and attributes that deliver operational accuracy and flexibility that static ontologies cannot.

6.1.2 RQ3

In Chapter 4, we addressed **RQ3**, which investigated how practitioners and researchers can instantiate models of MEP subsystems using heterogeneous APIs while maintaining semantic consistency. We proposed a five-layered architecture that connects clients to building systems through vendor APIs, ensuring interoperability.

Through extensive analysis of BMS APIs, we demonstrated both the potential and the limitations of current vendor offerings for supporting model instantiation and energy-efficient applications. We evaluated the capability of BMS APIs by mapping 12 feature groups against 85 building applications. Our findings show that while all BMS vendors support basic applications through access to objects and time-series data, more advanced applications—such as demand response and predictive control—require combinations of features that no single vendor fully supports.

We further examined whether semantic ontologies, specifically Brick and Project Haystack, can

bridge interoperability gaps by representing BMS API data in standardised ways. We concluded that although both ontologies offer rich semantic models and are widely adopted, they lack constructs for domains such as audits, LEED scoring, and energy modelling. Moreover, they cannot directly enact control logic without additional integration layers.

6.1.3 RQ4

In Chapter 5, we addressed **RQ4**, which asked how object-oriented modelling supports the integration of control logic and encourages broader adoption in energy-efficiency applications.

We demonstrated, through multiple use cases, how **MetamEnTh** defines interfaces such as `AbstractBinaryControl` and `AbstractPIDControl`, enabling researchers and practitioners to embed control logic directly into object-oriented models.

We demonstrated how the Observer pattern and structured interfaces enable dynamic control actions to be triggered by changes in sensor readings or equipment states, eliminating the need for additional integration layers. We demonstrated the practicality of our approach by applying it to case studies and by developing interfaces and behaviours that facilitate the real-world deployment of control strategies to improve energy efficiency.

6.2 Answer

Together, the results of Chapters 3, 4, and 5 addresses the thesis statement. An object-oriented modelling approach not only effectively represents MEP systems but also supports the integration of heterogeneous BMS data, enabling seamless deployment of control logic. By leveraging encapsulation, interfaces, and design patterns, we created a metamodel that is both flexible and extensible and directly usable by practitioners and researchers alike.

While challenges remain—particularly around API access, the implementation of advanced control strategies, and bridging the gap between simulated and deployed systems—this work shows that object-oriented modelling provides a practical, interoperable, and scalable path toward energy-efficient building management.

MetamEnTh thus provides a novel foundation for advancing digital twins, simulation platforms, and real-time control systems in the built environment, enabling the next generation of interoperable, actionable building models.

6.3 Future Work

This thesis advances object-oriented modelling, interoperable data access, and control integration for MEP subsystems. We identify several opportunities for future research, grouped into short-, mid, and long-term directions.

6.3.1 Short-Term

- **Broader evaluation of MetamEnTh (RQ1, RQ2).** We will extend the empirical experiments with larger developer groups and additional facility managers to validate usability and generality across diverse building types.
- **Expansion of MetamEnTh (RQ1, RQ2).** We will expand **MetamEnTh** to include entities for detailed occupancy modelling and sections within building spaces.
- **Implementation of BIM importers.** We will develop BIM importers to automatically convert files such as IFC into **MetamEnTh** models. This capability is essential for advancing digital twin research, where static building data must be seamlessly integrated with dynamic sensor streams.
- **Implementation of utility methods (RQ1, RQ2).** We will collaborate with researchers and practitioners to implement and incorporate various utility methods commonly used in energy-efficiency related simulation tasks.
- **Expanded API analysis (RQ3).** We will evaluate newer vendor APIs to determine their support for predictive control, demand response, and digital twin integration. We will also explore integrating **MetamEnTh** with third-party middleware to provide more coherent access to data from the built environment.

- **API Data Validation (RQ3).** We will implement data quality indicators, anomaly-detection techniques, and fault-tolerant measures to ensure the accuracy and reliability of data acquired through BMS and IoT APIs.
- **Richer control strategies (RQ4).** We will extend the current binary and PID interfaces to support advanced control strategies, including model-predictive and fuzzy-logic control.

6.3.2 Mid-Term

- **Cross-building pilots (RQ3).** We will demonstrate interoperability across campuses or multi-building portfolios using the five-layered architecture.
- **Integration with digital twins (RQ2, RQ3).** We will link **MetamEnTh** models to live digital twin platforms for real-time monitoring and predictive maintenance.
- **Tooling for adoption (RQ1, RQ2).** We will develop user-friendly editors and plugins to support practitioners without programming expertise.

6.3.3 Long-Term

- **Standardisation (RQ1–RQ4).** We will pursue alignment of **MetamEnTh** with initiatives such as ASHRAE, Brick, and Project Haystack to foster adoption by implementing various importers and exporters. This alignment enables users to export accurate, validated **MetamEnTh** models as ASHRAE 223P or Brick RDF graphs and to convert RDF graphs into MetamEnTh models.
- **Modelling of Intelligent Devices (RQ1).** We will introduce classes and methods to model intelligent devices and their behaviour, enabling users to utilise or enhance their functionality within models.
- **AI-driven optimisation (RQ4).** We will use **MetamEnTh** as the backbone for adaptive, learning-based on/off control strategies.

Together, these directions outline our plan to evolve **MetamEnTh** from a research prototype into a standardised, widely adopted metamodel that not only enables interoperable, intelligent, and sustainable built environments but also strengthens integration with BMS.

References

Abdullah, A., Cross, B., & Aksamija, A. (2014). Whole building energy analysis: A comparative study of different simulation tools and applications in architectural design. In *ACEEE summer study on energy efficiency in buildings* (1–12). ACEEE.

ABI Research. (2025). *ABI Research: Technology Market Research & Intelligence.* https://www.abiresearch.com/. (Accessed: Oct. 21, 2025)

Afram, A., & Janabi-Sharifi, F. (2014). Theory and applications of HVAC control systems–A review of model predictive control (MPC). *Building and Environment*, *72*, 343–355.

Agarwal, Y., Balaji, B., Dutta, S., Gupta, R., & Weng, T. (2011). Managing plug-loads for demand response within buildings. In *Proceedings of 3rd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings.* ACM.

Alan Ruttenberg. (2002). *Basic Formal Ontology (BFO).* Retrieved from https://basic-formal-ontology.org/ (Accessed Oct. 16, 2025)

Allied Market Research. (2023). *Building Management System Market by Component, Application, and Region: Global Opportunity Analysis and Industry Forecast, 2023–2032* (Market Research Report). Author. Retrieved from https://www.alliedmarketresearch.com/building-management-system-market-A12428 (Accessed: Oct. 22, 2025)

Alshammari, K., Beach, T., Rezgui, Y., & Alelwani, R. (2023). Built environment cybersecurity: development and validation of a semantically defined access management framework on a university case study. *Applied Sciences*, *13*(13), 7518.

Amazon Web Services (AWS). (2025). *AWS IoT Core: Cloud Service for IoT Devices.* https://

aws.amazon.com/iot-core/. (Accessed: Oct. 22, 2025)

Anthony Jnr, B., Abbas Petersen, S., Ahlers, D., & Krogstie, J. (2020). API deployment for big data management towards sustainable energy prosumption in smart cities—a layered architecture perspective. *International Journal of Sustainable Energy*, *39*(3), 263–289.

Archidata Inc. (2025). *Archidata — Web-based solution for 2D/3D building data and digital twins.* https://www.archidata.com/en/index. (Accessed: Oct. 21, 2025)

Arthur, R. (2023). A critical analysis of the What3Words geocoding algorithm. *PLOS ONE*, *18*(10). doi: 10.1371/journal.pone.0292491

ASHRAE. (2016). *Addendum m to ANSI/ASHRAE Standard 135-2012: BACnet – A Data Communication Protocol for Building Automation and Control Networks* (Standard Addendum). American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE). Retrieved from https://tinyurl.com/37dfhatu (Accessed: Oct. 23, 2025)

Aué, J., Aniche, M., Lobbezoo, M., & van Deursen, A. (2018). An exploratory study on faults in Web API integration in a large-scale payment company. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (13–22). ACM.

BACnet. (2025). *About BACnet - BACnet Committe.* https://bacnet.org/about/. (Accessed Jul. 10, 2025)

Balaji, B., Bhattacharya, A., Fierro, G., Gao, J., Gluck, J., Hong, D., ... Agarwal, Y. (2016). Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (41–50).

Balaji, B., Bhattacharya, A., Fierro, G., Gao, J., Gluck, J., Hong, D., ... Whitehouse, K. (2018). Brick: Metadata schema for portable smart building applications. *Applied Energy*, *226*, 1273-1292. doi: https://doi.org/10.1016/j.apenergy.2018.02.091

Bandara, S., Yashiro, T., Koshizuka, N., & Sakamura, K. (2016). Towards a standard API design for open services in smart buildings. In *2016 tron symposium (tronshow)* (1–7).

Barbier, F., Henderson-Sellers, B., Opdahl, A. L., & Gogolla, M. (2001). The whole-part relationship in the Unified Modeling Language: A new approach. In *Unified Modeling Language:*

Systems Analysis, Design and Development Issues (186–210). IGI Global Scientific Publishing.

Becerik-Gerber, B., & Kensek, K. (2010). Building information modelling in architecture, engineering, and construction: Emerging research directions and trends. *Journal of Professional Issues in Engineering Education and Practice*, *136*(3), 139–147.

Bhattacharya, A. (2016). *Enabling scalable smart-building analytics* (Ph.D. dissertation). University of California, Berkeley.

Bhattacharya, A., Ploennigs, J., & Culler, D. (2015). Short Paper: Analysing metadata schemas for buildings: The good, the bad, and the ugly. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments* (33–34).

Bill Swan. (2022). *The Language of BACnet.* Retrieved from https://bacnet.org/wp-content/uploads/sites/4/2022/06/The-Language-of-BACnet-1.pdf (Accessed Jul. 10, 2025)

BMS System. (2025). *Understand the Basic Concept of BMS System.* https://bms-system.com/understand-the-basic-concept-of-bms-system/. (Accessed: Oct. 22, 2025)

Bogner, J., Kotstein, S., & Pfaff, T. (2023). Do RESTful API design rules have an impact on the understandability of Web APIs? *Empirical Software Engineering*, *28*(6), 132.

Borgogno, O., & Colangelo, G. (2019). Data sharing and interoperability: Fostering innovation and competition through APIs. *Computer Law & Security Review*, *35*(5), 105314.

Bourreau, P., Chbeir, R., Cardinale, Y., Corchero, A., Lafréchoux, J., Frédérique, D., ... Kallab, L. (2019). BEMServer: An open source platform for building energy performance management. In *EC3 Conference 2019* (Vol. 1, 256–264).

Brian Mulloy. (2011). *RESTful API Design: What About Errors?* Retrieved from https://tinyurl.com/yvj2mdj5 (Accessed Jul. 10, 2025)

BrickSchema. (2025). *brick-example-server: A minimal server implementing a Brick data model.* GitHub Repository. Retrieved from https://github.com/BrickSchema/brick-example-server (Accessed: Oct. 23, 2025)

BuildingSMART International. (2022, Oct). *Industry Foundation Classes (IFC).* Retrieved from

https://technical.buildingsmart.org/standards/ifc (Accessed Sept. 18, 2023)

Butzin, B., Golatowski, F., & Timmermann, D. (2017). A survey on information modelling and ontologies in building automation. In *Iecon 2017-43rd annual conference of the ieee industrial electronics society* (8615–8621).

Campbell, A. T., Coulson, G., & Kounavis, M. E. (1999). Managing complexity: middleware explained. *IT Professional*, *1*(5), 22-28. doi: 10.1109/6294.793667

Chamari, L., Petrova, E., & Pauwels, P. (2022). A web-based approach to BMS, BIM and IoT integration: a case study. In *14th rehva hvac world congress, clima 2022.*

Chen, W., Das, M., Chen, K., & Cheng, J. C. (2020). Ontology-based data integration and sharing for facility maintenance management. In *Construction research congress 2020* (1353–1362).

Compton, M., Barnaghi, P., Bermudez, L., Garcia-Castro, R., Corcho, O., Cox, S., . . . others (2012). The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, *17*, 25–32.

Daniele, S. (2025). Ontologies for the Reconfiguration of Domestic Living Environments: A Systematic Literature Review. *Information*, *16*(9), 752.

DataPARC. (2025). *Process Control Loops: Trending PV, SP, OP and Mode.* DataPARC Blog. Retrieved from https://www.dataparc.com/blog/process-control-loops-trending-pv-sp-op-and-mode/ (Accessed: Oct. 24, 2025)

Dave, B., Buda, A., Nurminen, A., & Främling, K. (2018). A framework for integrating BIM and IoT through open standards. *Automation in construction*, *95*, 35–45.

David Beckett and Tim Berners-Lee. (2014). *Turtle: Terse RDF Triple Language* (Tech. Rep.). W3C Recommendation. Retrieved from https://www.w3.org/TR/turtle/ (Accessed: Oct. 23, 2025)

Dawson-Haggerty, S., Krioukov, A., Taneja, J., Karandikar, S., Fierro, G., Kitaev, N., & Culler, D. (2013). BOSS: Building operating system services. In *10th usenix symposium on networked systems design and implementation (nsdi 13)* (443–457).

Delta Controls. (2025). *Delta Controls enteliWEB BMS.* Retrieved from https://deltacontrols.com/products/enteliweb/ (Accessed Nov. 4, 2024)

DMA Group. (n.d.). *BMS Optimation and Energy Use - DMA Group.* https://dma-group.co.uk/bms-optimisation-energy-use/. (Accessed Oct. 14, 2025)

Donovan, P. (2020). *Three Essential Elements of Next Generation Building Management Systems (BMS)* (White Paper). 35 Rue Joseph Monier, 92500 Rueil-Malmaison, France: Schneider Electric - Science Center.

DQ, Rawlings JB Mayne. (2009). *Model predictive control: theory and design.* Nob Hill Publishing, LLC.

Eclipse Foundation. (2014). *Eclipse Vorto.* Retrieved from https://eclipse.dev/vorto/ (Accessed Jun. 20, 2025)

Eclipse Foundation. (2025). *Eclipse Mosquitto: An open source MQTT broker.* https://mosquitto.org/. (Accessed: Oct. 22, 2025)

Environment and Climate Change Canada. (2025). *Emission Factors and Reference Values for the Federal Greenhouse Gas Offset System.* https://tinyurl.com/36f8s3ef. (Accessed: Oct. 22, 2025)

ETSI. (2023). *ETSI GS CIM 006 V1.2.1 (2023-06).* Retrieved from https://tinyurl.com/2m9tyzzm (Accessed Jun. 20, 2025)

ETSI SmartM2M Technical Committee. (2015). *Smart Applications REFerence Ontology.* Retrieved from https://saref.etsi.org/ (Accessed Jun. 20, 2025)

European Parliament. (2023). *Energy Performance of Buildings Directive.* Retrieved from https://tinyurl.com/29tdvesz (Accessed Jun. 17, 2025)

European Parliament and Council of the European Union. (2021). *Regulation (EU) 2021/1119 of the European Parliament and of the Council of 30 June 2021 establishing a framework to achieve climate neutrality and amending Regulations (EC) No 401/2009 and (EU) 2018/1999 ('European Climate Law').* Official Journal of the European Union, L 243/1. Retrieved from https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32021R1119 (Accessed: Oct. 27, 2025)

Euzenat, J. (2004). An API for ontology alignment. In *International semantic web conference* (698–712).

Facility Executive. (2014). *Optimising Data from Building Management Systems.* Retrieved from

https://tinyurl.com/u9xv3wn4 (Accessed Jul. 10, 2025)

Favre, J.-M. (2005). Foundations of meta-pyramids: Languages vs. metamodels–Episode II: Story of thotus the baboon. In *Proceedings of the Dagstuhl Seminar on Language Engineering for Model-Driven Software Development.* Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Fierro, G. (2019). *Design of an Effective Ontology and Query Processor Enabling Portable Building Applications* (Master's thesis, EECS Department, University of California, Berkeley). Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-29.html

Fierro, G., Koh, J., Agarwal, Y., Gupta, R. K., & Culler, D. E. (2019). Beyond a House of Sticks: Formalising Metadata Tags with Brick. In *Proceedings of the 6th acm international conference on systems for energy-efficient buildings, cities, and transportation* (125–134).

Fitz, T., Theiler, M., & Smarsly, K. (2019). A metamodel for cyber-physical systems. *Advanced Engineering Informatics*, *41*, 100930.

FIWARE Foundation. (2019). *Orion Context Broker API v1 Walkthrough (version 1.12.0).* Orion Context Broker Documentation. Retrieved from https://fiware-orion.readthedocs.io/en/1.12.0/user/walkthrough_apiv1/index.html (Accessed: Oct. 27, 2025)

FIWARE Foundation. (2025). *FIWARE Data Models Documentation.* Read the Docs. Retrieved from https://fiware-datamodels.readthedocs.io/en/stable/index.html (Accessed: Oct. 27, 2025)

Fütterer, J. (2017). *Tuning of PID Controllers within Building Energy Systems* (Unpublished doctoral dissertation). Dissertation, RWTH Aachen University, 2017.

Gabe Fierro and Bharathan Balaji. (2025). *brickschema.* PyPI - The Python Package Index. Retrieved from https://pypi.org/project/brickschema/ (Accessed: Oct. 23, 2025)

Garnier-Moiroux, D., Silveira, F., & Sheth, A. (2013). Towards user identification in the home from appliance usage patterns. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (861–868).

gbXML.org. (2000). *Green Building XML (gbXML) is the language of buildings allowing disparate building design software tools to all communicate with one another.* Retrieved from https://www.gbxml.org/ (Accessed Jul. 14, 2025)

Geng, G., & Geary, G. (1993). On performance and tuning of PID controllers in HVAC systems. In *Proceedings of IEEE International Conference on Control and Applications* (819-824 vol.2). doi: 10.1109/CCA.1993.348229

Gil, S., Oakes, B. J., Gomes, C., Frasheri, M., & Larsen, P. G. (2024). Toward a systematic reporting framework for digital twins: a cooperative robotics case study. *Simulation*, 00375497241261406.

Gilani, S., Quinn, C., & McArthur, J. (2020). A review of ontologies within the domain of smart and ongoing commissioning. *Building and Environment*, *182*, 107099.

Gispert, D. E., Yitmen, I., Sadri, H., & Taheri, A. (2025). Development of an ontology-based asset information model for predictive maintenance in building facilities. *Smart and Sustainable Built Environment*, *14*(3), 740–757.

Google. (2023). *Digital Building.* Retrieved from https://google.github.io/digitalbuildings (Accessed Jun. 20, 2025)

Google. (2023). *Digital Building Ontology.* Retrieved from https://tinyurl.com/mr3p9r9p (Accessed Jun. 20, 2025)

Google, & CNCF. (2025). *gRPC - A high performance, open source universal RPC framework.* https://grpc.io/. (Accessed: Oct. 22, 2025)

Government of Canada. (2022). *Government of Canada.* Retrieved from https://tinyurl.com/h3zh4pvr (Accessed Jun 17, 2025)

Gray, J., & Rumpe, B. (2022). On the relationship between models and ontologies. *Software and Systems Modeling*, *21*(4), 1271–1272.

Guo, M., Xiao, Z., Liu, X., & Zhuge, J. (2023). Discovering and Understanding the Security Flaws of Authentication and Authorization in IoT Cloud APIs for Smart Home. In *International Conference on Security and Privacy in Communication Systems* (205–224). Cham, Switzerland: Springer.

Hammar, K., Wallin, E. O., Karlberg, P., & Hälleberg, D. (2019). The realestatecore ontology. In

International Semantic Web Conference (130–145).

Hamza. (2018). *Understanding the Basic Concept of BMS System.* Retrieved from https://tinyurl.com/5n7ukbfc (Accessed Jun. 13, 2024)

Han, Y., & Kim, W. (2021). Development and validation of building control algorithm energy management. *Buildings*, *11*(3), 131.

Hang-yat, L. A., & Wang, D. (2013). Carrying my environment with me: A participatory-sensing approach to enhance thermal comfort. In *Proceedings of the 5th ACM Workshop on Embedded Systems for Energy-efficient Buildings* (1–8).

Haystack, P. (2011). *Ontology, Project Haystack.* Retrieved from https://project-haystack.org/doc/docHaystack/Ontology (Accessed Jun. 26, 2025)

Haystack, P. (2025). *Haystack Point.* Retrieved from https://tinyurl.com/yc2dftre (Accessed Jul. 2, 2025)

Henderson-Sellers, B. (2011). Bridging metamodels and ontologies in software engineering. *Journal of Systems and Software*, *84*(2), 301–313.

Henze, G. P., Kircher, K. J., & Braun, J. E. (2024). Why has advanced commercial HVAC control not yet achieved its promise? *Journal of Building Performance Simulation*, 1–12.

Honeywell. (2025). *Enterprise Building Integrator.* Retrieved from https://tinyurl.com/2ejs3u3m (Accessed Jul. 10, 2025)

Horridge, M., & Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. *Semantic web*, *2*(1), 11–21.

IBM Cloud. (2025). *IBM Watson IoT Platform.* https://internetofthings.ibmcloud.com/. (Accessed: Oct. 22, 2025)

Jianbo, B., Yuzhe, H., & Guochang, M. (2011). Integrating building automation systems based on web services. *Journal of software*, *6*(11), 2209.

Johnson Controls. (2019). *Metasys Application Programming Interface.* Retrieved from https://on.jci.com/46Przl3 (Accessed Jul. 10, 2025)

Johnson Controls. (2022). *Introducing Metasys Release 12.0 by Johnson Controls.* Retrieved from https://tinyurl.com/2tb687bm (Accessed Jul. 10, 2025)

Johnson Controls. (2025a). *Metasys API License - Licensed Operations.* Retrieved from

https://tinyurl.com/hnrnrxkj (Accessed Jul. 2, 2025)

Johnson Controls. (2025b). *Metasys API v4.* Metasys API Documentation. Retrieved from https://jci-metasys.github.io/api-landing/api/v4 (Accessed: Oct. 22, 2025)

JSON-LD Community Group. (2025). *JSON-LD: A JSON-based Serialisation for Linked Data.* https://json-ld.org/. (Accessed: Oct. 21, 2025)

JSON-RPC Working Group. (2010). *JSON-RPC 2.0 Specification.* https://www.jsonrpc.org/specification. (Accessed: Oct. 22, 2025)

Kensek, Karen. (2014). *Building information modelling*. Routledge.

Khabbazi, A. J., Pergantis, E. N., Premer, L. D. R., Papageorgiou, P., Lee, A. H., Braun, J. E., ... Kircher, K. J. (2025). Lessons learned from field demonstrations of model predictive control and reinforcement learning for residential and commercial HVAC: A review. *arXiv preprint arXiv:2503.05022*.

Kim, D., Lee, J., Do, S., Mago, P. J., Lee, K. H., & Cho, H. (2022). Energy modelling and model predictive control for HVAC in buildings: A review of current research trends. *Energies*, *15*(19), 7231.

Kleiminger, W., Santini, S., & Mattern, F. (2014). Smart heating control with occupancy prediction: how much can one save? In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (947–954).

Kučera, A., & Pitner, T. (2018). Semantic bms: Allowing usage of building automation data in facility benchmarking. *Advanced Engineering Informatics*, *35*, 69–84.

Laakso, M., Kiviniemi, A., et al. (2012). The IFC standard-A review of history, development, and standardization. *Journal of information Technology in Construction*.

Liu, T., Liu, Y., Chen, S., Che, Y., Xu, Z., & Duan, Y. (2014). A user demand and preference profiling method for residential energy management. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (911–918).

Lu, Q., Parlikad, A. K., Woodall, P., Don Ranasinghe, G., Xie, X., Liang, Z., ... Schooling, J. (2020). Developing a digital twin at building and city levels: Case study of West Cambridge

campus. *Journal of Management in Engineering*, *36*(3), 05020004.

Manic, M., Wijayasekara, D., Amarasinghe, K., & Rodriguez-Andina, J. J. (2016). Building energy management systems: The age of intelligent and adaptive buildings. *IEEE Industrial Electronics Magazine*, *10*(1), 25–39.

Marinakis, V., Doukas, H., Karakosta, C., & Psarras, J. (2013). An integrated system for buildings' energy-efficient automation: Application in the tertiary sector. *Applied energy*, *101*, 6–14.

Mars BIM International. (2024). *The Role of BIM in MEP System Lifecycle Sustainability.* Retrieved from https://tinyurl.com/5n7cph6s (Accessed Jul. 21, 2025)

Massé, M. (2011). *REST API Design Rulebook*. Sebastopol, CA, USA: O'Reilly Media, Inc.

McGibney, A., Rea, S., & Ploennigs, J. (2016). Open BMS-IoT driven architecture for the internet of buildings. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society* (7071–7076).

McLaughlin, S., McDaniel, P., & Aiello, W. (2011). Protecting consumer privacy from electric load monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security* (87–98).

McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, *12*(2), 153–157.

Microsoft Azure. (2025). *Azure IoT Hub: Cloud-scale IoT device management.* https://azure.microsoft.com/en-ca/products/iot-hub. (Accessed: Oct. 22, 2025)

Moe, A., & Gibbs, P. (2023). *Equitable Electrification Analysis for Existing Buildings in Richmond, CA* (Tech. Rep.). National Renewable Energy Laboratory (NREL), Golden, CO (United States).

Moghadam, T. T., Morales, C. E. O., Zambrano, M. J. L., Bruton, K., & O'Sullivan, D. T. (2023). Energy efficient ventilation and indoor air quality in the context of COVID-19—a systematic review. *Renewable and Sustainable Energy Reviews*, *182*, 113356.

Mozilla Contributors. (2025). *Using server-sent events.* Mozilla Developer Network (MDN) Web Docs. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events (Accessed: Oct. 23, 2025)

Nilesh Mayani. (2025). *How Digital Twins Are Transforming MEP Design and Operations.* Retrieved from https://tinyurl.com/4asmrtcw (Accessed Jul. 21, 2025)

NIST. (2025). *Control Network.* NIST Glossary. Retrieved from https://csrc.nist.gov/glossary/term/control_network (Accessed: Oct. 22, 2025)

Norman, G. (2010). Likert scales, levels of measurement and the "laws" of statistics. *Advances in health sciences education*, *15*(5), 625–632.

OpenJS Foundation. (2025). *Node-RED: Low-Code Programming for Event-Driven Applications.* https://nodered.org/. (Accessed: Oct. 27, 2025)

OWASP. (2025). *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet.* OWASP Cheat Sheet Series. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html (Accessed: Oct. 23, 2025)

Paige, R. F., Kolovos, D. S., & Polack, F. A. (2014). A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, *96*, 396-416. (Selected Papers from the 5th International Conference on Software Language Engineering (SLE 2012)) doi: https://doi.org/10.1016/j.scico.2014.05.007

Pauen, N. P. M., Schlütter, D., Siwiecki, J., Frisch, J., & van Treeck, C. A. (2020). Integrated representation of building service systems: topology extraction and TUBES ontology. *Bauphysik: Wärme, Feuchte, Schall, Brand, Licht, Energie, Klima*, *42*, 299–305. doi: 10.1002/bap.202000028

Peter D. Kazarinoff. (2025). Positional and Keyword Arguments. In *Problem Solving with Python.* OTexts. Retrieved from https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments/ (Accessed: Oct. 27, 2025)

Peter Yefi. (2024a, Oct). *Metamenth case study uml designs.* https://www.ptidej.net/downloads/replications/iotj24a/photos/. (Accessed: Oct. 20, 2025)

Peter Yefi. (2024b, Oct). *Metamenth subsystem — documentation for the tools4cities-metamenth metamodel.* https://github.com/ptidejteam/tools4cities-metamenth/wiki. (Accessed: Oct. 20, 2025)

Peter Yefi. (2025a). *bms-api-mockup: A mock API for Building Management Systems.* GitHub Repository. Retrieved from https://github.com/ptidejteam/bms-api-mockup (Accessed: Oct. 23, 2025)

Peter Yefi. (2025b). *Metamenth — a python package for object-oriented modelling of building systems.* urlhttps://pypi.org/project/metamenth/. (Accessed: Oct. 20, 2025)

Peter Yefi. (2025c). *tools4cities-metamenth: Object-oriented metamodel for energy things.* https://github.com/ptidejteam/tools4cities-metamenth. (Accessed: Oct. 20, 2025)

Ploennigs, J., Chen, B., Schumann, A., & Brady, N. (2013). Exploiting generalised additive models for diagnosing abnormal energy use in buildings. In *Proceedings of the 5th ACM workshop on embedded systems for energy-efficient buildings* (1–8).

PR Newswire. (2024). *Siemens, Schneider Electric, and Honeywell Lead ABI Research's Energy Management System Vendors Competitive Ranking.* https://tinyurl.com/btmhwv8x. (Accessed: Oct. 22, 2025)

Pritoni, M., Paine, D., Fierro, G., Mosiman, C., Poplawski, M., Saha, A., . . . Granderson, J. (2021). Metadata schemas and ontologies for building energy applications: A critical review and use case analysis. *Energies*, *14*(7), 2024.

Priyanka. (2025). *Web Services vs APIs - What are they and how are they different?* https://tinyurl.com/43u9r8k9. (Accessed Jul. 10, 2025)

Project Haystack. (2025a). *Applying Haystack Tagging for a Sample Building.* Project Haystack Reference Implementation. Retrieved from https://bit.ly/44yaBrd (Accessed: Jul. 2, 2025)

Project Haystack. (2025b). *Haystack HTTP API Specification.* Project Haystack Documentation. Retrieved from https://project-haystack.org/doc/docHaystack/HttpApi (Accessed: Oct. 23, 2025)

Project Haystack. (2025c). *Haystack JSON Format Specification v3.* Project Haystack Documentation. Retrieved from https://project-haystack.org/doc/docHaystack/Json/#v3 (Accessed: Oct. 23, 2025)

Project Haystack. (2025d). *Haystack Trio Format Specification.* Project Haystack Documentation. Retrieved from https://project-haystack.org/doc/docHaystack/Trio (Accessed: Oct. 23, 2025)

Project Haystack. (2025e). *Space - Project Haystack.* Project Haystack Documentation. Retrieved from https://tinyurl.com/3furkxc2 (Accessed: Jul. 2, 2025)

Project Haystack Organization. (2025). *Project haystack ontology (tagging and metadata model).* https://project-haystack.org/doc/docHaystack/Ontology. (Accessed: Oct. 21, 2025)

QUDT. (2014). *QUDT: Quantities, Units, Dimensions and Data Types.* https://qudt.org/. (Accessed: Oct. 27, 2025)

Quinn, C., Shabestari, A. Z., Misic, T., Gilani, S., Litoiu, M., & McArthur, J. (2020). Building automation system-BIM integration using a linked data structure. *Automation in Construction*, *118*, 103257.

Rasmussen, M. H., Pauwels, P., Hviid, C. A., & Karlshoj, J. (2017). Proposing a central AEC ontology that allows for domain-specific extensions. In *Joint Conference on Computing in Construction* (Vol. 1, 237–244).

Ratiu, D., Feilkas, M., & Jurjens, J. (2008). Extracting domain ontologies from domain-specific APIs. In *2008 12th european conference on software maintenance and reengineering* (203–212).

Regulvar. (2025). *REGULVAR - Variable Speed Drive Specialists.* https://www.regulvar.com/en/. (Accessed: Oct. 20, 2025)

Reinisch, C., Kofler, M. J., & Kastner, W. (2010). ThinkHome: A smart home as digital ecosystem. In *4th IEEE International Conference on Digital Ecosystems and Technologies* (256-261). doi: 10.1109/DEST.2010.5610636

Rijgersberg, H., Van Assem, M., & Top, J. (2013). Ontology of units of measure and related concepts. *Semantic Web*, *4*(1), 3–13.

Rob J Hyndman and George Athanasopoulos. (2018). ARIMA models. In *Forecasting: Principles and Practice.* OTexts. Retrieved from https://otexts.com/fpp2/arima.html (Accessed: Oct. 23, 2025)

Schachinger, D., Kastner, W., & Gaida, S. (2016). Ontology-based abstraction layer for smart grid interaction in building energy management systems. In *2016 ieee international energy conference (energycon)* (1–6).

Schneider, G. F., Pauwels, P., & Steiger, S. (2017). Ontology-Based Modelling of Control Logic in Building Automation Systems. *IEEE Transactions on Industrial Informatics*, *13*(6), 3350-3360. doi: 10.1109/TII.2017.2743221

Schneider Electric. (2025a). *Analytics Building Energy Modeling API Reference.* Schneider Electric Exchange Developer Portal. Retrieved from https://tinyurl.com/2v4kkj4s (Accessed: Oct. 23, 2025)

Schneider Electric. (2025b). *EcoStruxure Building Operation Software.* https://www.se.com/ca/en/product-range/62111-ecostruxure-building-operation-software/#overview. (Accessed: Oct. 22, 2025)

Schneider Electric. (2025c). *EcoStruxure Building Operations Smart Connector.* Schneider Electric Exchange Product Page. Retrieved from https://exchange.se.com/develop/products/40647/ecostruxure-building-operations-smart-connector (Accessed: Oct. 24, 2025)

Schneider Electric. (2025d). *EcoStruxure Energy and Sustainability Scoring API Reference.* Schneider Electric Exchange Developer Portal. Retrieved from https://tinyurl.com/2xhns9x8 (Accessed: Oct. 23, 2025)

Schneider Electric, USA. (2013). *What is Modbus and how does it work? | Schneider Electric, USA.* https://tinyurl.com/ncpdkajh. (Accessed Jul. 10, 2025)

Shahzad, M., Shafiq, M. T., Douglas, D., & Kassem, M. (2022). Digital twins in built environments: an investigation of the characteristics, applications, and challenges. *Buildings*, *12*(2), 120.

Sharpe, D. M. (1987). Microclimatology. In *Climatology* (572–581). Springer. doi: 10.1007/0-387-30749-4_115

Siemens. (2025a). *Auth0 OAuth 2.0 Token Endpoint for Siemens Building Technologies.* OAuth 2.0 Service Endpoint. Retrieved from https://siemens-bt-015.eu.auth0.com/oauth/token (Accessed: Oct. 23, 2025)

Siemens. (2025b). *Building Structure API Reference.* Siemens Developer Documentation.

Retrieved from https://developer.siemens.com/building-x-openness/api/building-structure/api-reference.html (Accessed: Oct. 22, 2025)

Siemens. (2025c). *Desigo Building Automation.* https://www.siemens.com/global/en/products/buildings/desigo-building-automation.html. (Accessed: Oct. 20, 2025)

Siemens. (2025d). *Desigo CC - Building Management Station.* https://www.siemens.com/global/en/products/buildings/desigo-building-automation/building-management/desigo-cc.html. (Accessed: Oct. 22, 2025)

Siemens. (2025e). *Desigo Optic - Building Performance Software.* https://www.siemens.com/us/en/products/buildingtechnologies/automation/desigo-optic.html. (Accessed: Oct. 22, 2025)

Siemens. (2025f). *Geometry API Overview.* Siemens Developer Documentation. Retrieved from https://developer.siemens.com/building-x-openness/api/geometry-api/overview.html (Accessed: Oct. 22, 2025)

Siemens. (2025g). *Lifecycle Twin API Overview.* Siemens Developer Documentation. Retrieved from https://developer.siemens.com/building-x-openness/api/lifecycletwin-api/overview.html (Accessed: Oct. 22, 2025)

Siemens. (2025h). *Lifecycle Twin - Siemens Xcelerator.* https://xcelerator.siemens.com/global/en/products/buildings/building-x/solutions/applications/lifecycle-twin.html. (Accessed: Oct. 22, 2025)

Siemens. (2025i). *Siemens Building X Sandbox Access Request Form.* Online Form. Retrieved from https://bit.ly/4eUp2dg (Accessed: Oct. 23, 2025)

Siemens Building Technologies. (2018, September). *Desigo insight – operating the management station (volume 1), v6.0 sp2.* https://sid.siemens.com/v/u/A6V10362222. (Accessed: Oct. 21, 2025)

Simon Steyskal and Dimitris Kontokostas. (2017). *Shapes Constraint Language (SHACL)* (Tech. Rep.). W3C Recommendation. Retrieved from https://www.w3.org/TR/shacl/ (Accessed: Oct. 27, 2025)

Stavropoulos, T. G., Vrakas, D., Vlachava, D., & Bassiliades, N. (2012). BOnSAI: a smart building

ontology for ambient intelligence. In *Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics* (1–12).

Stolk, S., & McGlinn, K. (2020). Validation of IfcOWL datasets using SHACL. In *LDAC* (91–104).

Sturzenegger, D., Gyalistras, D., Morari, M., & Smith, R. S. (2012). Semi-automated modular modelling of buildings for model predictive control. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings* (99–106).

Sullivan, G. M., & Artino Jr, A. R. (2013). Analyzing and interpreting data from Likert-type scales. *Journal of graduate medical education*, *5*(4), 541.

Tang, S., Shelden, D. R., Eastman, C. M., Pishdad-Bozorgi, P., & Gao, X. (2019). A review of building information modelling (BIM) and the internet of things (IoT) devices integration: Present status and future trends. *Automation in Construction*, *101*, 127–139.

The Things Network. (2025). *What is LoRaWAN?* The Things Network Documentation. Retrieved from https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/ (Accessed: Oct. 27, 2025)

Thomas, D. (2003). Uml-unified or universal modeling language. *Journal of Object Technology*, *2*(1), 7–12.

Thomas Bush. (2023). *What is the difference between web service and API.* https://tinyurl.com/bdf932vs. (Accessed Jul. 10, 2025)

Tridium. (2025). *Niagara Framework.* https://www.tridium.com/us/en/Products/niagara. (Accessed: Oct. 2r, 2025)

UN CTCN. (2001). *Building Energy Management Systems (BEMS).* Retrieved from https://www.ctc-n.org/technologies/building-energy-management-systems-bems (Accessed: Jun. 17, 2025)

US EIA. (2018). *U.S. Energy Information Administration - EIA - independent statistics and analysis.* Retrieved from https://tinyurl.com/ypzan8nf (Accessed: Jun. 17, 2025)

VBIS. (2025). *Classification Structure.* https://vbis.com.au/classification-structure. (Accessed: Oct. 23, 2025)

W3C RDF Working Group. (2023). *RDF 1.2 Concepts and Abstract Syntax.* https://www.w3.org/TR/rdf12-concepts/. (Accessed: Oct. 21, 2025)

W3C Semantic Web Wiki. (2025). *Web Ontology Language (OWL).* https://www.w3.org/2001/sw/wiki/OWL. (Accessed: Oct. 21, 2025)

W3C SPARQL Working Group. (2013). *SPARQL 1.1 Query Language.* https://www.w3.org/TR/sparql11-query/. (Accessed: Oct. 21, 2025)

WAGO. (2025). *LONWORKS - Fieldbus Standard for Building Automation.* https://www.wago.com/global/lonworks. (Accessed Jul. 10, 2025)

Walczyk, G., & Ożadowicz, A. (2024). Building information modelling digital twins for functional and technical design of smart buildings with distributed iot networks—review and new challenges discussion. *Future Internet*, *16*(7), 225.

Wang, C., Zhang, L., & Yan, W. (2024). Enhancement and validation of IfcOWL ontology based on Shapes Constraint Language (SHACL). *Automation in Construction*, *160*, 105293.

Wang, Y., Kuckelkorn, J., & Liu, Y. (2017). A state of art review on methodologies for control strategies in low energy buildings in the period from 2006 to 2016. *Energy and Buildings*, *147*, 27–40.

Wei, T., Wang, Y., & Zhu, Q. (2017). Deep reinforcement learning for building HVAC control. In *Proceedings of the 54th annual design automation conference 2017* (1–6).

Wikipedia. (2023, Jan). *Cogeneration.* Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Cogeneration

Wikipedia Contributors. (2025). *Constructor (object-oriented programming).* Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming) (Accessed: Oct. 27, 2025)

Wikipedia contributors. (2025). *Solar Energy.* Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Solar_energy (Accessed August 2025)

XML-RPC Working Group. (2025). *XML-RPC Specification.* https://xmlrpc.com/. (Accessed: Oct. 22, 2025)

Yonglin, L., Zhi, Z., & Qun, L. (2020). An ontological metamodelling framework for semantic simulation model engineering. *Journal of Systems Engineering and Electronics*, *31*(3), 527–538.

Zhang, Y.-Y., Kang, K., Lin, J.-R., Zhang, J.-P., & Zhang, Y. (2020). Building information

modelling–based cyber-physical platform for building performance monitoring. *International Journal of Distributed Sensor Networks*, *16*(2), 1550147720908170.

Zheng, Y., Törmä, S., & Seppänen, O. (2021). A shared ontology suite for digital construction workflow. *Automation in Construction*, *132*, 103930.

Appendix A

Table A.1: Review of literature, identified gaps and their relations with the four research questions

Paper	Purpose	Related RQs	Research Gap
A review of ontologies within the domain of smart and ongoing commissioning (Gilani et al., 2020)	Comprehensive review of SOCx ontologies to identify their limitations and make recommendations for bridging the identified weaknesses	RQ1.1, RQ1.2	Reviewed ontologies lacked built-in validation, comprehensive documentation and real-world validation
Validation of IfcOWL datasets using SHACL (Stolk & McGlinn, 2020)	Identified OWL's limitations in enforcing constraints and introduced SHACL to validate OWL-based ontologies	RQ1.1, RQ1.2	SHACL validation is post-hoc and requires maintaining a validation model in addition to the ontology. Modellers cannot enforce validation

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Enhancement and validation of IFCOWL ontology based on Shapes Constraint Language (SHACL) (C. Wang et al., 2024)	Address a key limitation in ifcOWL validation by incorporating SHACL to enforce IFC's WHERE rules	RQ1.1, RQ1.2	SHACL validation is post-hoc and requires maintaining a validation model in addition to the ontology. Modellers cannot enforce validation
A survey on information modelling and ontologies in building automation (Butzin et al., 2017)	Analysed the scope of BAS semantic information models, focusing on how they provide contextual information and vocabulary to describe building systems and their operations	RQ1.1 RQ1.2, RQ2	Review highlighted inadequate description of device function or context by BAS ontologies
A metamodel for cyber-physical systems (Fitz et al., 2019)	Investigated metamodeling approaches for describing cyber-physical systems (CPSs) used in structural health monitoring (SHM) and control systems	RQ1.1, RQ1.2	Proposed metamodel lacks domain-specific constructs for representing building systems and their operational contexts
Proposing a central AEC ontology that allows for domain-specific extensions Rasmussen et al. (2017)	Proposed the Building Topology Ontology (BOT) for modelling the topological structure of buildings with extensibility to domain-specific ontologies	RQ1.1, RQ1.2	Lacks the semantic richness required to model the complex relationships and behaviours of MEP systems

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	**Purpose**	**Related RQs**	**Research Gap**
Short Paper: Analysing metadata schemas for buildings: The good, the bad, and the ugly (Bhattacharya et al., 2015)	Evaluated the limitations of three metadata schemas, Project Haystack, Industrial Foundation Class, and the Semantic Sensor Web, in representing the contextual, spatial, and functional relationships among sensors in smart buildings	RQ1.1, RQ2	The schemas fall short in providing comprehensive, adaptable models for MEP systems. The findings support extensibility through subclassing and property addition
Brick: Metadata schema for portable smart building applications (Balaji et al., 2018)	Proposed Brick, a semantic ontology designed to provide a unified schema for representing metadata in buildings	RQ1.1, RQ2	Brick inherits the limitations of RDF/OWL-based representations, which do not have a built-in validation mechanism. There is the possibility of creating flawed entity relationships
Beyond a House of Sticks: Formalising Metadata Tags with Brick (Fierro et al., 2019)	Proposed Brick+, an extension of the Brick ontology, to address semantic ambiguities and enhance expressiveness in modelling building metadata, with formalised class hierarchies and relationships	RQ1.1	Brick+ remains limited by RDF/OWL constraints, lacking built-in validation and the capacity to model system behaviour

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Metadata schemas and ontologies for building energy applications: A critical review and use case analysis (Pritoni et al., 2021)	Reviewed five major building-related ontologies (BOT, SSN/SOSA, SAREF, REC, and Brick) using energy-related use cases to evaluate their coverage and suitability for building operations	RQ1.1, RQ2	The ontologies lacked support for concepts such as control devices, spatial zoning, and equipment properties, limiting their effectiveness for energy-efficient applications
ETSI GS CIM 006 V1.2.1 (2023-06) (ETSI, 2023)	Developed NGSI-LD, a cross-domain information model that uses a property graph approach to model structural, functional, spatial, and temporal aspects of entities, enhancing RDF with metadata-enriched nodes and relationships	RQ1.1, RQ2	NGSI-LD interoperability with RDF requires complex reification strategies, making it difficult to use in operational building contexts
Haystack Tagging Ontology (HTO)(Haystack, 2011)	Proposes a flexible tagging framework for describing building systems and data points using standardised vocabularies to promote semantic interoperability across heterogeneous BMS	RQ1.1, RQ2	HTO lacks strict semantic enforcement, leading to potential ambiguity and misapplication of tags. This absence of validation limits its reliability for structured modelling in building systems

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Digital Buildings (Google, 2023)	Introduces a semantically rich modelling framework for structured building metadata using the Digital Buildings Configuration Language (DBCL) and validation tools	RQ1.1, RQ1.2, RQ2	It relies on external configuration files and scripts for validation, which increases complexity and hinders maintainability. Validation logic is not inherently part of the ontology
Smart Applications REFerence Ontology (ETSI SmartM2M Technical Committee, 2015)	Provides standardised, reusable components for modelling smart devices, categorised into sensing, actuating, and metering functionalities	RQ1.1	SAREF limited categories can constrain accurate representation of complex or novel sensor metadata in building contexts
Eclipse Vorto (Eclipse Foundation, 2014)	Provides a domain-specific language (DSL) and supporting tools for modelling IoT devices in a platform-independent way using reusable abstractions like information models and function blocks	RQ1.1	Vorto lacks built-in semantics for modelling the spatial and behavioural complexity of building systems

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
API deployment for big data management towards sustainable energy prosumption in smart cities-a layered architecture perspective (Anthony Jnr et al., 2020)	Proposed a multi-layered architecture to manage and integrate energy data from heterogeneous sources in smart city environments	RQ3.1, RQ3.2	The proposed architecture remains conceptual and was not validated with real-world smart city deployments
An API for ontology alignment (Euzenat, 2004)	Proposed the Alignment API to support semantic interoperability by aligning heterogeneous OWL-based ontologies, enabling discovery of relationships between ontology concepts	RQ3.2	Limited to static ontology schemas. Lacks integration with real-time building data or APIs, restricting its applicability to operational building management scenarios
Extracting domain ontologies from domain specific APIs (Ratiu et al., 2008)	Presented a method for extracting domain ontologies by identifying structural and naming similarities across APIs in the same domain	RQ3.2	Has not been applied to BMS-specific or real-time data interactions, limiting its relevance for operational use in building management systems

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Ontology-based abstraction layer for smart grid interaction in building energy management systems (Schachinger et al., 2016)	Developed an OWL 2-based ontology to standardise communication between smart grids and BEMS, integrating agents, scenarios, communication technologies, and grid structures	RQ3.2	Its focus on smart grid integration does not extend to interoperability challenges within commercial BMS APIs
The OWL API: A Java API for owl ontologies (Horridge & Bechhofer, 2011)	Developed the OWL Java API for programmatic creation and manipulation of OWL ontologies, supporting multiple syntaxes, modular reasoning, and ontology validation with a reasoner interface	RQ3.2	Does not address interaction with operational building systems or dynamic integration with real-time BMS data
Integrating building automation systems based on web services (Jianbo et al., 2011)	Proposed Web services-based integration of heterogeneous BAS subsystems with enterprise applications, defining roles and tasks; validated by exposing BACnet objects for real-time data access	RQ3.1, RQ3.2	Does not evaluate BMS APIs capabilities for integration with enterprise applications

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Towards a standard API design for open services in smart buildings (Bandara et al., 2016)	Addressed heterogeneity in IoT API specifications across manufacturers by proposing device and context APIs to distinguish static attributes and dynamic states, linking devices to spaces and occupants	RQ3.1, RQ3.2	The work is limited to generic IoT device APIs. Does not include BMS APIs
Semantic BMS: Allowing usage of building automation data in facility benchmarking (Kučera & Pitner, 2018)	Proposed a semantic approach extending the SSN ontology to integrate BAS data with BIM and facility management, including a semantic BMS middleware that enhances data accessibility and supports complex queries for benchmarking	RQ3.1, RQ3.2	The approach depends on extending existing ontologies, which may limit flexibility when adapting to highly domain-specific use cases or when modelling dynamic operational data in real-time
BEMServer: An open source platform for building energy performance management (Bourreau et al., 2019)	Presented BEMServer, an open-source BEMS integrating data analytics and visualisation with a semantic layer based on OntoH2G ontology to enhance data interpretability and standardisation	RQ3.1, RQ3.2	The approach lacks evaluation of OntoH2G's flexibility for comprehensive MEP modelling and does not explore real-time control support

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
An integrated system for buildings' energy-efficient automation: Application in the tertiary sector (Marinakis et al., 2013)	Proposed an integrated system for energy efficiency using simulation-based optimisation and Dupline BAS to monitor core functions like lighting and temperature	RQ3.1, RQ3.2	Tight coupling of hardware and software limits portability and scalability; lack of standardised interfaces and semantic abstraction hinders interoperability and broader deployment
A framework for integrating BIM and IoT through open standards (Dave et al., 2018)	Developed a platform integrating BIM and IoT to support unified access and API-based third-party application development at Aalto University	RQ3.2	The proposed system suffers from data heterogeneity across BMS, highlighting the need for semantic abstraction layers to enable unified and system-level reasoning
A review of building information modelling (BIM) and the internet of things (IoT) devices integration: Present status and future trends (Tang et al., 2019)	Reviewed BIM–IoT integration strategies, identifying application domains and integration methods, including BIM APIs, RDF, and relational databases	RQ2, RQ3.2	The hybrid use of RDF for static building semantics and relational databases for dynamic data often leads to fragmented, tightly coupled architectures with high complexity

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Building automation system-BIM integration using a linked data structure (Quinn et al., 2020)	Explored integration of IoT sensor data with FM-BIM using visual programming and two mapping strategies—direct naming convention and ontology-based alignment	RQ3.2	While ontology-based mapping improves semantic richness, the reliance on manual methods and tool-specific workflows limits scalability and automation, highlighting
Built environment cybersecurity: development and validation of a semantically defined access management framework on a university case study (Alshammari et al., 2023)	Proposed an ontology-based framework for access management in built environments, integrating CPS, IoT, BIM, and digital twins to enhance interoperability and security	RQ4	Does not address control logic integration through encapsulated behaviours and system interactions
BOSS: Building operating system services (Dawson-Haggerty et al., 2013)	Introduced BOSS, a framework to reduce BMS fragmentation using components like HAL, control transaction managers, and time series services for device interaction and data access	RQ4	Improves interface-level interoperability but lacks a structured, model-based representation of spatial and behavioural system relationships

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Open BMS-IoT driven architecture for the internet of buildings (McGibney et al., 2016)	Proposed OpenBMS, an open IoT-based architecture designed to simplify integration of heterogeneous building systems and reduce vendor lock-in	RQ4	Augments traditional BMS to facilitate interoperability through integration interfaces.
A shared ontology suite for digital construction workflow (Zheng et al., 2021)	Introduced DiCon, a suite of digital construction ontologies designed to standardise and integrate information across construction workflows	RQ4	DiCon's reliance on static RDF outputs and lack of runtime behavioural modelling limits applicability in operational environments
Developing a digital twin at building and city levels: Case study of West Cambridge campus (Lu et al., 2020)	Presented a digital twin framework integrating BIM, IoT, and simulation to support building performance monitoring and decision-making	RQ4	The framework lacks mechanisms for executing control actions via BMS APIs and embeds behavioural models within simulation tools
Building information modelling–based cyber-physical platform for building performance monitoring (Zhang et al., 2020)	Proposed a scalable platform integrating BIM with cyber-physical systems for real-time performance monitoring and analysis, using a four-layer architecture	RQ4	While the platform supports unified monitoring and forecasting through static and dynamic data integration, it does not extend to active control

Review of literature, identified gaps and their relations with the four research questions - continued from the previous page

Paper	Purpose	Related RQs	Research Gap
Ontology-Based Modelling of Control Logic in Building Automation Systems (Schneider et al., 2017)	Proposed an ontology-based approach, named CTRLont, to model control logic within building automation systems	RQ4	The CTRLont Ontology only focuses on the semantic representation of control logic. It does not provide support for execution.

```
1
2 class BuildingCreator:
3   def __init__(self, file_path: str, dynamic_data_reader):
4       self.structure = BuildingStructure()
5       # load building structural json data
6       path = Path(__file__).resolve().parent.parent / file_path
7       self.structure.load_building_data(path)
8       self.dynamic_data = dynamic_data_reader
9       self.building = None
10
11  def add_building_envelope(self, envelope_processor, envelope_name):
12    """
13    Creates an envelope of a building
14    :param envelope_processor: an object that reads and processes building
      envelope data
15    :param envelope_name: the name of the building envelope
16    :return:
17    """
18    envelope = Envelope(envelope_name)
19    # envelope layers data:
20    cover_block_layers = envelope_processor.process_cover_block_layers()
21    # add covers to the various building envelopes
```

Figure A.1: A UML instance of selected sections of the EV building showing various entities like building, floors, rooms, open spaces, meter, weather station, weather data, sensors, zones, and ventilation ducts

Figure A.2: A UML instance model of selected sections of the MB/JMSB Building, highlighting additional entities—such as dampers, variable air volume boxes, baseboard heaters, actuators, solar PV panels, setpoints, and duct connections—not modelled in the EV Building use case

```
22        cover_block_occurrences = \
23            envelope_processor.process_cover_block_occurrence()
24        # material properties
25        building_material_properties = \
26            envelope_processor.process_building_materials()
27        # roof layers
28        roof_layers = envelope_processor.process_building_roof()
29        for floor, block_cover in cover_block_occurrences.items():
30           floor_number = int(floor.split()[-1])
31           for block_details in block_cover:
32               # retrieve layer details
33               layers_data = cover_block_layers[block_details[0]]
34               # extract unique cover type
35               unique_cover_types = {details[5] for details in layers_data}
36               covers = [self._create_cover(cover_type, block_details[2], \
37                    floor_number, layers_data, building_material_properties) \
38                    for cover_type in unique_cover_types]
39               # create cover blocks with the layers
40               for i in range(block_details[1]):
41                   for cover in covers:
42                       envelope.add_cover(cover)
43       # create and add roofs
44       lower_roof, upper_roof = self._create_roof(roof_layers,
          building_material_properties)
45       envelope.add_cover(lower_roof)
46       envelope.add_cover(upper_roof)
47       self.building.add_envelope(envelope)
```

Listing A.1: *add_building_envelope* method in the *BuildingCreator* class that uses an object of *BuildingEnvelopeProcessor*